UNIVERSITÀ DEGLI STUDI DI MILANO
Facolta di Scienze e Tecnologie
Corso di Laurea Magistrale in Sicurezza Informatica

Time-Lock Encryption: una soluzione basata sulla crittografia ellittica e blockchain

Relatore: Prof. Stelvio Cimato

Tesi di:
Francesco Mancuso
Matricola: 942409

Anno Accademico 2020 – 2021

Dedicato alla mia famiglia

Ringraziamenti

Per la redazione di questo lavoro ci terrei a ringraziare prima di tutto il professore e relatore Stelvio Cimato per avemi dedicato il suo tempo prezioso e per avermi dato la possibilità di poter essere qui oggi. Si è dimostrato una persona gentile e sempre disponibile per ogni chiarimento.

Ringrazio la mia famiglia, in particolare mia madre, che in questi anni mi ha supportato e sopportato. Ha creduto in me quando a nessun'altro importava farlo.

Un importante ringraziamento va a Simone, una persona speciale entrata nella mia vita da poco che rende ogni momento speciale .

Indice

Capitolo 1

Introduzione

1.1 Obiettivo

Con questo lavoro di tesi, è mia intenzione effettuare un'analisi di particolari sistemi di cifratura, denominati time-lock encryption system, teorizzati e sviluppati negli anni scorsi e recenti. Sarà quindi mio preciso obiettivo effettuare una panoramica dettagliata di ogni protocollo evidenziandone pregi e difetti. Soprattutto, è di fonamentale importanza verificarne l'effettiva efficacia, in relazione alle proprietà della time-lock encryption e alla situazione computazionale odierna. Tra tutti i modelli presentati, ne verrà proposto uno in grado di garantire la maggior parte delle suddette caratteristiche. Sarà quindi un sistema capace di rendere una certa informazione inaccessibile fino al raggiungimento di una deadline specificata. Di questa nuova proposta verrano forniti i principali punti di forza, verrà sviluppata un'implementazione originale, verrà valutata la sua reale efficacia e verranno proposte eventuali modifiche e miglioramenti atti a risolverne le debolezze.

1.2 Conoscenze preliminari

1.2.1 **Definizione**. Il time-lock encryption è un metodo di cifratura atto a rendere un messaggio innaccessibile, decifrabile solo dopo che un certo istante di tempo è trascorso.

1.2.2 **Definizione**. Le proprietà chiave ([8]) che un time-lock encryption system può presentare simultaneamente sono:

- Non-interactive: non è richiesto che il mittente del messaggio sia in grado di effettuare la decifratura dello stesso.
- No trusted setup: non è richiesto che il mittente del messaggio sia obbligato a fidarsi di una terza parte per mantenere le chiavi di decifratura fino al trascorrere dell'istante di tempo deciso.
- No resource restrictions: le parti interessate alla decifratura del messaggio non devono essere obbligati a performare computazioni complesse prima dello trascorrere del tempo.

In realtà, quest'ultima proprietà, come vedremo più avanti, non si applica a tutti i sistemi di cifratura basati sui puzzle, ma solo a quelli basati su agenti e witness encryption.

1.3 Stato dell'arte

I sistemi nati per garantire la time-lock encryption sono stati classificati principalmente in due categorie: quelli basati sulla presenza di agenti e quelli costruiti sull'uso dei puzzle. Quelli facente parte della prima categoria nacquero oltre 30 anni fa. Già nel 1979 Shamir definì in [2] un approccio basato sugli agenti e sulla condivisione di un segreto, ripartito in frammenti tra tutti i partecipanti al protocollo. Solo successivamente, in [1], venne ampliato il progetto di Shamir includendo la gestione del tempo direttamente dagli agenti. Questo protocollo è chiamato Key Sharing. Nel 2006, in [3], venne implementata una variazione del sistema precedente, chiamato Time-Lapse Cryptography, basato sul cifrario asimmetrico ElGamal. Infine è stato proposto il Conditional Oblivious Transfer ([6]) che trae le caratteristiche principali dall'originale Oblivious Transfer di Rabin nel 1981.
Per quanto riguarda la seconda categoria, i puzzle sono definiti come problemi computazionali che richiedono un certo ammontare di tempo per essere risolti. Questo tempo deve essere il più vicino possibile alla differenza tra il tempo corrente e l'istante di tempo futuro dopo il quale il messaggio può essere decifrato. Abbiamo una prima modalità, proposta in [1], basata sull'uso dell'algoritmo di cifratura RC5 con chiave modulabile a seconda delle caratteristiche del destinatario del messaggio. Nello stesso documento ([1]) è stato sviluppato un problema computazione costruito sul calcolo ripetuto di quadrati. Si tratta di un sistema impossibile da parallelizzare e di conseguenza non riducibile.
Infine, di recente, nascono due nuovi sistemi implementati in un ambiente con Blockchain: il time-lock encryption system basato su witness encryption e quello basato su curve ellittiche e ElGamal.
Tutti i sistemi appena citati verranno descritti e analizzati in questo lavoro. Inoltre verrà fornita un'implementazione del sistema che utilizza le curve ellittiche, ritenuto il più promettente.

1.4 Applicazioni della time-lock encryption

La time-lock encryption è sempre stato un sistema più teorico che pratico che non ha avuto molte applicazioni in questi anni. Di seguito verrano fatti alcuni possibili esempi che potrebbero beneficiare di un sistema di cifratura a tempo.

1.4.1 Sistema di voti

Uno dei sistemi più ambiziosi riguarda l'implementazione di una sorta di piattaforma e-voting. Immaginiamo di voler realizzare una piattaforma online per la gestione di una votazione per un elezione politica dalle ore 7.00, con chiusura e inizio dello spoglio alle ore 23.00. Quello che vogliamo fare è dare la possibilità a ogni cittadino di votare elettronicamente e rendere il suo voto cifrato e inaccessibile fino al momento dello spoglio. E' necessario quindi trovare un sistema di time-lock encryption robusto, che garantisca la segretezza del voto per tutte le 16 ore di votazione.

1.4.2 Sistema di aste

Un altro sistema implementabile potrebbe essere quello delle aste online. Supponiamo di voler partecipare ad un'asta di beni della durata di 24 ore. Quello che vogliamo è dare la possibilità ad ogni cittadino di inviare un'offerta economica e di renderla disponibile solo quando lo ritiene necessario (entro i limiti richiesti).

1.4.3 Sistema militare

Un ulteriore sistema riguarda l'ambiente militare. Supponiamo che un centro di comando voglia organizzare un attacco in un territorio nemico coordinando diverse basi lontane tra loro. Il comando invierebbe le coordinate dell'attacco cifrate cosicchè possano essere decifrate solo al momento deciso. Il sistema di time-lock encryption deve essere così robusto da garantire che gli ordini non vengano scoperti prima del dovuto.

1.5 Risultati attesi

Il risultato che ci attendiamo è quello di trovare un sistema che garantisca la time-lock encryption e di poter fornire un'implementazione efficace e funzionante. Come vedremo successivamente in questo lavoro, uno dei problemi principali di tutti i sistemi basati sui puzzle, ritenuti i più promettenti, è quello di trovare una sorta di "incentivo" per spingere i partecipanti a comportarsi correttamente e a collaborare. Il modello che verrà implementato perseguirà questo obiettivo, introducendo degli elementi unici e personali.

1.6 Struttura della tesi

Il seguente lavoro di tesi è organizzato come segue:

- Nel Capitolo 2 viene affrontata una prima modalità di time-lock encryption basata sull'utilizzo, prima di agent non fidati, poi di trusted agents. Verranno analizzate nel dettaglio tecniche come il Secret Sharing, Key Sharing (decentralizzate con partecipanti non fidati) e tecniche come la Time–Lapse Cryptography e il Conditional Oblivious Transfer (centralizzate con partecipanti fidati ma curiosi).
- Nel Capitolo 3 viene affrontata una seconda modalità di time-lock encryption basata sulla risoluzione di puzzle complessi che richiedono un determinato quantativo di tempo prestabilito. Verranno analizzati nel dettaglio problemi computazioni basati su quadrati ripetuti e sul cifrario RC5 con chiave modulabile.
- Nel Capitolo 4 viene analizzato lo schema di time-lock encryption in ambiente Bitcoin basato sul Witness Encryption. Dopo la spiegazione del suo funzionamento dall'articolo originale ([8]), verrà approfondita l'implementazione proposta nell'articolo successivo ([11]) e analizzati i problemi che porta con sé.
- Nel Capitolo 5 verrà analizzato lo schema di time-lock encryption basato su curve ellittiche e Blockchain Ethereum. Verrà fornita un'implementazione di uno Smart Contract scritto nel linguaggio Solidity, un'approfondimento della

funzione di Icart per la mappatura di un messaggio in un punto di una curva e un'implementazione del Pollard Rho Attack scritto in Java per la valutazione delle vulnerabilità di una curva.

- Nel Capitolo 6 verrà svolta un'analisi sull'implementazione sviluppata riguardante la sua reale efficacia. Sarà inoltre mio interesse proporre delle modifiche all'implementazione per migliorare i problemi riscontrati dall'analisi stessa.
- Nel Capitolo 7 verrà fatto un riepilogo di tutto il lavoro effettuato e verrà mostrato un riassunto conclusivo di tutte le tecniche analizzate nel presente lavoro.

Capitolo 2

Time-lock encryption basato su agent

2.1 Introduzione

L'origine del concetto di time-lock encryption risale agli anni novanta da un famoso articolo scientifico ([1]) scritto da Rivest, Shamir e Wagner. L'obiettivo degli autori è stato sviluppare un metodo per cifrare un certo messaggio e renderlo decifrabile soltanto dopo il raggiungimento di un certo istante di tempo nel futuro. In questo caso, né il mittente né il destinatario dovrebbe essere in grado di accedere al contenuto dell'informazione prima del trascorrere della deadline desiderata. In questo capitolo verranno sottoposte ad una analisi approfondita quattro tecniche principali basate sulla presenza di agenti, fidati e non, in due contesti differenti: uno centralizzato e l'altro decentralizzato. Il primo passo sarà descrivere e studiare un primo modello decentralizzato, basato sulla condivisione di un'informazione segreta, chiamato Secret Sharing ([1]). Il secondo passo sarà vedere un'ulteriore evoluzione della tecnica precedente, denominato Key Sharing ([1]). In questo perfezionamento viene introdotta la gestione del tempo, prima non prevista, da parte di tutti i partecipanti. Il terzo passo sarà affrontare una nuova soluzione, nata nel 2006, chiamata Time-Lapse Cryptography ([3]). In questo caso si parla di un sistema costruito sullo sharing di un segreto e sul cifrario asimmetrico ElGamal. Infine, verrà valutata una tecnica centralizzata, chiamata Conditional Oblivious Transfer ([4]) che fa uso dei residui quadratici e basato sul concetto originale dell'Oblivious Transfer ([6]).

2.2 Secret Sharing

Il Secret Sharing è una tecnica proposta per la prima volta in [2] da Shamir nel 1979. Pur non essendo propriamente un modello di time-lock encryption, può essere considerato una base fondante di tutte le tecniche successive costruito sulla condivisione di un secreto. Questo approccio prevede l'utilizzo in un certo numero n di partecipanti che condividono dei pezzi di informazione. Essendo tutti i membri non fidati, è prevista la possibilità di utilizzare almeno k agenti "fidati" per ricostruire con facilità il dato. Altimenti, con meno di k partecipanti il segreto non potrebbere essere riassemblato con certezza.

Questa tecnica si basa sulla suddivisione di un certo dato D in n pezzi $D_1, \ldots, D_n$ tale che:

- Conoscere k o più pezzi D_i rende più semplice computare D.
- Conoscere $k-1$ o meno pezzi D_i rende praticamente impossibile computare il corretto D.

Lo schema appena citato si chiama schema (k, n) e viene descritto dal seguente polinomio di grado $k-1$:

$$q(x) = a_0 + a_1 * x + \ldots + a_{k-1} * x^{k-1}$$

Con

$$a_0 = D \quad e \quad D_1 = q(1), \quad D_2 = q(i) \quad e \quad D_n = q(n).$$

La distribuzione e generazione degli share da fornire ai diversi agenti può essere sintetizzata come segue:

- Il mittente Alice definisce il messaggio D da ricostruire (per semplicità supponiamo sia un numero intero).
- Alice sceglie un numero primo p, tale che D appartenga al campo finito Z_p ($\{1, \dots, p-1\}$).
- Alice genera casualmente $a_1, a_2, \dots, a_{k-1}$ valori appartenenti a $[0, p)$.
- Alice computa i k share e li distribuisce ad ogni agente nel seguente modo:
  - $D_1' = q(1) = D + a_1 * 1 + \dots + a_{k-1} * 1^{k-1}$.
  - $D_k' = q(k) = D + a_1 * k + \dots + a_{k-1} * k^{k-1}$.

Le k equazioni vengono raccolte all'interno di un sistema che deve essere risolto per ricostruire il dato D. Di conseguenza, ogni partecipante deve condividere il proprio share con tutti gli altri per garantire il corretto sviluppo del protocollo.

2.2.1 Esempio

Alice decide di prendere un messaggio $D = 3$ con primo $p = 7$. Sia il numero di partecipanti necessario $k = 2$ e viene generato casualmente $a_1 = 4$.
Alice costruisce i 2 shares nel seguente modo:

- $D_1' = q(1) = 3 + 4 * 1 + 4 * 1^1 = 11$
- $D_2' = q(2) = 3 + 4 * 2 + 4 * 2^1 = 19$

Si risolve il sistema di equazioni seguente:

$$\begin{cases} 11 = D + a_1 * 1 + a_1 * 1 = D + 2 * a_1 \\ 19 = D + a_1 * 2 + a_1 * 2 = D + 4 * a_1 \end{cases}$$

Il risultato è $D = 3$ e $a_1 = 4$.

Il problema principale della formulazione del Secret Sharing è che non viene prevista la gestione del tempo da parte dei partecipanti. E' possibile ovviare a questo fornendo delle istruzioni sulla schedule a tutti i partecipanti, tuttavia introduce un certo grado di centralizzazione nel protocollo. Questo approccio alternativo è molto simile al Key Sharing e al Time-Lapse Criptography spiegati successivamente.

2.3 Key Sharing

Nello stesso articolo di Rivest et al. ([1]), viene proposta un'alternativa al Secret Sharing con l'aggiunta della gestione del tempo da parte dei partecipanti.
Differentemente dalla tecnica precedente, abbiamo una ricostruzione, non più del messaggio, ma della chiave usata per cifrare l'informazione segreta.

Il messaggio M, rilasciato al tempo t, viene cifrato con un algoritmo convenzionale usando una chiave K scelta, nel seguente modo: $C = E(K, M)$. Alice individua una

chiave $K = y_1, y_2, \dots, y_d$, sceglie d agenti $i_1, i_2, \dots, i_d$ e pubblica $(C, a_1, a_2, \dots, a_d, r_1, r_2, \dots, r_d)$ dove $r_1, r_2, \dots, r_d$ sono i tempi in cui verrano rilasciati i segreti dagli agenti per ricostruire la chiave K.
Ad ogni agente a_i vengono forniti i valori y_i e t, che lui dovrà usare per pubblicare $E(s_{it}, y_i)$, ovvero il cifrato di y_i con chiave un segreto generato da lui s_{it}, il quale verrà rivelato al tempo futuro t. Ad ogni richiesta di qualsiasi altro partecipante, l'agente ritornerà questi valori digitalmente firmati con la sua chiave privata (così da poter verificare l'identità del mittente) e cifrati con la chiave pubblica del richiedente: $(i, t, t0, E(s_{it}, y))$ dove i è l'indice dell'agente, t è il tempo in cui pubblicherà il segreto s_{it}, t_0 è il tempo corrente e $E(s_{it}, y)$ è il cifrato richiesto. Chiaramente $t_0 < t_1$, anche se non richiesto espressamente.

Sintentizzando il protocollo, ogni agente è protagonista dei seguenti passi:

- L'agente genera un segreto s_{it}.
- L'agente riceve da Alice $(y_i, t, (e, n))$, cifrato con la sua chiave pubblica ((e, n) è la chiave pubblica di Alice) e firmato con la sua chiave privata (di Alice).
- L'agente cifra il valore y_i con il suo segreto s_{it}.
- Pubblica il testo cifrato creato e lo firma con la sua chiave privata.
- Al tempo t pubblica il segreto s_{it}.

Questo protocollo è costruito sul Secret Sharing, aggiungendo caratteristiche di time-lock encryption. Tuttavia, la sua debolezza risiede nel fatto che non c'è alcun motivo per cui i partecipanti debbano comportarsi correttamente perché non ottengono in cambio alcun incentivo. Per questo motivo, come vedremo in seguito, i sistemi più adatti sono quelli che lavorano in ambiente Blockchain.

2.4 Time-Lapse Cryptography

Nel 2006, Rabin e Thorpe scrissero un articolo ([3]) dove svilupparono un nuovo sistema che aveva l'obiettivo di garantire la time-lock encryption, chiamato Time-Lapse Cryptography. Questo nuovo modello si inserisce in un contesto decentralizzato, dove un certo numero di partecipanti, al tempo t, cooperano per consentire l'ottenimento del messaggio segreto. Entrando più nel dettaglio, questo approccio è fondato sulla distribuzione di chiavi e sul sistema di cifratura a chiave asimmetrica ElGamal ([5]). E' importante, quindi, prima di proseguire con la descrizione del protocollo, definire sinteticamente il funzionamente del cifraro ElGamal.

2.4.1 Cifrario a chiave pubblica ElGamal

Il cifrario ElGamal, è un sistema di cifratura di tipo asimmetrico, ovvero si basa sull'utilizzo di una coppia di chiavi (pubblica e privata) per cifrare e per creare firme digitali. Il suo schema si basa sulla difficoltà del calcolo del logaritmo discreto, ovvero trovare x, tale che $y = g^x$ con y e g forniti. Quello che a noi principalmente interessa in questo capitolo, è il meccanismo con il quale i messaggi vengono cifrati.
Sia p un numero primo molto grande e sia Z_p il campo finito costruito su p.
2.4.1.1 Definizione. ${Z_p}^*$ è l'insieme $\{a \in Z_p : mcd(a, p) = 1\}$.

2.4.1.2 Definizione. Un generatore g di $Z_p{}^*$ è un certo numero intero tale che l' insieme $\{g^i \mid 1 \leq i \leq p-1\} = Z_p{}^*$.

La chiave pubblica del sistema è composta dalla tripla (p, g, β), dove:

- p è un numero primo grande.
- g è un generatore di $Z_p{}^*$.
- $\beta = g^\alpha \ mod \ p$.

La chiave privata, invece, è $\alpha \in Z_{p-1}$ generato casualmente.
La cifratura di un messaggio $0 \leq M \leq p$ è così composta:

- $y_1 \leftarrow g^k \ mod \ p$ con $k \in Z_{p-1}$.
- $y_2 \leftarrow M * \beta^k \ mod \ p$.
- $C \leftarrow (y_1, y_2)$.

La decifratura di un testo cifrato C è così composta:

- $z \leftarrow {y_1}^\alpha \ mod \ p$.
- $M \leftarrow z^{-1} * y_2 \ mod \ p$, dove z^{-1} è l'inverso moltiplicativo modulo p.

2.4.1.3 Esempio

Alice vuole cifrare il messaggio $M = 3$ e sceglie $p = 11$. Di conseguenza, il generatore di ${Z_{11}}^*$ $g = 2$. Infine genera casualmente $\alpha = 5$ e $k = 7$.
La cifratura, con $\beta = 2^5 \ mod \ 11 = 10$, è cosi effettuata:

- $y_1 = 2^7 \ mod \ 11 = 7$.
- $y_2 = 3 * 10^7 \ mod \ 11 = 8$.
- $C = (7, 8)$.

La decifratura di C è cosi effettuata:

- $z = 7^5 \ mod \ 11 = 10$.
- $M = 10 * 8 \ mod \ 11 = 3$.

2.4.2 Funzionamento protocollo

Il protocollo Time-Lapse Cryptography si basa sulla presenza di una rete di entità (chiamate *party*), denominata "*the Service*" (d'ora in poi chiamato "il Servizio"), unite nel performare un determinato compito.
Il protocollo è così definito: al dato tempo T, il Servizio rende disponibile una chiave pubblica che chiunque può utilizzare, anche in modo anonimo. I mittenti (prendiamo Alice e Bob) cifrano il loro messaggio con la chiave pubblica (la cui chiave privata rimane sconosciuta), finchè, al tempo specificato $T + \delta$, la chiave di decifratura viene costruita e resa pubblica dal Servizio. In altri termini, Alice desidera spedire a Bob un messaggio m al tempo T, tale che m possa essere decifrato solo dopo che il tempo specificato $T + \delta$ è trascorso.
Dall'altro lato, il Servizio rende disponibile, al tempo T, la chiave pubblica PK, mentre la chiave privata DK verrà rivelata al tempo $T + \delta$. Alice usa la chiave pubblica PK per

cifrare m con un valore casuale k e invia $c = E_{PK}(m,k)$ a Bob. Al tempo $T + \delta$ il Servizio ricostruisce e pubblica DK, che Bob otterrà e userà per decifrare c e recuperare m.

Entrando nel dettaglio, questo protocollo è condotto dal Servizio, consistente di n partecipanti $P_1, \dots, P_n$. L'operazione di generazione di una chiave pubblica e ricostruzione di una chiave privata verrà chiamata *action* del Servizio. Assumiamo una soglia t, tale che durante una qualsiasi *action*, al massimo $t-1$ partecipanti possono ostacolare il protocollo rivelando informazioni segrete, inserendo informazioni false o rifiutandosi di partecipare. I partecipanti corretti si chiameranno *proper* (almeno t), gli altri *improper (massimo* $t-1$*).* Questo implica necessariamente che $n \geq 2t-1$. Riprendendo la definizione sopra dell'algoritmo di cifratura ElGamal, prendiamo un generatore $g \in Z_q^*$ e assumiamo un numero primo $p = 2q+1$. Il Servizio sarà implementato su una rete di computer autonomi, ognuno dei quali rappresenta un partecipante P_i che ottiene la schedule della generazione delle chiavi pubbliche e private da un set di computer manager. Questi k manager agiscono come managing team gestendo la schedule delle chiavi pubbliche e private e mantenendo una bacheca interna per i partecipanti e una esterna per gli utenti. L'integrità di queste bacheche è assicurata dal fatto che ogni manager ne mantiene una copia. Gli utenti e le parti P_i leggono le informazioni da tutte le bacheche di tutti i manager e scelgono in base alla maggioranza. Il tempo per tutte le *action* è regolato da un orologio universale accessibile (partecipanti e manager devono usarlo).
Assumiamo che ogni partecipante P_i può comunicare privatamente e segretamente con ogni altro partecipante P_j e conosce la sua chiave pubblica. Ogni partecipante P_i carica sulla bacheca interna un messaggio m e invia la sua firma digitale $SIGN_i(m)$. Il Servizio crea, pubblica e mantiene la "time-lapse cryptographic key structure", ovvero una key structure che contiene una chiave pubblica con uno specifico lifetime T e δ. Per ogni chiave richiesta da un utente (Alice o Bob per esempio), il Servizio genera una key structure $K_{ID} = (ID, T_{ID}, \delta_{ID}, PK_{ID})$ consistente di un identificatore univoco ID, un tempo di pubblicazione T_{ID}, un "*time-lapse*" (durata) δ_{ID} e una chiave pubblica PK_{ID}. Quando richiesta da Alice, ogni partecipante P_i pubblica la key structure e la firma sulla bacheca pubblica $(K_{ID}, SIGN_i(K_{ID}))$. Al tempo $(T_{ID} + \delta_{ID})$ il Servizio ricostruirà e pubblicherà la chiave privata associata DK_{ID}. La chiave pubblica e la chiave privata per la key structure K_{ID} rispettano la seguente equazione $PK_{ID} \equiv g^{DK_{ID}} \ (mod\ p)$. Come prima, ogni partecipante P_i pubblica la chiave privata ricostruita e la firma $(K_{ID}, SIGN_i(K_{ID}))$ sulla bacheca pubblica. La ricostruzione della chiave privata non è istantanea. Questo perchè il Servizio inizierà la ricostruzione della chiave privata DK_{ID} al tempo $(T_{ID} + \delta_{ID})$ e pubblicherà DK_{ID} al tempo $(T_{ID} + \delta_{ID} + \epsilon)$, dove ϵ è il tempo richiesto per ricostruire la chiave privata.
Quando Alice desidera inviare un messaggio m a Bob, lei richiede, o seleziona, un'appropriata key structure K_{ID} dal Servizio. Inoltre, verifica che la firma digitale pubblicata $SIGN_i(K_{ID})$ corrisponda alla key structure K_{ID} di almeno t partecipanti P_i, e che questi K_{ID} siano uguali. Questo garantisce che la chiave pubblica PK_{ID} è generata da tutti i partecipanti *proper*, e la sua corrispondente chiave privata DK_{ID} sarà ricostruita e postata dagli stessi. Per inviare un messaggio, Alice cifra m con il sistema di cifratura ElGamal; sceglie un valore casuale $k \in [1, q-1]$ e privatamente invia a Bob la coppia

$C = (g^k \ (mod\ p), m * {PK_{ID}}^k \ (mod\ p))$. Bob riceve C e lo salva, quindi aspetta il tempo $(T_{ID} + \delta_{ID})$.
A questo punto è fondamentale descrivere il funzionamento e il metodo di ricostruzione della chiave privata associata.
Innanzitutto, definiamo Q come l'insieme di tutti partecipanti qualificati *proper*, con $|Q| \geq t$. Ogni partecipante P_i sceglie un valore casuale $x_i \in [1, q-1]$. Questo x_i costituisce il candidato componente, del partecipante P_i, della chiave privata. La chiave privata DK sarà $x = \sum_{i \in Q} x_i \ (mod\ q)$. Ogni P_i dovrebbe computare $h_i = g^{x_i} \ (mod\ p)$ e pubblicare $\left(h_i, SIGN_i(h_i)\right)$ sulla bacheca interna. La chiave pubblica PK sarà $h = \prod_{i \in Q} h_i \ (mod\ p)$ e verrà pubblicata da ogni partecipante sulla bacheca pubblica come descritto precedentemente. Questo h_i è il candidato componente di P_i della chiave pubblica. Qualsiasi partecipante P_i che non pubblica h_i viene eliminato dall'insieme Q e diventa *improper*.
Ogni partecipante P_i dovrebbe creare un polinomio di grado $k = t - 1$ generando $a_{1i}, \dots, a_{ki} \in [1, q-1]$ casuali.
Il componente chiave segreto è $f_i(0) = x_i$. Ogni partecipante P_i dovrebbe computare i secret shares $x_{ij} = f(j)$ (tutti gli share che compongono x_i) e i valori di verifica $c_0 = h_i = g^{x_i}, c_1 = g^{a_{1i}}, \dots, c_k = g^{a_{ki}}$. (Tutti c_i sono computati $(mod\ p)$). Ogni partecipante P_i privatamente invia a tutti P_j, con $j \in [1, n]$, $\left(j, x_{ij}, SIGN_i\left(j, x_{ij}\right)\right)$ e pubblica sulla bacheca interna i verificatori $\left(c_0, SIGN_i(c_0)\right), \dots, \left(c_k, SIGN_i(c_k)\right)$.
A questo punto, ogni P_j può ora verificare che ogni x_{ij} ricevuto è un corretto share, controllando:

$$(*) \qquad g^{x_{ij}} \equiv c_0 * c_1^j * c_2^{j^2} * \dots * c_k^{j^k} \ (mod\ p)$$

A questo punto un *improper* P_i può ostacolare il processo in due modi:

- Può inviare uno share scorretto x_{ij} del suo componente x_i. In questo caso, P_i pubblica la tripla $(j, x_{ij}, SIGN_i\left(i, x_{ij}\right))$ sulla bacheca interna. I partecipanti *proper* controlleranno se x_{ij} è un valido share secondo la formula (*). Se non è un valido share, allora P_i viene eliminato dall'insieme Q. Tutti i partecipanti possono verificare se x_{ij} è un valido share secondo (*).
- Oppure P_i potrebbe fallire nell'inviare a P_j lo share x_{ij}. In questo caso P_j invia nella bacheca una protesta firmata. P_i è obbligato a rivelare x_{ij} sulla bacheca interna postando un messaggio firmato $\left(j, x_{ij}, SIGN_i\left(j, x_{ij}\right)\right)$. Ogni partecipante può verificare lo share x_{ij} pubblicato secondo (*). Se invalido, allora P_i viene eliminato dall'insieme Q.

Ora, ogni partecipante P_j qualificato mantiene la chiave pubblica h, un componente x_j della chiave privata x, e gli share x_{ij} di tutti i partecipanti qualificati P_i. Questi ultimi share sono mantenuti per la ricostruzione dei componenti x_i.
Ogni partecipante qualificato P_j crea $h = \prod_{i \in Q} g^{x_i} \ (mod\ p)$ (ottenuto dai h_i precedentemente inviati) e la key structure $K_{ID} = (ID, PK_{ID} = h, T_{ID}, \delta_{ID})$ e pubblica $(K_{ID}, SIGN_j(K_{ID}))$ sulla bacheca interna e sulla bacheca esterna. Tutti i

partecipanti *proper* durante questa *action* pubblicheranno lo stesso K_{ID}. Al tempo $(T_{ID} + \delta_{ID})$, per la ricostruzione della chiave privata DK_{ID}, tutti i partecipanti *proper* saranno coinvolti.
Primo, ogni partecipante P_i dovrebbe pubblicare la sua componente x_i della chiave privata $x = DK_{ID}$ nella bacheca interna. Alcune componenti potrebbero mancare, per il fatto che qualche partecipante qualificato potrebbe diventare *improper*. Ogni partecipante $proper$ controlla che per ogni partecipante qualificato, il x_i postato soddisfi l'equazione $g^{x_i} \equiv h_i \ (mod\ p)$, dove h_i è pubblicato nello step precedente. Per ogni $P_i \in Q$ che potrebbe non postare x_i, i partecipanti hanno bisogno di ricostruirlo.
Ogni partecipante P_j dovrebbe pubblicare sulla bacheca interna x_{ij} ricevuto da P_i precedentemente.
In sintesi, ogni *proper* P_j vede sulla bacheca interna almeno t validi share x_{ij} di componenti di x_i di P_i della chiave privata $x = DK_{ID}$. Il partecipante P_j usa i t share x_{ij} per ricostruire x_i dalla interpolazione polinomiale.
Dopo aver fatto questo, ogni *proper* P_j ha tutti i componenti x_i di tutti i partecipanti $P_i \in Q$. Ogni partecipante P_j computa la somma $DK_{ID} = x = \sum_i x_i \ (mod\ p)$ e pubblica $\left(ID, DK_{ID}, SIGN_j(ID, DK_{ID})\right)$ sulla bacheca pubblica.

Come il protocollo Key Sharing descritto precedentemente, resta il problema che ogni partecipante e, in questo caso, manager, non è incentivato in alcun modo (per esempio attraverso un reward) a comportarsi correttamente.

2.5 Conditional Oblivius Transfer

Il Conditional Oblivious Transfer è un protocollo costruito sull'originale Oblivious Transfer, definito da Rabin nel 1981 in [6]. Il principio su cui si basa riguarda l'invio di uno di molti pezzi di informazione (per semplicità supponiamo la trasmissione di un bit 0/1) senza che il mittente possa sapere concertezza quale pezzo è stato trasferito dal destinatario. Prima di definire il protocollo principale ([4]), vedremo il funzionamento dell'Oblivious Trasfer.

2.5.1 Oblivious Transfer

Rabin, nel suo articolo [6], definisce un nuovo protocollo, chiamato Oblivious Transfer, con l'obiettivo di descrivere una modalità di scambio di segreti tra due parti (Alice e Bob). Bob e Alice possiedono un segreto, SB e SA, rispettivamente, che desiderano scambiarsi. Supponiamo per esempio che SB rappresenti la password di un file che Alice vuole leggere e SA la password di un file che Bob vuole leggere. L'obiettivo di questo protocollo è effettuare uno scambio di segreti, garantendo il corretto funzionamento senza una parte terza di fiducia. Per rendere tutto più "realistico", supponiamo che, nel caso le password di entrambi i file fossero sbagliate, questi ultimi vengano immediatamente cancellati, e, per semplicità, supponiamo che sia SA sia SB siano singoli bit (0 o 1). Inoltre, premettiamo che ogni file contiene un prefisso (cifrato) rappresentante la password necessaria per aprirlo. Tuttavia, questo prefisso è accessibile solo nel caso in cui il file devesse essere cancellato a causa dell'inserimento della password sbagliata.

Alice e Bob, durante ogni passaggio del protocollo, si scambiano messaggi firmati con le rispettive chiavi private. Per esempio, Bob invia ad Alice : "Il mio segreto è S, firmato Bob". Alice utilizza il segreto S e, se $S \neq SB$, allora il suo file viene distrutto (tranne il prefisso che contiene SB). In generale, un protocollo di Oblivious Transfer può essere descritto in questo modo: Alice fornisce a Bob un'informazioni I_1, Bob fornisce ad Alice J_1, e cosi via. Tuttavia, deve esistere un primo k tale che Bob può determinare SA da $I_1, \dots, I_k$, mentre Alice non può determinare SB da $J_1, \dots, J_{k-1}$. In altre parole, bisogna evitare che Bob mantenga segreto J_k e riesca ad ottenere SA senza rivelare SB. L'unico modo per risolvere il problema è costruire un protocollo tale che, se Bob conosce SA, allora Alice può dedurre SB.

Assumiamo che Alice e Bob posseggano entrambi una chiave pubblica e una privata K_A e K_B che possono usare per cifrare e firmare digitalmente. Ogni messaggio da Alice per Bob sarà firmato usando K_A, e viceversa. Alice sceglie due numeri primi molto grandi p, q e computa una chiave one-time $n_A = p * q$. Invia a Bob un messaggio: "La chiave one-time è n_A, firmato Alice". Bob sceglie due primi p_1, q_1 e invia $n_B = p_1 * q_1$ ad Alice in un messaggio firmato. Bob adesso sceglie casualmente un $x \leq n_A$, computa $c = x^2 \ mod \ n_A$, e invia ad alice il messaggio "$E(x)$ è la cifratura con la mia chiave pubblica del mio numero scelto, e c è il quadrato $mod \ n_A$ di quel numero, firmato Bob".
Alice, che conosce i fattori p, q di n_A calcola un x_1 tale che ${x_1}^2 = c \ mod \ n_A$ e trasmette a Bob il messaggio "x_1 è la radice quadrata $mod \ n_A$ di c, firmato Alice". Bob calcola $\text{mcd}(x - x_1, n_A) = d$ e, con probabilità $\frac{1}{2}$, abbiamo che $[d = p \ oppure \ d = q]$, ovvero con probabilità $\frac{1}{2}$ Bob conosce la fattorizzazione $n_A = p * q$. Comunque, siccome Alice non conosce la x di Bob, lei non sa se Bob possiede effettivamente la fattorizzazione di n_A. Noi ci riferiamo a questo trasferimento di informazioni, dove il mittente non sa se il destinatario ha realmente ricevuto l'informazione, come un oblivious transfer.
Successivamente, Bob effettua un oblivious transfer di n_B ad Alice allo stesso modo. Definiamo $v_B = 0 \ se \ \text{mcd}(x - x_1, n_A) = p \ oppure \ q$.
$v_B = 0$ se e solo se, dopo l'oblivious transfer della fattorizzazione di n_A da Alice a Bob, Bob conosce i fattori. Il bit v_A di Alice è definito come descritto precedentemente.
Ricordiamo che SA e SB sono singoli bit. Bob calcola l'or esclusivo (xor) $\varepsilon_B = S_B \ xor \ v_B$ e invia ad Alice il messaggio "ε_B è l'or-esclusivo del mio segreto con il mio stato di conoscenza dei fattori di n_A, firmato Bob". La conoscenza di ε_B non contribuisce in nessun modo all'abilità di Alice di accedere al suo file.
Similarmente Alice calcola $\varepsilon_A = S_A \ xor \ v_A$ e lo manda a Bob in un messaggio firmato. Alla fine del protocollo, Alice mette il suo segreto S_A come bit centrale in un messaggio casuale m_A. Lei cifra il messaggio come $E_{n_A}(m_A) = C$ usando qualsiasi sistema a chiave pubblica che richiede i fattori p, q di n_A per la decodifica (RSA per esempio). Alice invia C a Bob e lo firma. Bob fa la stessa cosa per Alice. A questo punto, conoscere la fattorizzazione di $n_A(n_B)$ permette di ottenere il messaggio $m_A(m_B)$ e, di conseguenza, ottenere il segreto $SA(SB)$.
Il protocollo appena descritto, ha una probabilità di $\frac{1}{4}$ che nessuno delle due parti ottenga il segreto dell'altro. Partiamo comunque dal presupposto che entrambi portino a conclusione il protocollo, pena perdita del segreto. Se Bob conosce effettivamente la

fattorizzazione n_A ($v_B = 0$), può decodificare d_A, trovare m_A e di conseguenza SA. Allo stesso modo anche Alice può conoscere SB e utilizzarlo per leggere il suo file.

2.5.2 Funzionamento protocollo

Come già accennato, il Conditional Oblivious Tranfer è un evoluzione dell'originale Oblivious Transfer ([6]). Si tratta di un sistema centralizzato che garantisce la time-lock encryption attraverso la presenza di un Time Server. Questo server definisce la natura del tempo e restituisce (periodicamente oppure no) il tempo corrente. Una proprietà fondamentale è la possibilità di garantire l'anonimato del mittente, infatti non c'è alcuna interazione tra il server e il mittente.
Il server possiede solo una chiave pubblica e non conosce le chiavi degli altri partecipanti. Prima di procedere, è necessario fare alcune assunzioni sul sistema in esame:

- Il mittente non è sempre presente.
- Il mittente non si fida del destinatario per fornigli la chiave di decifratura del messaggio in anticipo.
- Il mittente e il server non comunicano in alcun modo, ma il destinatario può interagire al minimo col server.

Fatte queste considerazioni, il tutto può essere sintetizzato come segue: "Può il mittente, senza comunicare col server, creare un documento cifrato con un tempo di rilascio, tale che un destinatario può decifrare questo documento solo dopo che il tempo di rilascio è trascorso interagendo con il server, e il server non impara l'identità del mittente?". Bob e Alice (mittente/destinatario) hanno degli input privati e condividono un predicato pubblico valutato sugli input ed è computabile in tempo polinomiale. Il Conditional Oblivius Transfer di un bit b (per semplicità) da Alice a Bob ha i seguenti requisiti: se il predicato è vero, allora Bob riceve con successo il bit che Alice voleva inviargli, ma se il predicato è falso, allora lui non riceverà alcun'informazione sul bit che Alice voleva inviargli.
I partecipanti di questo protocollo sono tre: il mittente, il destinatario e il server. Il mittente trasmette al destinatario dei messaggi cifrati e il release-time. Il ricevente può comunicare col server e iniziare con lui un Conditional Oblivious Transfer tale che, se il release-time non è minore del tempo corrente (definito dal server), allora il destinatario riceve il messaggio, altrimenti non riceve nulla. Inoltre, il server non impara nulla sul release-time o sull'identità del mittente. In particolare, il server non sa se il release-time è minore, uguale o più grande del tempo corrente e un'esecuzione dello schema consiste in un singolo messaggio dal mittente al destinatario e un richiesta-risposta tra destinatario e time server.

Prima di entrare nel dettaglio dell'algoritmo, ci sono alcune definizioni necessarie per comprendere il sistema nella sua interezza.
2.5.2.1 Definizione. Con $x \leftarrow^D S$ denotiamo l'esperimento probabilistico in cui viene scelto un elemento x dall'insieme S secondo una distribuzione D. Possiamo semplificare in $x \leftarrow S$ nel caso D sia una distribuzione uniforme su S. La notazione $y \leftarrow A(x)$, dove A è un algoritmo, rappresenta l'esperimento probabilistico di ottenere y quando un algoritmo A è in esecuzione su un input x. Similarmente, la nozione $t \leftarrow (A(x), B(y))(z)$ denota l'esperimento probabilistico rappresentante l'esecuzione di un protocollo

interattivo (A, B), dove x è un input di A, y è un input di B, z è un input comune ad A e B, e t è la trascrizione della comunicazione tra A e B durante tale esecuzione.

2.5.2.2 Definizione. Con $Prob[R_1; \dots; R_n : E]$ denotiamo la probabilità dell'evento E, dopo l'esecuzione degli esperimenti probabilistici $R_1, \dots, R_n$.

2.5.2.3 Definizione. Date due sequenze di k bit $t_1, \dots, t_k$ e $d_1, \dots, d_k$, viene definito il predicato GE come segue: $GE(t_1, \dots, t_k, d_1, \dots, d_k) = 1$ se e solo se $(t_1 * \dots * t_k) \geq (d_1 * \dots * d_k)$, quando le due stringhe sono interpretate come interi.

Siano Alice e Bob due macchine di turing probabilistiche che eseguono in tempo polinomiale con un parametro di sicurezza n. Sia anche $x_A(x_B)$ l'input di Alice (Bob), sia b il bit privato che Alice vuole inviare a Bob, e sia $q(\dots, \dots)$ un predicato computabile in tempo polinomiale. Diciamo che $(Alice, Bob)$ è un protocollo Conditional Oblivious Transfer, per il predicato q, se esiste una costante a tale che vale la seguente proprietà:

- Transfer Validity. Se $q(x_A, x_B) = 1$ allora per ogni $b \in \{0,1\}$, vale che
$$Prob\begin{bmatrix} \sigma \leftarrow \{0,1\}^{n^a}; \\ tr \leftarrow \big(Alice(x_A, b), Bob(x_B)\big)(\sigma) : \\ Bob(\sigma, x_B, tr) = b \end{bmatrix} = 1.$$
dove σ rappresenta una stringa pubblica casuale condivisa tra le parti.

La stringa σ, chiamata anche *reference string*, viene utilizzata per la generazione della chiave pubblica e per la verifica della stessa. Il suo utilizzo viene descritto in [7], ma per i nostri scopi non ci interessa approfondirlo.

Prima di proseguire, forniamo una prima informale descrizione del Timed-Release Encryption associato al Conditional Oblivious Transfer. Gli elementi fondamentali di questo protocollo sono i tre partecipanti: un mittente S, un destinatario R e un server V. Il tempo (rappresentato come un intero positivo) viene descritto da una stringa di $k - bit$ ed è interamente gestito dal server V. Ogni messaggio cifrato, spedito da S a R, sarà associato a un release-time $d = (d_1, \dots, d_k)$, dove $d_i \in \{0,1\}$, per ogni $i = 1, \dots, 2^k$. R sarà in grado di decifrare il messaggio solo dopo che il release-time d è trascorso. Ad R è permesso di interagire con V, mentre S non ha mai necessità di farlo. La conversazione che V vede non rivela informazioni circa il messaggio attuale, il suo release-time o la coppia mittente/destinatario interessata nella corrente conversazione. Il tempo è gestito da V rispondendo alle richieste di R: prima R invia un messaggio a V, poi V risponde con una serie di informazioni che permettono ad R di decifrare il messaggio ricevuto da S se e solo se il tempo corrente è più grande del release-time.
Descrivendo il sistema in modo più formale, denotiamo con (p_r, s_r) (rispettivamente (p_v, s_v)) la coppia di chiave pubblica/privata di R (V) e con σ una stringa casuale pubblica sufficientemente lunga. Con $m \in \{0,1\}^*$ un messaggio, t e d il tempo corrente secondo l'orologio di V e il release-time del messaggio m, siano entrambi rappresentati come una stringa di k bit. Sia S, R, V le tre macchine di turing probabilistiche in tempo polinomiale con un parametro di sicurezza n. Diciamo che (S, R, V) è uno Timed-Release Encryption scheme se esiste una costante a tale che vale la seguente proprietà:

- Correttezza. Per qualsiasi messaggio $m \in \{0,1\}^*$ e qualsiasi $d \in \{0,1\}^k$,

$$Prob\left[\begin{array}{c}\sigma \leftarrow \{0,1\}^{n^a};\\ (p_v, s_v) \leftarrow V(\sigma);\\ (p_r, s_r) \leftarrow R(\sigma);\\ (enc, d) \leftarrow S(p_r, p_v, m, d);\\ req \leftarrow R(p_r, s_r, p_v, enc, d);\\ ans \leftarrow V(p_v, s_v, t, req, \sigma):\\ (t < d)\ or\ (R(p_r, s_r, ans) = m)\end{array}\right] = 1$$

Il Conditional Oblivious Transfer si basa sull'uso dei residui quadratici per il trasferimento di informazioni. Di seguito le definizioni necessarie per il proseguo della descrizione.

2.5.2.4 Definizione. Un numero intero q è un residuo quadratico modulo p, se esiste un x tale che $x^2 \equiv q\ (mod\ p)$. In caso contrario, q è detto essere un non-residuo quadratico.

2.5.2.5 Definizione. Un numero naturale n è un intero di Blum se $n = p * q$, dove p e q sono numeri primi congruenti a 3 in modulo 4, cioè se 4 divide sia $p - 3$ che $q - 3$.

2.5.2.6 Definizione. Il gruppo ciclico Z_x^{+1}, significa che tutti gli elementi di Z_x hanno simbolo di Jacobi +1 se $\left(\frac{a}{n}\right) = \left(\frac{a}{p_1}\right)^{\alpha_1} * \left(\frac{a}{p_2}\right)^{\alpha_2} * \ldots * \left(\frac{a}{p_k}\right)^{\alpha_k} = 1$, con a elemento in Z_x e $n > 2$ è numero naturale dispari con $n = {p_1}^{\alpha_1} * {p_2}^{\alpha_2} * \ldots * {p_k}^{\alpha_k}$. Se p è un numero primo dispari e α è un quadrato $mod\ p$, ossia se esiste un intero k tale che $k^2 \equiv \alpha\ (mod\ p)$, oppure equivalentemente se α è un residuo quadratico modulo p, allora il simbolo di Jacobi è +1.

Quello mostrato di seguito è lo pseudo codice preso direttamente dall'articolo originale del Conditional Oblivious Transfer ([4]).

THE ALGORITHM ALICE: On input $b, t_1, \ldots, t_k \in \{0,1\}$, Alice does the following:

1. **Receive:** $x, D_1, \ldots, D_k$ from Bob and set $b_1 = b$.
2. For $i = 1, \ldots, k$,
   uniformly choose $a_i, c_i \in \{0,1\}, r_i \in Z_x^*$ and compute $T_i = r_i^2(-1)^{t_i} \bmod x$;
   if $i = k$ then set $c_i = b_i$;
   compute $mes_{i1} = \text{NQR-COT-Send}(a_i, (x, T_i))$;
   compute $mes_{i2} = \text{NQR-COT-Send}(a_i \oplus b_i, (x, D_i T_i \bmod x))$;
   compute $mes_{i3} = \text{NQR-COT-Send}(c_i, (x, -D_i T_i \bmod x))$;
   set $b_{i+1} = b_i \oplus c_i$;
   set $p_A = (T_1, \ldots, T_k)$ and $mes = ((mes_{11}, mes_{12}, mes_{13}), \ldots, (mes_{k1}, mes_{k2}, mes_{k3}))$;
   send: (p_A, mes) to Bob.

THE ALGORITHM BOB: On input a sufficiently long string σ and $d_1, \ldots, d_k \in \{0,1\}$, Bob does the following

1. Uniformly choose two n-bit primes p, q such that $p \equiv q \equiv 3 \bmod 4$ and set $x = pq$; for $i = 1, \ldots, k$, uniformly choose $r_i \in Z_x^*$ and compute $D_i = r_i^2(-1)^{d_i} \bmod x$; let $p_B = (x, D_1, \ldots, D_k)$ and **send:** p_B to Alice.
2. **Receive:** $((T_1, \ldots, T_k), (mes_{11}, mes_{12}, mes_{13}, \ldots, mes_{k1}, mes_{k2}, mes_{k3}))$ by Alice.
3. For $i = 1, \ldots, k$,
   compute $a_i = \text{NQR-COT-Receive}(mes_{i1}, (x, p, q, T_i))$;
   compute $e_i = \text{NQR-COT-Receive}(mes_{i2}, (x, p, q, D_i T_i \bmod x))$;
   if $a_i \neq -$ and $e_i \neq -$ then
   output: $a_i \oplus e_i \oplus c_{i-1} \oplus \cdots \oplus c_1$ and halt;
   else compute $c_i = \text{NQR-COT-Receive}(mes_{i3}, (x, p, q, -D_i T_i \bmod x))$;
   if $i = k$ and $c_i \neq -$ then **output:** c_i and halt;
   output: $-$.

THE ALGORITHM S:

Key-Generation Phase: no instruction required.

Encryption Phase:

1. Let (m, d) be the pair message/release-time input to S, where $m \in \{0,1\}$;
2. let (x) be R's public key;
3. uniformly choose $r \in Z_x^*$ and compute $c_m = r^2(-1)^m \bmod x$;
4. let $d = d_1, \ldots, d_k$, where $d_i \in \{0,1\}$, for $i = 1, \ldots, k$;
5. for $i = 1, \ldots, k$, uniformly choose $r_{2i} \in Z_x^*$ and compute $D_i = r_{2i}^2(-1)^{d_i} \bmod x$;
6. let $c_d = (D_1, \ldots, D_k)$;
7. compute $cc = \text{nm-E}(nm\text{-}pk, c_d \circ c_m)$ and **output:** (cc, c_d, d).

THE ALGORITHM R:

Key-Generation Phase:

1. Uniformly choose two $n/2$-bit primes p, q such that $p \equiv q \equiv 3 \bmod 4$ and set $x = pq$;
2. let L be the language $\{x \mid x \text{ is a Blum integer }\}$;
3. using σ, p, q, compute a non-interactive zero-knowledge proof Π for L;
4. **output:** (x, Π).

Decryption Phase:

1. Let (cc, c_d, d) be the triple received by S;
2. let $d = d_1, \ldots, d_k$, where $d_i \in \{0,1\}$, for $i = 1, \ldots, k$;
3. for $i = 1, \ldots, k$,
   using p, q, set $d_i' = 1$ if $(x, D_i) \in NQR$ or $d_i' = 0$ otherwise;
   if $d_i' \neq d_i$ then **output:** $-$ and halt.
4. run step 1 of algorithm Bob, by sending $(x, D_1, \ldots, D_k)$ to V;
5. send cc, Π to V;
6. run step 2 of algorithm Bob, by receiving $(T_1, \ldots, T_k), mes$ from V;
7. run step 3 of algorithm Bob, by decoding mes as c_m;
8. if $c_m \neq -$ then compute $m = \text{D}(sk, pk, c_m)$ and **output:** m else **output:** $-$.

THE ALGORITHM V:

Key-Generation Phase:

1. Run algorithm nm-G to generate a pair $(nm\text{-}pk, nm\text{-}sk)$;
2. output: $nm\text{-}pk$.

Timing service phase:

1. run step 1 of algorithm Alice, by receiving $(x, D_1, \ldots, D_k)$ from R;
2. receive cc, Π from R;
3. verify that the proof Π is accepting;
4. compute $(c'_d, c'_m) = \text{nm-D}(nm\text{-}sk, nm\text{-}pk, cc)$;
5. if $c_d \neq c'_d$ or the above verification is not satisfied then **output** $-$ to R and **halt**;
6. let $t = (t_1, \ldots, t_k)$ be the current time, where $t_i \in \{0, 1\}$, for $i = 1, \ldots, k$;
7. for $i = 1, \ldots, k$, uniformly choose $r_i \in Z_x^*$ and compute $T_i = r_i^2(-1)^{t_i} \bmod x$;
8. run step 2 of algorithm Alice, by computing mes;
9. **output:** $(T_1, \ldots, T_m), mes$.

Come visibile dallo pseudocodice sopra, il server V e il destinatario R hanno una fase iniziale di generazione delle chiavi. R genera un intero di Blum x e un *non-interactive zero knowledge proof II* usando la stringa σ, mentre V genera una coppia di chiavi pubblica e privata con un qualche algoritmo.

Per quanto riguarda S, ha una fase di *encryption* che prevede i seguenti passaggi:

- Prende il messaggio m e lo cifra attraverso i residui quadratici usando la chiave pubblica x di R.
- Prende il release time d e lo cifra allo stesso modo usando la chiave pubblica x di R.
- Prende i due cifrati e li cifra con la chiave pubblica del server V.
- Restituisce come output il cifrato finale.

Per quanto riguarda R, ha una fase di *decryption* che prevede i seguenti passaggi:

- Riceve il cifrato da S.
- Decifra il release time con i residui quadratici e verifica se è corretto.
- Genera un nuovo intero di Blum x come chiave pubblica, cifra con i residui quadratici il release time e invia il risultato (insieme a x) al server V.
- Invia a V il cifrato finale cc generato da S.
- Riceve il *current time* T e mes da V.
- Per ogni $mes_{i1}, mes_{i2}, mes_{i3}$ effettua una operazione di decifratura all'inverso.
- Infine, usa la sua chiave privata per decifrare c_m e ottenere m.

Per quanto riguarda V, ha una fase di *timing service* che prevede i seguenti passaggi:

- Riceve x e il cifrato del release time da R.
- Riceve il cifrato cc e verifica il *proof II*.
- Effettua una decifratura di cc con la sua chiave privata e verifica che il cifrato c_d (S) è uguale al $c_d{}'$ (R).
- Cifra il *current time* con i residui quadrati usando la chiave pubblica x di R e prende come b il dato da inviare attraverso l'Oblivious Transfer.

- Cifra una serie di messaggi richiamando il metodo NQR-COT-Send(m, x, y). Con questo metodo si cifra m nel seguente modo: $c_m = y^m * r^2 \ mod \ x$.
- Invia a R il *current time* e il cifrato finale mes.

E' facile verificare che c corrisponde a un bit 0 se si tratta di un residuo quadratico, oppure a 1 se è un residuo non-quadratico modulo x. Questa osservazione mostra che il sistema di decifratura è molto semplice: è sufficiente che Bob computi la residuità quadratica di c per trovare m. Infatti, questo sistema di cifratura non è un puzzle che richiede un certo quantitativo di tempo per essere risolto, ma è fatto per essere risolto immediatamente dopo che il release time è trascorso.
In questo sistema è Alice (V) che effettua un Oblivious Transfer di una serie di informazioni, che permettono a Bob, se correttamente ottenute, di decifrare il messaggio originale di S.
Per i nostri scopi non ci interessa affrontare la questione del non-interactive zero knowledge proof, ma per completezza verrà mostrato di seguito lo pseudocodice direttamente preso da [7].

Algorithm *Key_Generator*(σ)

Input: An n^3-bit reference string $\sigma = \sigma_1 \circ \ldots \circ \sigma_{n^2}$, where $|\sigma_i| = n$ for $i = 1, 2, \ldots, n^2$.

1. *Select public and secret keys.*
   Randomly select two n-bit primes $p, q \equiv 3 \bmod 4$ and set $x = pq$.
   Randomly select $r \in Z_x^*$, $z \in Z_x^{+1}$ and compute $y = -r^2 \bmod x$.
2. *Validate public key.*
   Set Val= empty string.
   For $i = 1, \ldots, n^2$
   if $\sigma_i \in Z_x^{+1}$ then
   if $Q_x(\sigma_i) = 0$ then append $\sqrt{\sigma_i} \bmod x$ to Val.
   if $Q_x(\sigma_i) = 1$ then append $\sqrt{y\sigma_i} \bmod x$ to Val.
   else append σ_i to Val.
3. Set $PK = (x, y, z, Val)$ and $SK = (p, q)$.

Output:(PK, SK).

Algorithm *Verify_PublicKey*(σ, PK)

Input: An n^3-bit reference string $\sigma = \sigma_1 \circ \ldots \circ \sigma_{n^2}$, where $|\sigma_i| = n$ for $i = 1, 2, \ldots, n^2$. A Public Key $PK = (x, y, z, Val)$, where $Val = v_1 \circ \ldots \circ v_{n^2}$.

For $i = 1, \ldots, n^2$
if $\sigma_i \in Z_x^{+1}$ then verify that either $\sigma_i = v_i^2 \bmod x$ or $y\sigma_i = v_i^2 \bmod x$.

Output: If all checks are successfully passed then output VALID else NONVALID.

Figura 1 - Rappresenta l'utilizza della reference string per generare la chiave pubblica e verificarla successivamente ([7]).

Capitolo 3

Time-lock encryption basato su puzzle

3.1 Introduzione

In questo capitolo verrà analizzata una modalità molto differente rispetto a quella basata sugli agenti. Nello stesso articolo ([1]), Shamir et al. hanno proposto un nuovo sistema con l'obiettivo di garantire la time-lock encryption ed eliminare l'importante grado di fiducia richiesto nelle tecniche precedente: Time-Lock Puzzles.
Questo approccio si fonda sulla realizzazione di un problema computazionale (chiamato puzzle), la cui soluzione permette di rivelare una chiave di decifratura, grazie alla quale è possibile accedere ad un segreto. La sfida maggiore nello sviluppo di problemi complessi risiede nel cercare di rendere il tempo di risoluzione necessario alla CPU il più vicino possibile al tempo reale desiderato. Insieme a questo fatto, bisogna prendere anche in considerazione la possibilità che il destinatario possa utilizzare più sistemi in parallelo per ridurre drasticamente i tempi di risoluzione del puzzle. In [1] viene fornito un semplice paragone per comprendere bene il concetto: "*risolvere un problema computazione è come avere un bambino. Due donne non possono partorirlo in 4.5 mesi*". Tutte le soluzioni mostrate d'ora in avanti, non si basano sul rendere risolvibile il puzzle in un certo istante (la deadline prestabilita per esempio), ma sull'esecuzione continua di un codice fino alla completa risoluzione.

3.2 RC5 con chiave di lunghezza modulabile

Un primo modello proposto da Rivest et al. in [1], è quello basato sull'uso del crittosistema RC5 con lunghezza della chiave modulabile sulla base di alcuni parametri. Sia M il messaggio segreto e sia S il numero di decifrature (con RC5) che il sistema avversario è in grado di eseguire ogni secondo. Cifriamo M utilizzando l'algoritmo RC5 con chiave di dimensione $|k| = log(2 * S * T)$ bit, dove T corrisponde al tempo in secondi dopo il il messaggio può essere decifrato. Dopo di che, la chiave viene buttata via e viene richiesto l'utilizzo di un attacco a forza bruta, sullo spazio della chiave, della durata di circa T secondi.

Questo approccio richiede alcune considerazioni da fare:

- Un attacco a forza bruta sulla ricerca della chiave può essere parallelizzabile e, di conseguenza, risolvibile in tempi più brevi.
- Il tempo T di ricerca della chiave è solo stimato, infatti può risultare più grande o più piccolo in base all'ordine con cui le chiavi sono esaminate.

- La funzione logaritmica è una funzione che cresce molto lentamente con valori molto grandi. Di conseguenza, con deadline molto distanti tra loro, si avranno chiavi di lunghezza praticamente uguali.

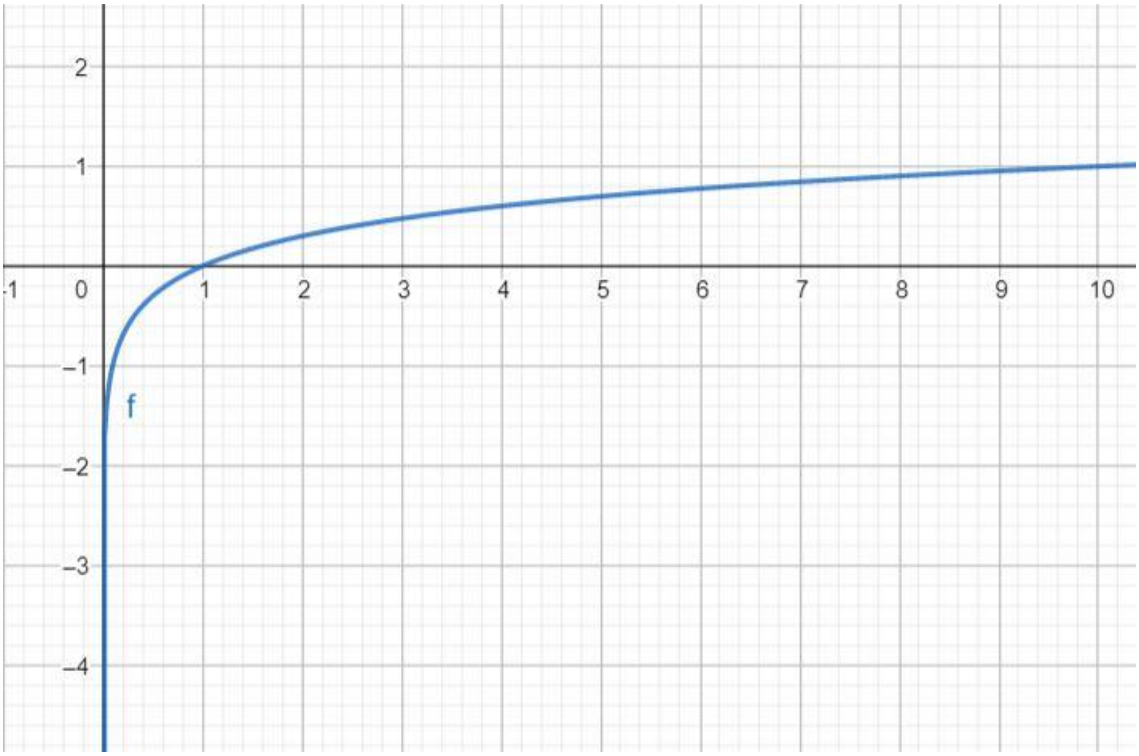


Figura 2 - Rappresentazione della funzione logaritmica

Per valutare l'efficacia di questo puzzle, ho utilizzato un implementazione in Javascript di RC5 con chiave a 128 bit. La mia macchina effettua in media 7000 decifrature al secondo. Sia $S = 7000$ e $T = 7200$ secondi (due ore), abbiamo una chiave $|k| = \log(2 * 7000 * 7200) \cong 8\ bit$. E' facile intuire che RC5 con chiave da 8 bit è facilmente rompibile in meno di due ore (soprattutto se la ricerca della chiave viene fatta in parallelo). Di conseguenza, da questa analisi emerge che anche questo approccio non è adeguato.

3.3 Puzzle basato sui quadrati ripetuti

Rivest et al., in [1], proposero una nuova soluzione per risolvere il problema precedente: la creazione di un problema computazionale basato sul calcolo ripetuto di quadrati e sul sistema di cifratura a chiave pubblica RSA. Come vedremo tra poco, l'operazione di calcolo di quadrati ripetuti è prettamente sequenziale e non è possibile parallelizzarla per ridurne i tempi di risoluzione.
Supponiamo che Alice abbia un messaggio M che vuole cifrare per un periodo di T secondi.
Le fasi di creazione del puzzle sono:

- Alice genera un numero $n = p * q$, tale che n sia il prodotto di due numeri primi casuali molto grandi.

- Alice calcola il toziente di n: $\varphi(n) = (p-1) * (q-1)$. Il toziente di n corrisponde al numero di interi positivi minori di n che non hanno divisori in comune con n.
- Alice calcola $t = T * S$, dove S è il numero di quadrati modulo n al secondo che possono essere calcolati da Bob.
- Alice genera una chiave casuale K per un crittosistema, come RC5. Questa chiave deve essere abbastanza lunga (160 bit o più). In questo modo, la sua ricerca risulta infattibile prima che i T secondi trascorrano.
- Alice cifra M con la chiave K utilizzando l'algoritmo RC5. Ottiene così il testo cifrato $Cm = RC5(K, M)$.
- Alice genera casualmente un valore a (con $1 < a < n$) tale che $MCD(a, n) = 1$, e cifra la chiave K come $Ck = K + a^{2^t} \ (mod\ n)$.
  Per fare questo efficentemente, prima computa $e = 2^t \ (mod\ \varphi(n))$ e dopo computa $b = a^e \ (mod\ n)$.
- Alice produce come output il Time-Lock Puzzle (n, a, t, Ck, Cm) ed elimina ogni altra variabile (come $p\ e\ q$).
- Bob, per ottenere la chiave K, deve effettuare la seguente operazione $K = -CK + a^{2^t} \ (mod\ n)$ in almeno T secondi. Ma come vedremo, questo non è corretto.

E' necessario fare alcune considerazioni:

- A differenza del modello precedente, la ricerca della chiave K non può essere effettuata tramite un attacco a forza bruta perché richiederebbe un tempo enorme. L'approccio più veloce è determinare $b = a^{2^t} \ (mod\ n)$. Conoscere il $\varphi(n)$ permetterebbe di ridurre 2^t a e, $mod\ \varphi(n)$, cosicchè b possa essere computato più efficacemente. Il problema è che, conoscere il toziente di n, equivale a fattorizzare n, il quale è a sua volta un problema quasi impossibile per n molto grandi.
- Questo problema non può essere parallelizzabile. Il calcolo dei quadrati ripetuti è solo ed esclusivamente sequenziale.
- Il numero t di quadrati richiesti per risolvere il puzzle può essere controllato, creando diversi livelli di difficoltà. Questo necessita una conoscenza approfondita della macchina del destinatario.

Ci sono diversi metodi per ridurre la complessità del calcolo dell'esponenziazione modulare: metodo diretto, metodo con memoria efficiente e i metodi binari da destra a sinistra. Usando i primi due casi, il puzzle mantiene la sua efficacia per tutto il tempo T, mentre usando il terzo metodo viene rotto prima della deadline.

3.3.1 Metodo diretto

Sia $x = 2^t$, il metodo diretto prevede di moltiplicare semplicemente il valore 2 per t volte. Richiede $O(t)$ moltiplicazioni per completare il procedimento.
Lo pseudo codice è il seguente:

```
function direct_method(base, exponent, module){
        result := base;
        for (2 to exponent){
                result := result * base;
        }
        return result % module;
}
```

3.3.2 Metodo memoria efficiente

Questo metodo è molto simile al precedente, ma ad ogni passo di moltiplicazione effettua un'operazione di modulo. Questo è utile in caso di valori molto grandi per risparmiare spazio. Anche in questo caso, richiede $O(t)$ di moltiplicazioni per completare il procedimento. Lo pseudo codice è il seguente:

```
function memory_efficient_method(base, exponent, module){
        result := base % module;
        for (0 to exponent - 1){
                result := (result * base) % module;
        }
        return result;
}
```

3.3.3 Metodo binario da destra a sinistra

Questo terzo metodo riduce drasticamente il numero di operazione per il calcolo dell'esponenziazione modulare. Richiede che l'esponente venga convertito in notazione binaria in questo modo:

$$t = \sum_{i=0}^{n-1} a_i * 2^i$$

Dove n è la lunghezza di t in bit. a_i può prendere il valore 0 oppure 1 per ogni i tale che $0 \leq i < n$. Il valore b^t può essere scritto come segue:

$$b^t = b^{(\sum_{i=0}^{n-1} a_i * 2^i)} = \prod_{i=0}^{n-1} b^{a_i 2^i}$$

La soluzione è $c \equiv \prod_{i=0}^{n-1} b^{a_i 2^i} \pmod m$

Lo pseudo codice è il seguente:

```
function right_to_left_method(base, exponent, module){
        result := 1;
        base:= base %  module;
        while (exponent > 0){
                if (exponent % 2 == 1)
                        result:= (result * base) % module;
                exponent:= exponent >> 1;
                base:= (base * base) % module;

        }
        return result;
```

Capitolo 4

Witness encryption

4.1 Introduzione

In questo capitolo verrà discusso una nuova modalità molto diversa dalle precedenti. Liu et al. in [8] hanno applicato il concetto di Witness Encryption ([10]) ad un sistema di Blockchain per garantire la Time-Lock Encryption. La Blockchain su cui è basata è la rete Bitcoin ([9]). Questo sistema è basato sulla possibilità di decifrare un messaggio senza dover eseguire una computazione continua (come i puzzle), ma aspettando il naturale scorrere del tempo fino alla deadline prestabilita.
Prima di entrare nel dettagio del protocollo, farò una breve descrizione della rete Bitcoin. Successivamente verranno introdotte le definizioni di relazione, witness encryption e computational reference clock e analizzato il funzionamento del protocollo. In [11], è stato proposta un'implementazione con l'approfondimento di nuovi concetti: CNF-SAT e Subset-Sum.

4.2 Bitcoin

Bitcoin nasce nel 2009 da Satoshi Nakamoto ([9]) come sistema di pagamento elettronico, basato sull'utilizzo della crittografia per la costruzione di una catena di blocchi contenenti una serie di transazioni digitalmente firmate.
Ogni blocco è costituito queste informazioni:

- Header
  - Versione del blocco.
  - Valore dell'hash calcolato sul blocco precedente (funzione di hash SHA 256 applicata due volte).
  - Istante di tempo in cui l'header è stato creato.
  - Merkle root: radice del merkle hash tree contenente tutte le transazioni del blocco.
  - Difficult target: valore usato nel protocollo di consenso Proof-of-Work.
  - Nonce: valore usato per il calcolo del blocco.
- Body
  - Lista di transazioni sotto forma di merkle hash tree.

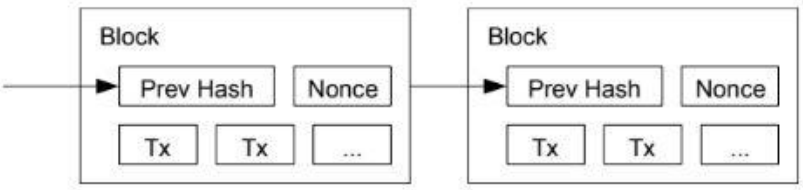


Figura 3 - Rappresentazione della Blockchain Bitcoin con due blocchi tratto da [9]

Ogni utente emette delle transazioni che vengono inserite all'interno della Memory Pool in attesa di essere validate e estratte dei miner. Un nodo, chiamato miner, estrae alcune delle transazioni disponibili, le valida verificando la firma digitale del mittente e computa un blocco che le contiene. La creazione dei blocchi è regolata da un protocollo di accordo chiamato Proof-of-Work, secondo il quale un blocco è valido se il miner trova un certo valore nonce, contenuto nell'header, tale che

$$H(T_1 \,||\, Tn \,||\, nonce \,||\, Hprev) < target$$

dove $T_1, \dots, T_n$ sono le transazioni contenute nel blocco, $Hprev$ è il valore di hash ottenuto applicando lo SHA256 al blocco precedente e il $target$ è un valore molto piccolo (difficult target).

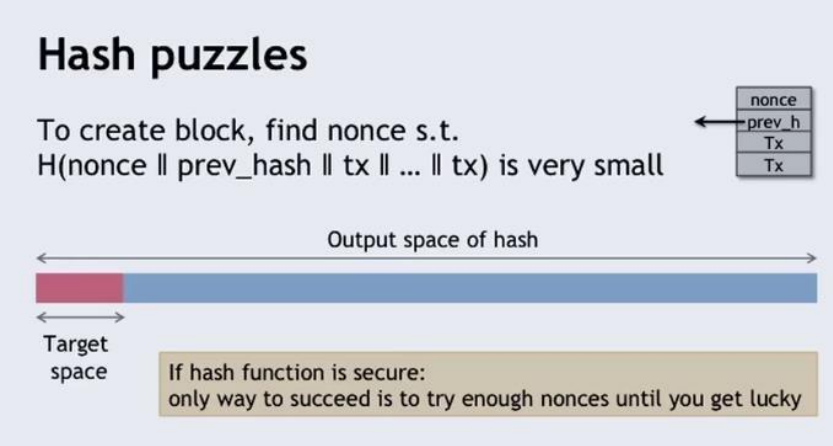


Figura 4 - Rappresentazione del valore target e dello spazio della funzione di hash

Il valore target viene adattato dalla rete dinamicamente al fine di evitare che un miner possa investire più risorse (rispetto agli altri) per creare blocchi validi più velocemente e monopolizzare il sistema. La rete fa in modo che il tempo di generazione di un blocco sia sempre di circa 10 minuti.

Conoscenze preliminari

4.3.1 Definizione. R è una relazione NP, se esiste un algoritmo deterministico tale che, su un input (x, w), restituisce 1 se e solo se $(x, w) \in R$. Entrando nel dettaglio, definiamo R come una relazione NP se:

- Per $x \in N$, gli statement hanno la forma di 1^x.
- Qualsiasi valida blockchain Bitcoin $w = (B_1, \dots, B_x)$ di lunghezza almeno x è un witness per lo statement 1^x, con $(1^x, w) \in R$.

4.3.2 Definizione. Uno schema di witness encryption ([10]) per una relazione NP R, consiste nei seguenti algoritmi:

- $WE.Enc\ (1^\lambda, x, m)$: è un algoritmo di cifratura che prende in input un parametro di sicurezza 1^λ, una stringa x di lunghezza non fissata, e un messaggio $m \in M$, e restituisce un testo cifrato c.
- $WE.Dec(c, w)$: è un algoritmo di decifratura che prende in input un testo cifrato c e un vettore di bit w, e restituisce un messaggio m.

- **Proprietà di Correttezza.** Per qualsiasi $(x, w) \in R$, abbiamo che
$$\Pr[(WE.Dec(WE.Enc(1^{\lambda}, x, m), w) = m] = 1$$

La witness encryption gode di una proprietà chiamata extractability. L'extractability afferma che, quando l'avversario può distinguere due testi cifrati, ottenuti da due differenti messaggi con la stessa istanza x, allora è in grado di conoscere il witness w dell'istanza.

4.3.3 Definizione. Un computational reference clock (CRC) è una macchina probabilistica a stati $C(1^k)$ che prende in input un parametro di difficoltà k e restituisce una sequenza infinita $w_1, w_2, \ldots$ CRC esegue un algoritmo probabilistico f_c che computa $w_t = f_c(w_{\tau-1})$ e restituisce τ. Scriviamo $w_\tau \leftarrow^{\$} C(1^k, \tau)\ con\ \tau \in N$ per abbreviare il processo di esecuzione del clock τ volte di fila, partendo dallo stato iniziale w_0, e restituendo lo stato w_τ di C dopo t esecuzioni. Intuitivamente, w_τ rappresenta il "witness".
Diciamo che R è associata a C, se r è una relazione NP, e per tutti $x <= \tau$ allora

$$\Pr[(1^x, w_\tau) \in R : w_\tau \leftarrow C(1^k, \tau)] = 1$$

L'obiettivo di questa relazione è descrivere quali valori w_τ sono accettabili come "witness al tempo τ ". Nota che ha senso accettare un witness w_τ anche con tempo precedente con $x <= \tau$. Quindi è richiesto che $(1^x, w_\tau) \in R$ vale per tutti gli $x <= \tau$.
Il computational reference clock è una nuova primitiva, estensione dello modello computazionale standard, che fornisce un metodo realistico per emulare il tempo del "mondo reale" in un modello computazionale. Il sistema Bitcoin performa computazioni pubbliche, iterative e su larga scala, dove i miner contribuiscono ad estendere la blockchain della Bitcoin. Di conseguenza, la blockchain può essere rappresentata come un reference clock, dove la lunghezza corrente τ della catena rappresenta il tempo corrente, e ci sono circa 10 minuti tra ogni "clock tick".

4.3.4 Definizione. Lo schema Time-Lock encryption per un Computational Reference clock $C(1^k)$, con uno spazio dei messaggi M, consiste in due algoritmi $(TL.Enc, T.Dec)$:

- L'algoritmo di cifratura $c \leftarrow TL.Enc(1^{\lambda}, \tau, m)$ prende in input un parametro di sicurezza λ, un intero $\tau \in N$ e un messaggio $m \in M$ e restituisce un testo cifrato c.
- L'algoritmo di decifratura $TL.Dec(w, c)$ prende in input $w \in \{0,1\}^*$ e un testo cifrato c, e restituisce un messaggio $m \in M$ o un simbolo di errore.
- **Proprietà di Correttezza**. Per la correttezza è richiesto che :
    - $c \leftarrow TL.Enc(1^{\lambda}, \tau_{dec}, m)$.
    - $w_\tau \leftarrow C(1^k, \tau)$.
    - $m' := TL.Dec(w_\tau, c)$.
    - $Pr(m = m') = 1$.

  Per tutti $\lambda \in N$, tutti i $\tau \in N$, con $\tau >= \tau_{dec}$, e tutti i $m \in M$.

4.3 Funzionamento protocollo

Lo schema di Time-Lock encryption definito precedentemente può essere istanziato sulla base delle caratteristiche della rete Bitcoin. Una blockchain Bitcoin è rappresentata (in maniera semplificata) come una sequenza di tuple $(T_1, r_1, D_1, B_1), \dots, (T_s, r_s, D_s, B_s)$ tale che $B_i := H(T_i,\ r_i, D_i, B_{i-1})$ dove H è una funzione di hash SHA-256 e $B_1, \dots, B_s$ sono i blocchi e B_0 è chiamato blocco genesi.
$H : \{0,1\}^* \rightarrow \{0,1\}^d$ è una funzione di hash per una qualche costante d, $\beta \in \{0,1\}^d$ è il blocco d'inizio e δ: $N \rightarrow [0, 2^{d-1}]$ è la funzione target.

Sia $R_{\beta,\delta}$ la relazione con $(1^x, w) \in R_{\beta,\delta}$ se e solo se $w = ((T_1, r_1, D_1, B_1), \dots, (T_\tau, r_\tau, D_\tau, B_\tau))$ e soddisfa le seguenti proprietà:

- $|T_i|\ e\ |r_i|$ sono limitate polinomialmente, e $D_i \in [0, 2^{d-1}]$.
- W contiene almeno x tuple (T_i, r_i, D_i, B_i).
- $B_1 = H(T_1, r_1, D_1, \beta)$.
- $B_i = H(T_i,\ r_i, D_i, B_{i-1})$ per tutti $i \in [2, x]$.
- $\delta(i) >= B_i$ per tutti $i \in [1, x]$ dove noi interpretiamo stringhe di bit $B_1, \dots, B_x$ canonicamente come interi.

Applicando il parametri di Bitcoin, sia H: $\{0,1\}^* \rightarrow \{0,1\}^{256}$ la funzione di hash SHA256 e sia $R_{\beta,\delta}$ la relazione (definita sopra), istanziata con H e i parametri seguenti $\beta\ e\ \delta$:

- $\beta := B_0$, dove B_0 è il blocco genesi di Bitcoin.
- Fisso la funzione target δ: $N \rightarrow [0, 2^{256} - 1]$.

Sia f_{btc} la funzione che espande la blockchain Bitcoin. Su un input

$$w_{\tau-1} = (T_1, r_1, D_1,\ B_1), \dots, (T_{\tau-1},\ r_{\tau-1}, D_{\tau-1}, B_{\tau-1})$$

e un input ausiliario $aux = T_\tau$, computa e restituisce il nuovo stato w_τ, tale che $w_\tau = (T_1, r_1, D_1,\ B_1), \dots, (T_\tau,\ r_\tau, D_\tau, B_\tau)$ con $H(T_\tau,\ r_\tau, D_\tau, B_{\tau-1}) = B_\tau <= D_\tau$, dove D_τ è il corrente $target$ di Bitcoin.

Sia C_{btc} il Computational Reference Clock che computa f_{btc}. Lo stato di C_{btc} al tempo τ consiste delle prime τ tuple della blockchain. Assumendo che la corrente lunghezza della catena è di τ tuple, e che nuovi blocchi saranno continuamente trovati ogni 10 minuti circa, quindi sappiamo che in circa $10 * x$ minuti la blockchain conterrà $t + x$ tuple.

Le relazioni $R_{\beta,\delta}$ sono associate a C_{btc}, con $\beta = B_0$ e δ soddisfa $\delta(i) >= B_i$ per tutti $i \in N$. Sfortunatamente quest'ultima relazione non è garantita, ma dipende dalla scelta della funzione target δ e del futuro sviluppo della dimensione del target di Bitcoin. Quindi δ deve essere scelto attentamente. Sia x tale che $x > \tau$, nessuno witness w per $(1^x, w) \in R_{\beta,\delta}$ è contenuto nella blockchain.

E' importante sottolineare che ogni sequenza $w = (T_1, r_1, D_1,\ B_1), \dots, (T_x,\ r_x, D_x, B_x)$ computata dalla rete Bitcoin è solo un potenziale witness per $(1^x, w) \in R_{\beta,\delta}$. Questo perché la relazione $R_{\beta,\delta}$ dipende dalla funzione target δ. Di definizione, w sarà solo un witness per $(1^x, w) \in R_{\beta,\delta}$, se $B_i <= \delta(i)$ per tutti gli $i \in [1, x]$. Potrebbe succedere

che allo stesso punto $\gamma \in [\tau + 1, x]$, il target Bitcoin aumenta a un valore D_γ tale che $D_\gamma >= \delta(\gamma)$. In questo caso avremo che $(1^x, w)$ non appartiene a $R_{\beta,\delta}$. Per ovviare al problema, è necessario scegliere δ accuratamente. Tuttavia:

- δ non deve essere troppo piccola, altrimenti ci troveremmo nel caso in cui esiste $i \in N$ tale che $\delta(i) < B_i$. Il nostro C_{btc} non fornirebbe un witness w_i per $(1^i, w_i) \in R_{\beta,\delta}$ e di conseguenza il time-lock encryption non sarebbe corretto.
- Al contrario, δ non deve essere troppo grande, altrimenti $\delta(i) > B_i$ vale per tutti $i \in N$. Quindi, diventa molto facile computare witness w_i per $(i, w_i) \in R_{\beta,\delta}$, semplicemente valutando la funzione di hash H i-volte per costruire una catena lunga i.

Idealmente δ dovrebbe essere scelta in modo che corrisponda perfettamente a D_i per ogni $i > t$. Sfortunatamente D_i dipende dalle risorse computazionali dei miner presenti nella rete. Di conseguenza occorre predire l'evoluzione di D_i, o almeno tentare di approssimarlo così da avere $\delta(i) = D_i - \epsilon_i$ con ϵ_i molto piccolo.
L'articolo [8] getta le basi di questo nuovo sistema introducendo due concetti fondamentali, quali il CNF-SAT e il Subset-Problem. Tuttavia, non fornisce alcuna implementazione, ma solo qualche spunto. Solo successivamente, in [11] verranno ripresi questi ultimi concetti e verrà proposta un'implementazione dello schema.

4.4 Implementazione

In questo paragrafo analizzerò l'implementazione offerta nell'articolo [11] da Wesley Smith introducendo i concetti di CNF-SAT e Subset-Problem.

4.4.1 Definizione. Una formula proposizionale è in forma CNF quando è composta da un insieme di clausole disgiuntive connesse da operatori di congiunzione.
La struttura è la seguente: $(\ldots or \ldots)\ and\ (\ \ldots or \ldots)$.

4.4.2 Definizione. Il CNF-SAT è un problema che riguarda la soddisfacibilità di una formula in forma CNF. La formula è soddisfacibile se esiste una qualche valutazione delle variabili contenute al suo interno che la rendono tale.

Quello che ci interessa è effettuare una conversione della blockchain in una formula proposizionale in forma CNF. Le principali relazioni e vincoli da convertire sono i seguenti:

- $B_i \leq D_i$.
- $B_i = H(T_i, r_i, D_i, B_{i-1})$, dove H consiste in due applicazioni della funzione di hash $SHA - 256$. Questo vincolo verrà semplificato in $B_i = H(B_{i-1})$.

In [11] viene fornita una rappresentazione in formula proposizionale del primo vincolo $B_i \leq D_i$:

$$(not(B_{i,1})\ and\ D_{i,1})\ or\ (((B_{i,1}\ and\ D_{i,1})\ or\ (not(B_{i,1})\ and\ not(D_{i,1})))\ or\ ((not(B_{i,2})\ and\ D_{i,2})$$

$$or\ (((B_{i,2}\ and\ D_{i,2})\ or\ (not(B_{i,2})\ and\ not(D_{i,2})))\ \ldots\ (not(B_{i,256})\ and\ D_{i,256})\ or$$

$$(((B_{i,256}\ and\ D_{i,256})\ or\ (not(B_{i,256})\ and\ not(D_{i,256})))))))$$

Chiaramente non si tratta di una formula in forma CNF. Siccome la conversione tramite leggi di DeMorgan richiede una crescita esponenziale della formula, si possono fare delle restrizione sul dominio di D. $D_i \in \{2^n - 1 : 0 < n \leq 256\}$. Tutti gli elementi di questo dominio sono della forma $0^a 1^b$, dove $a + b = 256$. Quindi, gli elementi della forma $0^a\{0,1\}^b$ sono minore o uguale di $0^a 1^b$: segue che confrontare una stringa con una della forma $0^a 1^b$ richiede solo il verificare che i primi a bit siano 0:

$$(not(B_{i,1})\ and\ not(B_{i,2})\ and\ \dots\ not(B_{i,a}))$$

Per semplicità, i parametri di difficoltà vengono scelti dal dominio ristretto. Il secondo vincolo: $B_i = H(T_i, r_i, D_i, B_{i-1})$ viene convertito in modo simile.

Una volta effettuata la conversione, in [1] viene affrontato il problema del Subset-Sum.
4.4.3 Definizione. Dato un vector multiset $\triangle = (v_i : l_i)$ e un target sum s, esiste un sottoinsieme di $\triangle$ che sommato è uguale a s (dove ogni vettore v_i può essere usato al massimo l_i volte).

Matematicamente:

$\sum_{v_i \in \triangle} b_i * v_i = s$ per qualche b_i, dove $0 \leq b_i \leq l_i$

Il prossimo passo è costruire $\triangle$, il multiset di vettori risultante dalla conversione del problema CNF-SAT a un Subset-Sum problem.
Per la costruzione dei vettori v_i (componenti del problema di Subset-Sum) sono necessari i seguenti elementi:

- n: il numero di variabili $x_1, \dots, x_n$ in forma CNF.
- k: il numero di clausole nell'espressione.
- m_i con $1 \leq i \leq k$: il numero di letterali nella clausola k.

Vengono costruiti i seguenti vettori binari, tutti di lunghezza $(n + 2 * k)$:

- $u_1, u_2, \dots, u_n$: l'i-esimo elemento di ogni u_i è 1, e ogni $n + j$-esimo elemento è 1 se e solo se il letterale x_i occorre positivamente nella clausola j. Tutti gli altri elementi sono 0.
- $\sim u_1, \sim u_2, \dots, \sim u_n$: l'$i$-esimo elemento di ogni $\sim u_i$ è 1, e ogni $n + j$-esimo elemento è 1 se e solo se il lietterale x_i occorre negativamente nella clausola j. Tutti gli altri elementi sono 0.
- $v_1, v_2, \dots, v_k$: l'$n + i$-esimo e $n + k + i$-esimo elementi di v_i sono 1. Tutti gli altri elementi saranno 0.
- $z_1, z_2, \dots, z_k$: l'$n + k + i$-esimo elemento di ogni elemento di z_i è 1. Tutti gli altri sono a 0.

Dobbiamo costruire un target vector s di interi di lunghezza $n + 2 * k$.

- $1 \leq i \leq n$: $s_i = 1$.
- $(n + 1) \leq i \leq (n + k)$: $s_i = m_i$.
- $(n + k + 1) \leq i \leq (n + 2 * k)$: $s_i = m_i - 1$.

Quindi:

- I vettori u e $\sim u$ corrispondono alle variabili $x_1, \dots, x_n$ nella forma CNF. Specificatamente i due possibili assegnamenti di ogni variabile.
- I vettori v_j corrispondono ai letterali non soddisfatti $(= 0)$ nella clausola j.
- Mentre z_j corrisponde a un letterale soddisfatto nella clausola j.

Il teorema di correttezza di Subset-Sum è il seguente: Una istanza CNF-SAT x è soddisfacibile se e solo se esiste una soluzione al suo corrispondente problema Subset-Sum costruito sopra.

Per ridurre la dimensione dei vettori, la soluzione migliore è usare delle liste di interi che contengono gli indici dei vettori che contengono valori 1. (facciamo la compressione dei vettori vecchi togliendo gli zeri).

In [11] viene pubblicato il seguente pseudo codice di generazione del vettore u.

```
Data   : btcSAT.cnf with clauses C_1...C_k
Input  : int n, int k
Output: small integer lists u_1...u_n
%create empty lists for each vector
for i from 1 to n do
 | u_i ← []
end
%for each variable in each clause, append the clause index to the
 relevant list
for j from 1 to k do
 | for x_m ∈ C_j do
 |  | u_m.append(n + j)
 | end
end
```

Algorithm 5: makeVecs v3

Per quanto riguarda la costruzione di l_i di $\triangle$, questo denota il numero massimo di volte che il vettore v_i può essere usato per creare il target sum.

Il modo in cui i vettori sono costruiti rende la costruzione di l_i semplice. Consideriamo le seguenti cose:

- u_i e $\sim u_i$ corrispondono agli assegnamenti opposti dello stesso letterale. Un witness assegnerà ogni letterale esattamente uno; quindi per u_1, …, u_n e $\sim u_1, \dots, \sim u_n$, $l_i = 1$.
- v_j corrisponde a un letterale non soddisfatto nella clausola j. Ci sono m_j letterali nella clausola j, e naturalmente una valutazione soddisfacente può avere al massimo $m_j - 1$ di loro valutati false. Cosi $l_{v_j} = m_j - 1$.
- z_j corrisponde a un letterale non soddisfatto nella clausola j diverso dal primo. Ci sono m_j letterali nella clausola j, e naturalmente una valutazione può avere al massimo m_j di loro valutati a vero. Cosi $l_{z_j} = m_j - 1$.

Poiché il problema che ci interessa è il Subset-Sum, un witness consiste di coefficienti $(b_0, \dots, b_{|\triangle|})$ tale che $\sum_{v_i \in \triangle} b_i * v_i = s$.

Di seguito le regole per la costruzione di ogni coefficiente b_i del witness dalla blockchain attuale:

- u_i: $b_{u_i} = 1$ se l'i-esimo bit valore di hash del blocco è 1, altrimenti è 0.
- $\sim u_i$: $b_{\sim u_i} = 1$ se l'i-esimo bit valore di hash del blocco è 0, altrimenti 1.
- Tutto v_j: attraversa la formula CNF. Per ogni clausola j, per ogni letterale non soddisfatto aggiungi uno a b_{v_j}.
- Tutto z_j: attraversa la formula CNF. Per ogni clausola j, per ogni letterale soddisfatto oltre il primo aggiungi uno a b_{z_j}.

4.4.4 Esempio

Prendiamo questo esempio tratto da [11].

Abbiamo una blockchain formata da tre bit con, di conseguenza, tre vettori $u_1, \dots, u_3$ e tre vettori $\sim u_1, \dots, \sim u_3$. La rappresentazione CNF è la seguente:

$$\left(x_1 \bigvee x_2\right) \bigwedge \left(x_1 \bigvee \sim x_2\right) \bigwedge \left(x_1 \bigvee x_3\right) \bigwedge \left(x_2 \bigvee x_3\right)$$

Avendo quattro clausole, abbiamo quattro vettori $v_1, \dots, v_4$ e quattro vettori $z_1, \dots, z_4$. Assumiamo inoltre che il valore di hash del presente blocco sia uguale a 101. Da questo possiamo computare un witness utilizzando le regole precedenti: sappiamo $x_1 = 1$, $x_2 = 0$, $x_3 = 1$, quindi $b_{u_1} = b_{u_3} = 1$ e $b_{u_2} = 0$. Allo stesso modo $b_{\sim u_1} = b_{\sim u_3} = 0$ e $b_{\sim u_2} = 1$. Stesso discorso per gli altri vettori.

L'idea è di prendere un'istanza per la cifratura come formula proposizionale, e un witness usato per la decifratura che soddisfa la valutazione dei letterali.
Un esempio banale è il seguente:

$X = (x_1 \; or \; x_2) \; and \; (x_3 \; or \; x_4) \; and \; (x_1 \; or \; x_4)$.

Un witness a questa istanza è una valutazione dei letterali $x_1 \dots x_4$ tale che la valutazione sia vera: per esempio $[x_1 = 1, x_2 = 0, x_3 = 0, x_4 = 1]$. Quindi, per qualche $c \in C, m \in M$:

- WE.encrypt($1^p, ((x_1 \; or \; x_2) \; and \; (x_3 \; or \; x_4) \; and \; (x_1 \; or \; x_4)), m) = c$.
- WE.decrypt($c, [x_1 = 1, x_2 = 0, x_3 = 0, x_4 = 1]) = m$.

Chiaramente questa è una versione semplificata che non prevede ancora l'introduzione del Subset-Sum. Con $\triangle$, il prossimo passo è implementare uno schema di witness encryption associato a Subset-Sum.
Il problema del logaritmo discreto può essere visto come un esempio di witness encryption. Dato g, h e un gruppo G, trovare x, tale che $g^x = h$, può essere visto come x per un witness al problema generale $g^? = h$. La nostra istanza è un Subset-Sum: questo corrisponde alla lunghezza desiderata della blockchain, lo starting block, e i parametri di difficoltà.

Richiediamo un witness encryption scheme da applicare a $\triangle$. Un encryption scheme deve lavorare per uno specifico $\triangle$ che può essere generalizzato.

Originariamente in [8] venne proposto il seguente witness encryption scheme:

- $\triangle$: un Subset-Sum con target vector s (tutti i vettori hanno lunghezza d).
- $e()$: l'operazione di mapping corrispondente a un set di mappe crittografiche multilineari associate a $\triangle$.
- w: un witness associato a $\triangle$.
- a: vettore casuale di lunghezza d.
- a^{v_i}: l'operazione ${a^{v_{i,1}}}_1 * \ldots * {a^{v_{i,d}}}_d$.
- g_v: un generatore di un gruppo corrispondente a un vettore v.

La seguente coppia di algoritmi costituisce il witness encryption:

- WE.encrypt($\triangle$, m): scegli un casuale a e genera un insieme di mappe crittografiche multilineari compatibile con $\triangle$ (denotato come params). Quindi il ciphertext $c = (params, \{g_{v_i}^{a^{v_i}}\}_{i \in \triangle}, m * g_s^{a^s})$.
- WE.decrypt(c, w): computa la chiave K:
$$K = e(g_{v_1}^{a^{v_1}}, \ldots, g_{v_1}^{a^{v_1}}, g_{v_2}^{a^{v_2}}, \ldots, g_{v_2}^{a^{v_2}}, g_{v_{|\triangle|}}^{a^{v_{|\triangle|}}})$$
Quindi può essere usato per provare e invertire $m * g_s^{a^s}$.

Segue che se un corretto witness $w = b_0, \ldots, b_{|\triangle|}$ è fornito,

$$K = g_{\sum b_i * v_i}^{a^{\sum b_i * v_i}} = g_s^{a^s}$$

Tuttavia, in [11] viene implementato in maniera più completa tramite l'uso di mappe lineari su interi.

Sulla base dello schema CLT, i vettori sono codificati come interi come segue:

- $a = (a_1, \ldots, a_n)$: un vettore che viene codificato.
- $p_1, \ldots, p_n$: primi segreti.
- z: un casuale intero modulo $\prod_{i=1}^{n} p_i$.
- $g_1, .., g_n$: piccoli segreti primi.
- $r_1, \ldots, r_n$: piccoli interi casuali (chiamato rumore).

Quindi un level-k encoding di m è un intero d tale che:

$$\bigvee i \in [1, n]: d = \frac{r_i * g_i + a_i}{z^k} \; mod \; p_i$$

Level-k si riferisce al livello di multilinearità della codifica, il più importante aspetto di questo concetto è che accoppiando un level-$k_1$di codifica con un livello k_2 di codifica risulta una codifica di livello $k_1 + k_2$.

L'idea è di usare la mappa multilineare per realizzare lo scambio di chiavi di Diffie-Hellman. Moltiplicando i risultati codificati sommando i loro livelli di multilinearità, possiamo usare una mappa multilineare level-k per fare un Diffie-Hellman multiparte con $k + 1$ utenti come segue:

- Codificare ogni chiave privata di un utente (livello 0) come codifica di livello 1.

- Per ogni insieme di $\binom{k+1}{k}$ level-1 encodings, computa il prodotto come un level-k encodings.
- Ogni utente computa il prodotto della loro chiave privata e il level-k encoding delle altre chiavi degli utenti. Questo sarà lo stesso per tutti, il segreto condiviso.

Tramite questo modello possiamo definire una forma primitiva di witness encryption a un problema Subset-Sum attraverso mappe crittografiche multilineari. Questo witness encryption è differente da [6], infatti risultava infattibile.
Invece di codificare vettori, generiamo interi corrispondenti a questi vettori e creiamo codifiche di livello 1. Da queste codifiche di livello 1 e un witness noi generiamo una codifica di livello k (dove k è $\sum_{i\in|\triangle|} b_i$) che corrisponde al target sum e lo usiamo per cifrare un messaggio. Dall'altro lato, il decifratore usa la sua conoscenza sul witness per ricomputare la codifica di livello k.

Sia l'encrypter sia il decrypter computano la chiave K come segue:

$$K = s_{enc} = e(g_1, \dots, g_1, g_2, \dots, g_2, \dots, g_{|I|}, \dots, g_{|I|})$$

Dove g_i è una codifica di livello 1. Ogni g_i è costruito dalla codifica casuale di livello 0 a_i corrispondente a $v_i \in \triangle$ e $w = b_1, b_2, \dots$ è un witness associato a $\triangle$. Come detto prima, $e()$ è l'operazione di mapping. Ci riferiamo alla chiave come s_{enc} perché in un certo senso rappresenta una codifica del vettore target s del problema Subset-Sum. Mentre s è computato come somma di vettori, s_{enc} è computato dall'accoppiamento di codifiche.

Di seguito lo pseudo codice, fornito in [11], del sistema di cifratura e decifratura:

Input : (params), $\Delta = \{v_i : l_i\}_{i \in I}$, target s, witness $w = (b_1, b_1 ... b_{|I|})$, *randomness* $(\alpha_1, \alpha_2, ... \alpha_n)$
Output: Message m corresponding to ciphertext c

$s_{enc} \leftarrow 1$

`%encode received randomness`

for *i from 1 to* $|I|$ **do**
  $g_i \leftarrow params.encode(\alpha_i)$
end

`%use witness to recompute top-level encoding`

for *i from 1 to* $|I|$ **do**
  for *j from 1 to* b_i **do**
    $s_{enc} \leftarrow s_{enc} * g_i$
  end
end

`%use recomputed top-level encoding to recover message from ciphertext`

$m = c - s_{enc}$
return m

Algorithm 7: Pseudocode for witness decryption under our scheme

Input : (params), $\Delta = \{v_i : l_i\}_{i \in I}$, target s, witness $w = (b_1, b_1 ... b_{|I|})$, message m
Output: Ciphertext c corresponding to target sum s, randomness $[\alpha_1, \alpha_2, ..., \alpha_{|I|}]$

`%generate random parameters`

$[\alpha_1, \alpha_2, ..., \alpha_{|I|}] \leftarrow random$
$s_{enc} \leftarrow 1$

`%compute top-level encoding by pairing encodings representing each vector the amount of times dictated by the witness`

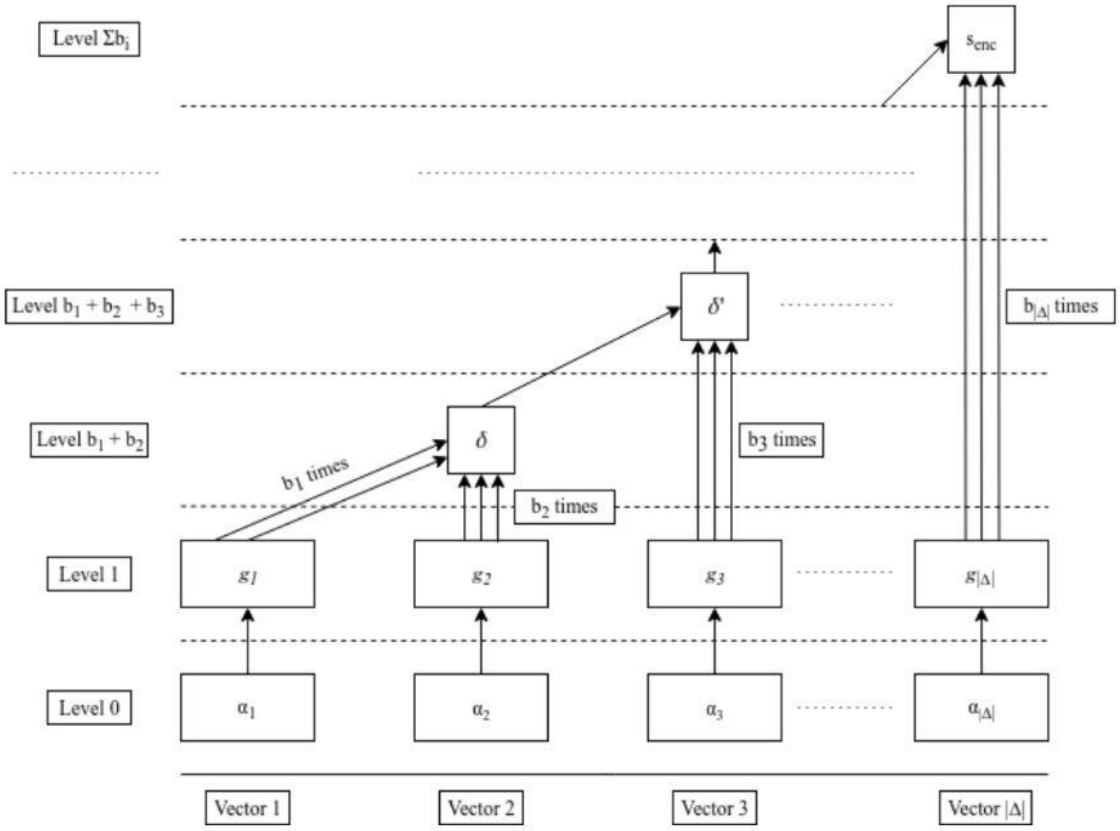
for *i from 1 to* $|I|$ **do**
  $g_i \leftarrow params.encode(\alpha_i)$
end
for *i from 1 to* $|I|$ **do**
  for *j from 1 to* b_i **do**
    $s_{enc} \leftarrow s_{enc} * g_i$
  end
end

`%use top-level encoding corresponding to witness to create and publish ciphertext`

$c = m + s_{enc}$
return $c, [\alpha_1, \alpha_2, ..., \alpha_{|I|}]$

Algorithm 6: Pseudocode for witness encryption under our scheme

Come visibile dallo pseudocodice, da un vettore α di interi casuali vengono generati gli interi primi g_i con qualche metodo di *encoding*. Dal witness, viene creato s_{enc} effettuando un prodotto per ogni g_i.



La precedente immagine ([11]) è una rappresentazione del funzionamento del sistema di cifratura per generare s_{enc} come chiave di cifratura/decifratura. Al livello 0 viene utilizzato il vettore α di interi casuali di dimensione $|\Delta|$. Al livello 1, ogni α_i viene codificato per generare un primo g_i. D'ora in avanti, per ogni livello successivo, si moltiplica ogni g_i per se stesso b_i volte e si somma a coppie per ottenere δ. Il procedimento procede sommando δ del livello precedente per il prossimo g_i moltiplicato per stesso b_i volte. All'ultimo livello $\sum_i b_i$, si ottiene s_{enc} che corrisponde alla chiave di cifratura. Alla fine, il testo cifrato sarà $c = m + s_{enc}$.

4.4.5 Esempio

Illustrerò un semplice esempio per mostrare il funzionamento di questo protocollo.
Prendiamo i seguenti valori:

- Witness $w = [1, 3, 1]$.
- Messaggio $m = 6$.
- $\alpha = [200, 32, 67]$.
- α_1 viene mappato in $g_1 = 3$, α_2 viene mappato in $g_2 = 7$, α_3 viene mappato in $g_3 = 5$.

Al livello 4 (1 + 3) abbiamo $\delta = 3 * 1 + 7 * 7 * 7 = 346$. Al livello 5 $(1 + 3 + 1)$ abbiamo $s_{enc} = 346 + 5 * 1 = 351$.
Il cifrato diventa $c = m + s_{enc} = 6 + 351 = 356$.
Per la decifratura, il destinatario necessita di tutte le informazioni di base per ricostruire s_{enc}.

4.5 Problemi

Come facilmente intuibile, la cifratura e la decifratura richiedono l'utilizzo del witness. Questo contraddice l'originale funzionamento del witness encryption, dove solo la decifratura lo richiedeva. Dalla definizione del witness, questo è ottenibile solo al momento in cui la blockchain ha ottenuto la lunghezza desiderata e la decifratura è prontamente fattibile. Ciò significa che cifrare un messaggio richiede di aspettare lo stesso tempo richiesto per decifrare. Da questo assunto, l'implementazione proposta non è efficace ed utilizzabile.

Capitolo 5

Time-lock encryption basato su curve ellittiche

5.1 Introduzione

In [12] Lai et al. hanno proposto un nuovo sistema di time-lock encryption, basato sull'utilizzo delle curve ellittiche con cifrario asimmetrico ElGamal in ambiente Ethereum. In questo capitolo verrà fornita un'introduzione all'ambiente Ethereum e all'utilizzo delle curve ellittiche applicate al sistema di cifratura ElGamal. Successivamente verrà svolta un'analisi della nuova tecnica di time-lock encryption presentata in [12] e verrà fornita una personale implementazione con una comparazione dei risultati ottenuti con i risultati dell'articolo originale. Infine, verrà fatto approfondimento dei vantaggi e svantaggi e delle eventuali migliorie apportabili.

5.2 Ethereum

Ethereum ([13]) è un sistema decentralizzato rilasciato nel 2015 con un funzionamento molto simile a Bitcoin. Come per Bitcoin, chiunque voglia aggiungere un blocco alla catena deve risolvere un puzzle che richiede un potere computazionale molto elevato. Tuttavia, a differenza di Bitcoin, ogni blocco viene costruito ogni 12 secondi circa. Ma l'elemento più interessante, e attinente al nostro protocollo, sono gli Smart Contract. Gli Smart Contract sono dei programmi che vengono pubblicati da sviluppatori all'interno della rete Ethereum. Chiunque può richiedere l'esecuzione del codice, o una parte di codice, del contratto attraverso l'emissione di transazioni. Gli utenti possono interagire con le funzioni definite sul contratto inviando transazioni nella rete. Di fondamentale importanza per Ethereum è il concetto di Gas. Il Gas è l'unità che misura l'ammontare di sforzo computazionale richiesto per eseguire delle specifiche operazioni sulla rete Ethereum. Siccome ogni transazione richiede delle risorse computazionali, ogni transazione richiede una tassa. Il Gas si riferisce quindi alla tassa richiesta per condurre con successo una transazione su Ethereum. Il prezzo del Gas è denotato in Gwei, il quale corrisponde a $1 * 10^{-9}$ ETH. Per esempio, invece di dire che un Gas costa 0.000000001 Ether, si può dire che costa 1 Gwei. Richiedere una tassa per ogni transazione evita che un nodo malevolo possa effettuare uno spamming nella rete. In altri termini, l'esecuzione di una funzione di un contratto ha un costo in termini di Gas, e di conseguenza, in termini di Ether. Il costo varia in base al "peso" del codice, cioè in base alla sua complessità, all'ammontare di dati elaborati e al numero di "linee di codice" eseguite. Per questo, è molto importante progettare degli Smart Contract con funzioni che consumino meno risorse di quelle che un nodo guadagnerebbe eseguendole. Questo è uno degli aspetti che verrà preso in considerazione, successivamente, nell'implementazione del nostro Smart Contract.

5.3 Curve ellittiche

In questo paragrafo verrà effettuata una desrizione approfondita delle curve ellittiche e della loro applicazione nella crittografia, in particolare con il cifrario ElGamal. I sistemi basati su curve ellittiche hanno il vantaggio di avere un efficienza comparabile all'algoritmo di cifratura a chiave asimmetrica RSA, utilizzando chiavi di lunghezza inferiore.

5.3.1 Definizione. Un campo finito consiste in un insieme finito di elementi su cui sono definite due operazioni binarie che soddisfano la proprietà associativa e commutativa: addizione e moltiplicazione. Il suo ordine è il numero di elementi che lo compongono. F_p è un campo finito di ordine p composto dall'insieme $\{0, 1, \dots, p-1\}$.

5.3.2 Definizione. La caratteristica di un campo è definita come il più piccolo naturale n tale che $1 + 1 + 1 + \cdots + 1$ (n volte) $mod\ p\ = 0$. Nello specifico, F_p ha caratteristica p con p primo.

Le curve ellittiche E sono descritte da equazioni cubiche generalizzate:

$$y^2 + axy + by = x^3 + cx^2 + dx + e$$

Tuttavia, le curve che a noi interessano sono le curve ellittiche, su campo F_p, descritte dall'equazione di Weyerstrass:

$$y^2 = x^3 + ax + b$$

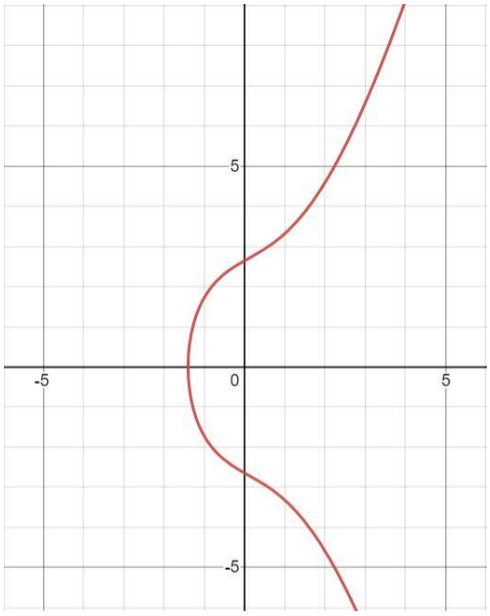


Figura 5 - Curva con a=3 e b=7

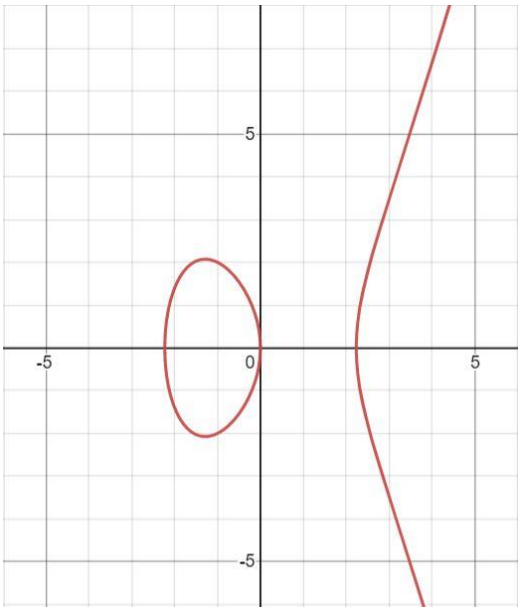


Figura 6 - Curva solo con a=-5

5.3.3 Definizione. Un punto $P(x, y)$ appartiene alla curva $E(a, b)$, definita sul campo F_p, se l'equazione di Weyerstrass è verificata per $p > 2$ e $a, b \in F_p$, con $4a^3 + 27b^2 \neq$

$0\ (mod\ p)$.
Le operazioni sui punti $P(x,y) \in E_p(a,b)$ sono:

- Addizione: $P_1 + P_2 = R$
  Nel caso in cui sia $P_1, P_2 \in E_p(a,b)$ con stesso valore per x, allora $P_1 + P_2 = O$ (punto all' infinito).
  Le regole generali per il calcolo della somma tra punti sono:
  - $x_{P_1+P_2} = \lambda^2 - x_{P_1} - x_{P_2}\ mod\ p.$
  - $y_{P_1+P_2} = \lambda * (x_{P_1} - x_{P_1+P_2}) - y_{P_1}\ mod\ p.$
  - Se $P_1 \neq P_2$ allora $\lambda = (\frac{y_{P_2}-y_{P_1}}{x_{P_2}-x_{P_1}})$ oppure $\lambda = \frac{((3*x_{P_1})^2+a)}{2*y_{P_1}}$ se $P_1 = P_2$.
- Moltiplicazione: $k * P_1 = P_1 + P_1 + \cdots + P_1\ (k\ volte)$.

5.3.4 Definizione. L'inverso additivo di un punto P, corrisponde a $-P = (x, -y)$.

5.3.5 Definizione. $\#E_p(a,b)$ rappresenta il numero di punti P che soddisfano l'equazione più il punto infinito O. Il punto O è quel punto tale che $P_1 + O = O + P_1 = P_1$ e $O = -O$.

5.3.6 Definizione. L'ordine di un punto P è il più piccolo naturale n tale che $n * P = O$.

5.3.7 Definizione. Un punto G si dice punto base o generatore di una curva $E_p(a,b)$, se l'ordine di G è primo e $E_p(a,b) = \{k * G\ |1 \leq k \leq n-1\}$. Detto in altri termini, un punto G è quel punto che, moltiplicato per un valore k, genera tutti i punti della curva.

Nel capitolo 2 è stato affrontato il cifrario Elgamal, il cui schema è basato sulla difficoltà del calcolo del logaritmo discreto (DLP). Le curve ellittiche possono essere applicate al suddetto cifrario, per rendere più complesso il DLP:
Dati $E_p(a,b), G, z$ calcolare x, tale che $x * G = z$.
La chiave pubblica è composta da $[E_p(a,b), G, P_A]$ dove G è un punto generatore della curva $E_p(a,b)$ di ordine n primo e P_A è un punto di $E_p(a,b)$. Mentre la chiave privata corrisponde a un valore $x \in [1, n-1]$.
La chiave pubblica si ottiene nel seguente modo: $P_A = x * G$.
La cifratura di un messaggio $P_M \in E_p(a,b)$ è cosi composta:

- Scegli k casuale $\in [1, n-1]$.
- $C_1 = k * G$.
- $C_2 = P_M + k * P_A$.
- $C = (C_1, C_2)$.

La decifratura di C è cosi composta:

- $P_M = C_2 - x * C_1 = C_2 + inv(x * C_1)$, dove inv è l'inverso additivo di un punto.

5.3.8 Esempio

Prendiamo una curva $E_{23}(1,4)$ con $\#E_{23}(1,4) = 29$ e il generatore $G = (0,2)$. Vogliamo cifrare il messaggio $P_M = (8,15)$.

La chiave pubblica $P_A = 3 * (0,2) = (11,9)$, con $x = 3$.
La cifratura è cosi composta:

- $k = 9$.
- $C_1 = 9 * (0,2) = (4,7)$.
- $C_2 = (8,15) + 9 * (11,9) = (10,5)$.
- $C = ((4,7), (10,5))$.

La decifratura di C è cosi composta:

- $P_M = (10,5) - 3 * (4,7) = (8,15)$.

5.4 Funzionamento protocollo

In [12] viene proposto un primo metodo meno efficiente e poco utilizzato che si basa sull'uso dell'algoritmo di scambio chiavi Dieffie-Hellman (DHKE) e curve ellittiche. Alice genera $x * G$, Bob genera $y * G$, e calcolano $x * y * G$ come "chiave pubblica" per cifrare i messaggi, dove G è un punto base della curva ellittica. Questo approccio permette di cifrare messaggi senza considerare la chiave privata, ma .
I problemi di questo metodo sono i seguenti:

- Una volta che tutti i partecipanti colludono, le informazioni sensibili potrebbero essere pubblicate prima.
- Una volta che un partecipante perde il suo segreto, il messaggio cifrato non verrà mai più aperto da lui.

Per evitare il secondo problema si potrebbe incoraggiare un partecipante tramite un qualche tipo di incentivo. Ed è proprio questo il fattore fondamentale del metodo successivo.
E' il nuovo approccio in [12] che suscita il nostro interesse. Riguarda l'uso di una curva ellittica che abbia un ordine appropriato, tale che il calcolo del logaritmo discreto (DLP) sulla curva non sia computazionalmente infattibile. In altre parole, il DLP deve esssere compromissibile dopo un tempo accettabile e previsto. Questa curva verrà chiamata Semi-Feasible Elliptic Curve (SFEC). In base ai parametri di decifratura stabiliti, è necessario costruire una SFEC che garantisca il tempo richiesto per la sua risoluzione.
Il procollo si basa sull'individuazione di una funzione di hash H_e che mappi una stringa di lunghezza arbitraria in un punto predefinito sulla SFEC:

$$H_e: \{0,1\}^* \rightarrow E(F_q)$$

dove F_q è un campo finito con q primo e E la SFEC selezionata.

Ora, dato un input casuale , determinato da ogni partecipante, in H_e, possiamo ottenere le coordinate di un punto su $SFEC$ che verrà trattato come la chiave pubblica del nostro Time-Lock encryption scheme. E' importante sottolineare che nessuno conosce in quel momento la corrispondente chiave privata. Essendoci diversi partecipanti, ognuno applicherà un certo valore come input nella funzione di hash H_e. Tutti gli i componenti verranno concatenati e inseriti come input della funzione H_e. Per evitare che qualcuno modifichi il proprio valore prima che venga generata la chiave pubblica, viene creato uno Smart Contract vincolante che conterrà tutti i componenti di ogni partecipante.

Dopo la generazione della chiave pubblica, iniziano, da parte dei partecipanti, i tentativi di risoluzione del DLP mediante la ricerca della chiave privata. Se la curva è stata scelta correttamente, il tempo necessario per individuare la chiave privata sarà equivalente al tempo desiderato.

L'intero sistema può essere semplificato in questi tre passi:

- Ogni partecipante può selezionare un numero casuale e pubblicarlo su una bacheca immutabile, come la blockchain Ethereum.
- Viene creato uno Smart Contract di Ethereum contenente la concatenazione di tutti i numeri pubblicati nello step precedente. Il valore finale viene usato come input in H_e, e viene ottenuto un punto in SFEC. Questo punto è la chiave pubblica che verrà utilizzata per cifrare il messaggio segreto.
- Chiunque, sulla Blockchain, può provare a risolvere il DLP sulla SFEC scelta e pubblicare la corrispondente chiave privata sullo Smart Contract di Ethereum. Grazie alla risoluzione, gli altri possono decifrare il messaggio.

Fino allo step 2, nessuno conosce l'input esatto della funzione di hash perché tutti i partecipanti possono ancora modificare la loro scelta. Tuttavia, questo è solo una proposta teorica del protocollo. La mia implementazione si baserà su queste caratteristiche, ma rielaborerà alcuni punti per adattarlo ad uno sviluppo pratico.

In [12] è stata effettuata una valutazione della durezza di alcune SFEC scelte in base ai bit del primo del campo finito. I bit testati sono: 32, 36, 40, 44, 48 e 52. L'obiettivo è stimare il tempo di rottura di ogni curva, attuando l'attacco di Pollard Rho e prendendo nota dei risultati classificandoli in base al tempo minimo, medio e massimo. I risultati hanno mostrato che, aumentando il numero di bit, aumentano i secondi necessari per scoprire la chiave privata. Tuttavia, a causa della randomicità dell'algoritmo di Pollard Rho, la differenza tra tempo minimo e massimo è molto.
Per risolvere questo problema viene suggerito di convertire un singolo puzzle con bit elevati, in molti puzzle con bit inferiori.

Per esempio, immaginiamo di costruire un time-lock puzzle su una curva ellittica a 52 bit. Siccome la differenza tra i tempi attesi di rottura è molto alta, potremmo costruire e concatenare quattro puzzle identici su una curva ellittica a 44 bit. Per concatenazione si intende utilizzare la soluzione di un puzzle come input del successivo.
L'operazione di concatenazione non complica la procedura di generazione della chiave, pubblica ma necessita solo l'applicazione della funzione di hash alla soluzione del puzzle precedente, con la stessa H_e.

Sono stati fatti ulteriori esperimenti prendendo in considerazione l'uso di più puzzle. Il risultato ottenuto ha mostrato che, all'aumentare del numero di puzzle (con lunghezza in bit sempre più piccola), si riduce il tempo di rottura atteso, ma si riduce, fino a quasi annullarsi, la differenza tra minimo e massimo (aumentando l'accuratezza sul tempo di compromissione atteso).

In [12] vengono sottolineate delle sfide ancora da risolvere: "in un sistema decentralizzato, chi risolve il puzzle per gli altri?. Come settare una corretta difficoltà del puzzle?"
Per ovviare a questi problemi, si potrebbe lavorare in un ambiente basato su Blockchain

con un proof of work modificato. Dovremmo cambiare lo schema di base da “trovare un nonce, tale che l’hash del blocco sia minore di un target” a “trovare una chiave privata appropriata associata a una chiave pubblica su SFEC”. Quindi, il sistema decentralizzato proposto di time-lock encryption potrebbe essere combinato con la blockchain per beneficiare di proprietà come immutabilità e trasparenza. Grazie agli incentivi previsti in una struttura a blockchain, tutti i miner sarebbero propensi a computare la chiave privata perché riceveranno un compenso.

Capitolo 6

Implementazione proposta

6.1 Introduzione

Nel capitolo precedente è stata analizzata la proposta di Lai et al. dal punto di vista prettamente teorico. Tuttavia, è mio preciso obiettivo fornire un'implementazione del protocollo sopra descritto. Alcuni concetti verranno riadattati e ampliati, e verrano applicati nuovi elementi per rendere il sistema progettato perfettamente applicabile e funzionante.

Prima di tutto dobbiamo valutare quelli che sono gli elementi alla base dell'implementazione proposta:

- La funzione H_e verrà descritta nel dettaglio nel paragrafo successivo e verrà incorporata nello Smart Contract.
- Lo Smart Contract verrà implementato nel linguaggio Solidity in ambiente Ethereum.
- L'algoritmo di Pollard's Rho verrà sviluppato tramite il linguaggio di programmazione Java seguendo le indicazioni dell'articolo [16].

Tutti e tre sono degli aspetti chiave di questo lavoro. La funzione H_e è fondamentale per mappare un valore in un punto della curva, lo Smart Contract è l'unico elemento che permette la condivisione di informazioni tra i partecipanti e la risoluzione del problema computazionale, mentre l'algoritmo di Pollard's Rho rimane il principale e più efficiente metodo di valutazione della vulnerabilità di una curva ellittica.

Il mio percorso di sviluppo di questa implentazione è partito dallo studio dell'articolo di Lai ([12]), analizzando il protocollo di base proposto (già descritto nel capitolo precedente) e proponendo delle soluzioni che potessero risolvere le mancanze e debolezze riscontrate. L'obiettivo finale è riuscire a realizzare un sistema completo che possa adattarsi a tutte le esigenze e che si avvicini a garantire il principio di time-lock encryption: cifrare un messaggio in modo che possa essere decifrato in un istante di tempo nel futuro.

6.2 Funzione di Icart

La definizione originale della funzione H_e prevedeva come dominio una stringa di lunghezza arbitraria e come codominio un punto sulla curva prescelta. Per implementare questa funzione, ho deciso di restringere il dominio, prevedendo come valore di ingresso un elemento appartenente al campo finito F_p, su cui è costruita la curva, applicando le regole alla base della funzione definita da Icart in [14, 15].

Consideriamo la curva con equazione di Weyerstrass $y^2 = x^3 + ax + b$ su un campo F_{p^n} dove $p > 3$ e $p^n = 2 \; mod \; 3$.

La funzione di Icart è definita come segue:

$$f_{a,b} : F_{p^n} \rightarrow E_{a,b}$$
$$u \rightarrow (x, y)$$

dove

$$x = \left(v^2 - b - \frac{u^6}{27}\right)^{\frac{1}{3}} + \frac{u^2}{3}$$
$$y = u * x + v$$

con

$$v = \frac{3 * a - u^4}{6 * u}$$

In altre parole, questa funzione accetta come input un elemento del campo finito F_{p^n} (corrispondente a u), e restituisce un punto sulla curva $E_{a,b}$ definito sullo stesso campo finito.

Per la stringa $u = 0$, la funzione $f_{a,b}(0) = O$, ovvero il punto all'infinito. Ogni operazione di frazione (per il calcolo di x, y e v) è un prodotto del numeratore per l'inverso modulo q (p^n) del denominatore.

L'unico problema della funzione di Icart è che possono esserci alcuni punti della curva che non vengono mappati. Ovvero, non esistono degli elementi del dominio che vengono mappati ad alcuni punti del codominio.

6.2.1 Esempio.

Prendiamo un esempio citato nell'articolo [15].
Sia $q = 11$, $a = -1$, $b = 0$ e $\#E_{-1,0} = 12$.
Applicando la funzione di Icart su tutti gli elementi appartenenti al campo F_{11} otteniamo che alcuni punti appartenenti alla curva non vengono mappati: $(0, 0), (1, 0), (10, 0)$. Circa $\frac{1}{4}$ del totale.
Per i nostri scopi, questo non rappresenta un problema. Questa funzione, come vedremo in seguito, è stata implementata all'interno dello Smart Contract.

6.3 Smart Contract

Nel protocollo originale viene proposta la creazione di uno Smart Contract che potesse contenere tutti i valori che i partecipanti hanno inserito, per poi concatenarli. La mia proposta è leggermente diversa. Ho deciso di realizzare uno Smart contract in Solidity che, al momento della sua creazione, permette di decidere quanti partecipanti inseriranno il loro valore. Una volta che tutti i membri avranno aggiunto il proprio numero casuale, tutti i componenti verranno sommati insieme e inseriti come input $(mod\ p)$ nella funzione di Icart per ottenere un punto appartenente alla curva prescelta, corrispondente alla chiave pubblica Q. A questo punto, come previsto dal puzzle, tutti i partecipanti concorreranno per scoprire la corrispondente chiave privata e ottenere il reward promesso.

Di seguito le funzioni principali in Solidity dello Smart Contract:

```
function icart_function(uint256 _seed) private returns (uint256, uint256){
            uint256 exp = ((((P * 2) - 1)) * (mod_inv(3, P)) % P);
            int256 v = (int(int(A * 3) - int(_seed ** 4)) * int(mod_inv(6 *_seed, P)));
            uint256 x = ( uint(v * v) - B - ( (_seed ** 6) * mod_inv(27, P) ) );
            x = right_exp_mod(x, exp, P);
            x = ( x + ( (_seed ** 2) * mod_inv(3, P) ))  % P;
            uint256 y = uint((((int(_seed * x) + v) % int(P)) + int(P)) % int(P));
            return (x, y);
        }
```

Il presente codice rappresenta la funzione di Icart che prende in input un certo valore appartenente a F_p e restituisce un punto della curva $E_{a,b}$ prescelta.

```
function insert_new_seed(uint256 _seed) public{
            if (processCompleted == false){
                  seeds[counter] = _seed;
                partecipants[counter] = msg.sender;
                counter++;
                if (counter == nPartecipants){
                    processCompleted = true;
                }
            }
}
```

Il presente codice rappresenta la funzione che accetta come input un valore scelto da un partecipante e lo aggiunge alla somma di tutti i valori inseriti fino a quel momento.

```
function get_point_on_curve() private returns (uint256, uint256){
            if (processCompleted == true)
            {
               (pKeyX, pKeyY) = icart_function(sum_seeds() % P);
               return (pKeyX, pKeyY);
            }
            else
               return (0,0);
         }
```

Il presente codice rappresenta la funzione che viene chiamata da ogni partecipante quando vuole ottenere il punto sulla curva.

```
function transfer_reward(uint256 _amount, address payable _winner)  private{
            _winner.transfer(_amount);
            emit log(_winner.balance);
}
```

Il presente codice rappresenta la funzione che fornisce il reward al partecipante che ha trovato la chiave privata corretta. Questa funzione privata viene richiamata quando il partecipante invia al contratto la chiave privata candidata che lui ha trovato.

```
function insert_candidate_private_key(uint256 _cPKey) public returns (bool){
      if ((verifiedPrivateKey == false) && (processCompleted == true))
     {
         (uint256 qX, uint256 qY) = EllipticCurve.ecMul(
         _cPKey,
         GX,
         GY,
         A,
         P
        );
        if ((qX == pKeyX) && (qY == pKeyY)){
          sKey = _cPKey;
          verifiedPrivateKey = true;
          puzzleWinner = msg.sender;
          transfer_reward (REWARD, msg.sender);
       }
     }
     return verifiedPrivateKey;
  }
```

Il presente codice rappresenta la funzione che prende in input la possibile chiave privata corrispondente alla chiave pubblica e, se corretta, restiuisce il valore true e salva l'indirizzo del partecipante "vincitore".

Quindi, sinteticamente lo smart contract implementato funziona nel seguente modo:

- Il contratto viene creato e viene deciso il numero di partecipanti coinvolti nel puzzle.
- Ogni partecipante richiama una funzione dello Smart Contract inviando un valore casuale scelto da lui.
- Una volta terminato l'inserimento, tutti i valori vengono sommati.
- Quando un partecipante richiede la chiave pubblica della curva ellittica, chiama la funzione specifica che applica la funzione di Icart alla somma dei componenti ottenendo un punto sulla curva.
- Da questo momento in poi, tutti i partecipanti concorrono per scoprire la chiave privata corripondente. Una volta trovata, il partecipante richiama la funzione specifica inviando la chiave privata candidata. Se corretta, riceverà un certo quantitativo di reward, se sbagliata potrà ripetere il tentativo nuovamente.

Per quanto riguarda il consumo di Gas che ogni partecipante effettua per eseguire le funzioni dello Smart Contract, questo è un piccolo esempio con una curva piccola prestabilita:

$$P = 659 \quad A = 416 \quad B = 569 \quad G = (23, 213) \quad |< G >| = 673$$

Numero di partecipanti = 2.

Deployment contract:

- Transaction cost: 1285240 Gas.

Inserimento valore "43256562323" del partecipante 1:

- Transaction cost: 89548 Gas.

Inserimento valore "736562223" del partecipante 2:

- Transaction cost: 95351 Gas.

Ottenimento della chiave pubblica $Q = (267, 164)$:

- Transaction cost: 175725 Gas.

Verifica chiave privata $K = 364$:

- Transaction cost: 95837 Gas.

Quindi è facilmente intuibile che un partecipante al puzzle che riesce a individuare la chiave privata al primo tentativo, consuma circa 360000 Gas, che equivale a 360000 Gwei oppure a 0,00036 eth.

Di conseguenza, il reward previsto per aver trovato con successo la chiave privata deve essere maggiore del costo richiesto per partecipare, altrimenti non ci sarebbe alcun incentivo perché i partecipanti debbano comportarsi correttamente.

6.4 Pollard Rho attack

L'ultimo tassello della nostra implementazione riguarda lo sviluppo di un metodo per stimare il tempo di rottura della curva prescelta, ovvero di verificarne la debolezza. Trovando una curva con un tempo di rottura adeguato potremmo cifrare un messaggio con la sicurezza che non venga decifrato prima del dovuto. In [12] viene utilizzato l'attacco di Pollard Rho per valutare in quanto tempo la chiave privata della curva viene scoperta. In [16] viene illustrato il funzionamento dell'algoritmo evidenziandone i passaggi chiave.
L'obiettivo di questo metodo è trovare la chiave privata n tale che $n * G = Q$, dove G è il punto generatore e Q è la chiave pubblica della curva.
Innanzitutto, bisogna creare tre insiemi S_1, S_2, S_3 e assegnare ogni punto sulla curve a uno di questi sulla base del valore della coordinata x.

L'assegnamento è il seguente, dove oG è l'ordine del punto generatore G della curva:

- S_1 : con $0 \leq x \leq \frac{oG}{2}$
- S_2 : con $\frac{oG}{2} < x \leq 7 * \frac{oG}{8}$
- S_3 : con $7 * \frac{oG}{8} < x \leq oG$

Calcoliamo un primo punto $R_0 = a_0 * G + b_0 * Q$, dove a_0 e b_0 sono due interi casuali scelti in $[1, oG - 1]$. Il metodo prosegue chiamando iterativamente una funzione f che assegna al successivo punto R_{i+1} un differente valore in base all'input.

$$R_{i+1} = f(R_i) = \begin{cases} G + R_i & se\ R_i \in S_1 \\ 2 * R_i & se\ R_i \in S_2 \\ Q + R_i & se\ R_i \in S_3 \end{cases}$$

per ogni $i \geq 0$.

Quindi, in base all'appartenenza ad un gruppo del punto R_i possiamo assegnare un certo valore al punto R_{i+1}. Nel caso in cui $R_i = O$ allora $R_i \in S_1$.
Anche la sequenza di a_i e b_i cambia il suo valore in base all'appartenenza di R_i ad un determinato gruppo.

$$a_{i+1} = \begin{cases} a_i + 1\ mod\ oG & se\ R_i \in S_1 \\ 2 * a_i\ mod\ oG & se\ R_i \in S_2 \\ a_i\ mod\ oG & se\ R_i \in S_3 \end{cases}$$

e

$$b_{i+1} = \begin{cases} b_i\ mod\ oG & se\ R_i \in S_1 \\ 2 * b_i\ mod\ oG & se\ R_i \in S_2 \\ b_{i+1}\ mod\ oG & se\ R_i \in S_3 \end{cases}$$

La funzione f, con input R_i, verrà richiamata più volte finchè non occorre una collisione tra due punti:

$$R_i = a_i * G + b_i * Q = a_{2i} * G + b_{2i} * G = R_{2i}$$

Infatti, riscrivendo l'equazione diventa:

$$(b_i - b_{2i}) * Q \equiv (a_{2i} - a_i) * G$$

E per i logaritmi discreti diventa:

$$n = log_G(Q) \equiv (b_i - b_{2i})^{-1} * (a_{2i} - a_i)\ (mod\ oG)$$

Di seguito il codice scritto in Java dell'algoritmo Pollard Rho attack.

```
public void pollardMethod(){
      BigInteger a0;
      BigInteger b0;
      Random random = new Random();
      do{
         a0 = new BigInteger(this.oG.bitLength(), random);
      }while ((a0.compareTo(this.oG) >= 0) || (a0.compareTo(BigInteger.ZERO) == 0));
      do{
         b0 = new BigInteger(this.oG.bitLength(), random);
      }while ((b0.compareTo(this.oG) >= 0) || (b0.compareTo(BigInteger.ZERO) == 0));
      Point r1 = EllipticCurve.multiplication(this.g, a0, this.a, this.p);
      Point r2 = EllipticCurve.multiplication(this.q, b0, this.a, this.p);
      Point r0 = EllipticCurve.addition(r1, r2, this.a, this.p);
      BigInteger i = BigInteger.ONE;
      BigInteger pKey = BigInteger.ZERO;
      R.put(i, r0);  aMap.put(i, a0); bMap.put(i, b0);
      do{
         f(i);
         f(f(i.multiply(new BigInteger("2"))));
         i = i.add(BigInteger.ONE);
      }while ((R.get(i).getX().compareTo(R.get(i.multiply(new BigInteger("2"))).getX()) != 0)
|| (R.get(i).getY().compareTo(R.get(i.multiply(new BigInteger("2"))).getY()) != 0));
}
```

```
private BigInteger f(BigInteger i){
      Point p1;
      if (R.get(i).getX().compareTo(s1) <= 0){
         p1 = EllipticCurve.addition(g, R.get(i), a, this.p);
         R.put(i.add(BigInteger.ONE), p1);
         aMap.put(i.add(BigInteger.ONE), aMap.get(i).add(BigInteger.ONE).mod(oG));
         bMap.put(i.add(BigInteger.ONE), bMap.get(i));
      }
      else
      if ((R.get(i).getX().compareTo(s1) > 0) && (R.get(i).getX().compareTo(s2) <= 0)){
         p1 = EllipticCurve.multiplication(R.get(i), new BigInteger("2"), a, this.p);
         R.put(i.add(BigInteger.ONE), p1);
         aMap.put(i.add(BigInteger.ONE), aMap.get(i).multiply(new BigInteger("2")).mod(oG));
         bMap.put(i.add(BigInteger.ONE), bMap.get(i).multiply(new BigInteger("2")).mod(oG));
      }
      else
      if ((R.get(i).getX().compareTo(s2) > 0) && (R.get(i).getX().compareTo(s3) <= 0)){
         p1 = EllipticCurve.addition(q, R.get(i), a, this.p);
         R.put(i.add(BigInteger.ONE), p1);
         aMap.put(i.add(BigInteger.ONE), aMap.get(i).mod(oG));
         bMap.put(i.add(BigInteger.ONE), bMap.get(i).add(BigInteger.ONE).mod(oG));
      }

      return i.add(BigInteger.ONE);
   }
```

Capitolo 7

Analisi dei risultati ottenuti

7.1 Introduzione

In questo capitolo, dopo aver realizzato l'implementazione dell'algoritmo di Pollard's Rho, è mio obiettivo effettuare, su delle curve da me scelte, una valutazione sui tempi di risoluzione e scoperta della chiave privata. Nella tabella che pubblicherò, riporterò tre differenti tempi, calcolati eseguendo più volte l'algoritmo: minimo, massimo e medio. Nell'articolo [12] è stata fatta un'analisi simile, ma con curve molto più grandi. Dai risultati che scaturiranno in entrambe le prove, proporrò una nuova implementazione dello Smart Contract per risolvere gli evidenti problemi di Pollard.
Prima di procedere è importante fare una premessa sulle curve che utilizzerò. A causa della difficoltà di reperimento di curve "piccole" (dell'ordine di 32/40 bit), ne userò alcune che potrebbero essere definite "giocattoli", ma comunque soddisfacenti per i nostri scopi. Inoltre, a causa della scarsa potenza dei mezzi a mia disposizione, curve troppo grandi non mi permetterebbero un'analisi rapida ed efficace.
Le caratteristiche del computer utilizzato sono le seguenti:

- CPU: Intel Core i7 – 6500U, up to 3.1 GHz
- Ram: 12GB
- OS: Win10

Le curve analizzate sono da 5/10/13/14 bit. Sono molto piccole, ma utili per mostrare alcuni evidenti difetti dell'attacco di Pollard's Rho.

7.2 Analisi implementazione Pollard's Rho attack

bits of prime	32	36	40	44	48	52
Minimum	0.12	0.71	1.52	3.14	32.46	67.55
Average	3.08	10.62	47.21	146.36	684.45	2727.81
Maximum	8.56	39.64	161.42	506.27	2116.94	8918.28

Figura 7 - Classificazione del tempo di rottura delle curve con attacco di Pollard Rho tratta da [12]

Tabella 1 - Tempi (in millisecondi) di rottua delle curve con l'attacco di Pllard's Rho

Bit	P	A	B	G	Q	O	K	T. min	T. max	T. med
5	23	1	4	(0, 2)	(17, 14)	29	17	4	16	10
10	659	416	569	(23, 213)	(150, 25)	673	10	12	78	45
13	7919	1001	75	(4023, 6036)	(4135, 3169)	7889	4334	125	282	203
14	15017	3823	2424	(517, 12902)	(12340, 14339)	14887	12769	125	359	242

La tabella 1 rappresenta i tempi (in millisecondi) necessari alla rottura delle curve, da me scelte, attuando l'attacco di Pollard's Rho. Come viene presentato nella figura 7 ([12]), anche in questo caso, seppur più grosse, il divario tra il tempo minimo e massimo è sempre elevato. A causa della casualità insita nell'algoritmo, dovuta alla generazione dei valori a_0 e b_0 e all'appartenenza dei punti R_i ai diversi insiemi (e al raggruppamento degli insiemi), diventa difficile scegliere una curva con un tempo di rottura affidabile e facilmente prevedibile. Più la curva è grande, più la differenza aumenta e più difficile è stabilire il tempo correto. In [12] viene offerta una soluzione che si basa sull'utilizzo di più puzzle, concatenati uno dentro l'altro, composti da curve ellittiche con meno bit. Per concatenazione, si intende l'inserimento della soluzione di un puzzle (la chiave privata) come input della funzione H_e del puzzle successivo. Chiaramente, una simile modifica può comportare (a meno di specifici controlli) l'eliminazione del contributo casuale dei partecipanti, in favore di un input della funzione H_e (per il primo puzzle) fissato a priori dal creatore dello Smart Contract.

Tabella 2 - Tempi (in millisecondi) di rottura delle curve con l'attacco di Pllard's Rho

Bit	P	A	B	G	O	K	T. min	T. max	T. med
5 x 256	23	1	4	(0, 2)	29	17	201	293	232
10 x 128	659	416	569	(23, 213)	673	10	850	1440	1145
13 x 8	7919	1001	75	(4023, 6036)	7889	4334	673	976	824
14 x 4	15017	3823	2424	(517, 12902)	14887	12769	1048	1656	1352

Nella tabella 2, per ogni curva della tabella 1, ho applicato Pollard's Rho con più puzzle. Il risultato abbastanza evidente è che, concatenando più puzzle costruiti su una stessa

curva, la differenza tra tempo massimo e minimo di rottura è molto inferiore dell'usare un singolo puzzle. Questo aspetto può essere sfruttato per scegliere la curva più adatta.

7.2.1 Esempio

Facendo un rapido esempio, supponiamo di voler cifrare un messaggio M per circa 240 millisecondi. Se utilizzassimo la curva a 14 bit, avremmo un tempo medio di circa 242 millisecondi, ma con un minimo di 125 e un massimo di 359. E' facile capire che questo puzzle non è adeguato perché c'è troppa differenza tra i due diversi tempi. Per risolvere il problema, potremmo usare una curva più piccola di 5 bit, con 256 puzzle. Dalla tabella 2, vediamo che il tempo medio è 232, il massimo 293 e il minimo 201. Di conseguenza, nonostante queste siano curve molto piccole e difficilmente applicabili, utilizzare 256 puzzle concatenati con curve a 5 bit permette di prevedere il tempo di rottura con più affidabilità.
Chiaramente per effettuare questi test, ho fatto una leggera modifica all'algoritmo di Pollard's Rho per adattarlo a risolvere più puzzle, ma senza intaccarne il funzionamento

7.3 Modifica Smart Contract

Come anticipato nel paragrafo precedente, il fatto di utilizzare più puzzle concatenati richiede una modifica dello Smart Contract implementato nel Capitolo 5.
Lo Smart Contract originale prevedeva che tutti i valori di ogni partecipante venissero sommati e inseriti all'interno della funzione H_e (funzione di Icart). Questo significa introdurre un certo grado di casualità nell'algoritmo. Tuttavia, con l'utilizzo di più puzzle, il risultato di un problema è l'input del successivo. Per evitare di fare controlli specifici per il primo puzzle, ho deciso di dare la possibilità allo sviluppatore dello Smart Contract di decidere l'input fisso della prima funzione H_e. Quindi, supponendo quattro puzzle concatenati, abbiamo un seme iniziale fissato nel contratto che verra mappatto in un punto sulla curva corrispondente alla chiave pubblica del primo problema. Da questo punto in poi, i partecipanti concorrono per scoprire la chiave privata e utilizzarla come input per il puzzle successivo. Stessa cosa per il secondo, terzo e quarto problema. Fino alla chiave segreta definitiva che permetterà l'assegnamento del reward a chi l'ha fornita.

7.4 Riflessioni

In questa implementazione e, in generale, nella creazione di un puzzle basato su curve ellittiche, bisogna tenere conto delle capacità computazionali di tutti i partecipanti. Come già detto in precedenza, per scegliere una curva con un tempo di rottura desiderato utilizziamo l'algoritmo di Pollard's Rho. Tuttavia, questo attacco dipende molto anche dal potere computazionale del computer che lo esegue: usando il mio computer una curva potrebbe avere un tempo di rottura superiore ad una stessa curva con un computer con capacità maggiori. Per il mittente può essere un problema perché, nella scelta della curva giusta, deve tenere anche in considerazione delle capacità computazionali degli avversari. In un ambiente dove tutti i partecipanti sono conosciuti, questo non è un grosso problema, ma in un ambiente dove i partecipanti possono essere chiunque può diventare diffcile da prevedere. Una possibile soluzione potrebbe essere cercare di fare due cose: obbligare tutti i partecipanti ad attendere il tempo richiesto e parallelamente obbligarli a trovare la chiave privata. In che modo fare questo? Simulando il funzionamento del Proof-of-Work della blockchain. Nel capitolo successivo farò un'analisi di questa variazione.

C’è un ulteriore problema da prendere in considerazione: il reward. Ogni partecipante consuma un certo numero di Gas richiamando le funzioni dello Smart Contract e questo si trasforma in un consumo di Ether, ovvero di denaro. Siccome è nostro interesse che un individuo venga coinvolto nel puzzle, dobbiamo prevedere un incentivo che sia maggiore del costo richiesto per partecipare. Questo può essere un piccolo problema perché richiede di conoscere all’incirca i Gas consumati per richiamare ogni funzione del contratto. Per curve molto piccole è facile individuarli, per curve molto grande può diventare abbastanza operoso. L’unica soluzione al momento è offrire un reward fisso di due Ether (lo stesso reward previsto per la costruzione di un blocco in Ethereum).

Capitolo 8

Conclusioni

8.1 Riepilogo

In questo capitolo conclusivo farò un riepilogo di tutto il lavoro svolto in questa tesi. Il mio obiettivo era effettuare un'analisi dei sistemi sviluppati negli anni che garantiscono la time-lock encryption, valutare le loro caratteristiche e capire quelli che sono i più promettenti. Ho fatto una classificazione delle tecniche in base a due parametri: puzzles e agents. Dopo l'analisi di tutti i metodi ho scelto quello basato su curve ellittiche che può essere considerato il migliore (al netto di evidenti problemi già descritti) e fornito un'implementazione completa e adattabile ad ogni tipo di curva presa in considerazione. Per quanto riguarda l'implementazione, ho creato uno Smart Contract rappresentante il problema computazione. In particolare ho sviluppato al suo interno la funzione di Icart per mappare un valore in un punto sulla curva, e poi ho creato tutto il sistema di verifica della chiave privata e di reward in caso il puzzle venga risolto.
Inoltre, ho implementato il metodo di Pollard's Rho in Java per la valutazione della debolezza della curva ellitta prescelta e ho effettuato un'analisi comparativa dei tempi di rottura di curve piccole scelte da me. Come già illustrato in [12], l'algoritmo di Pollard's Rho ha evidenti difetti tra i tempi minimi e massimi di rottura. Per ovviare a questo, ho sviluppato una modifica allo Smart Contract per prevedere l'uso di più puzzle concatenati, ovvero l'uno dentro l'altro.
Riepilogando, tra tutte le modalità analizzate, quella basata che utilizza le curve ellittiche è sicuramente la più promettente. Permette di cifrare un messaggio e di renderlo decifrabile dopo il tempo prestabilito. Il quantitativo di tempo desiderato è strettamente dipendente dalla curva scelta e questo può risultare un problema. La soluzione proposta è l'algoritmo di Pollard's Rho che riesce a stimare il tempo di rottura di una curva ellittica.

8.2 Sviluppi futuri

Questo sistema è, tuttavia, a sua volta dipendente dalle capacità computazionali dei computer che lo eseguono. Questo comporta che il mittente del messaggio deve riuscire a fare una previsione delle risorse degli avversari, e al momento resta un tema aperto su cui cercare di trovare una soluzione. Nel caso di un sistema chiuso con partepanti "controllati", il problema non sussiste, ma in un sistema aperto bisogna fare qualche considerazione. Ho fornito un piccolo spunto da cui partire per affrontare questa incombenza: utilizzare il principio alla base del Proof-of-Work dei sistemi su Blockchain, obbligando i partecipanti ad attendere il tempo richiesto per la decifratura. In questo modo, oltre a fornire la chiave privata candidata, il partecipante dovrà trovare un certo numero di nonce (partendo da un seme fornito dallo sviluppatore dello Smart Contract), tale che l'hash calcolato sull'attuale nonce, concatenato a quello precedente, sia minore di un target fissato (target scelto per rispettare il tempo stabilito).
La mia proposta cambia l'uso delle curve in questo sistema. Come detto, l'idea è sfruttare il Proof-of-Work usando come base un seme presente nello Smart Contract. A questo

punto, si obbliga ogni partecipante a trovare un certo numero di nonce in base al tempo di decifratura desiderato. Una volta che un partecipante ha trovato tutti i valori richiesti, li invia allo Smart Contract e, nel caso fossero corretti, riceverà un reward adeguato. La chiave pubblica verrà ottenuta applicando la funzione di Icart all'ultimo nonce e verrà usata per scoprire la relativa chiave privata per decifrare il messaggio finale segreto. Quindi, il tempo richiesto per tutto il processo di time-lock encryption corrisponde a:

$$T_{dec} = T_n + T_c$$

dove T_n è il tempo richiesto per ottenere tutti i nonce, mentre T_c è il tempo di rottura della curva prescelta.

In questa proposta, questi due valori (T_n e T_c) possono essere calibrati. Si potrebbe rendere T_n maggiore (quindi con un Proof-of-Work più complesso) e un T_c minore (usando una curva molto piccola). Oppure il contrario, in base alle esigenze del mittente del messaggio segreto.
Ci sono due principali problemi in questa soluzione:

- Il primo problema, riguarda il fatto che la scelta della curva giusta diventa secondaria. Tutto il protocollo di time-lock encryption diventerebbe dipendente, non solo dalla curva prescelta, ma soprattutto dalla scoperta dei nonce. In altri termini, lo sviluppatore dello Smart Contract potrebbe scegliere una curva con tempo di rottura bassissimo e obbligare tutti i partecipanti ad attendere la deadline desiderata sfruttando solo il Proof-of-Work. In questo modo, la curva prescelta viene messa in secondo piano e tutto il protocollo originale verrebbe stravolto.
- Il secondo problema, riguarda il fatto che un partecipante potrebbe sentirsi scoraggiato nel dover trovare tutti valori richiesti. Questo perché non tutti possiedono capacità computazionali cosi elevate. Una possibile soluzione è quella di slegare i nonce dalla curva, ovvero un partecipante più debole potrebbe aspettare che qualcun altro risolva il Proof-of-Work (ricevendo in cambio un reward) e, quando la chiave pubblica viene resa disponibile, mettersi in gioco cercando di rompere la curva, ottenendo, in caso di esito positivo, un altro reward. Quindi, ricapitolando, i reward possibili sarebbero due: uno per i nonce e uno per la curva. Cosi da accontentare i partecipanti più "forti" e i partecipanti più "deboli".

Al momento si tratta solo di una proposta, senza ancora nessuna implementazione, ma con i giusti accorgimenti e modifiche potrebbe rappresentare un possibile sviluppo di questo promettente sistema.

8.3 Confronto finale

Il seguente schema riepiloga tutte le tecniche analizzate con relativi vantaggi e svantaggi:

- Secret Sharing
  - E' una tecnica che non prevede la gestione del tempo.
  - Base di tutte le tecniche successive.
  - Non fornisce alcun incentivo per cui i partecipanti si comportino correttamente.
  - Non adatto a garantire il time-lock encryption.

- Key Sharing
    - I partecipanti gestiscono il tempo.
    - Non fornisce alcun incentivo per cui i partecipanti si comportino correttamente.
    - Se un partecipante non contribuisce, le informazioni si perdono.
    - Non adatto a garantire il time-lock encryption.
- Time-Lapse Cryptography
    - Un gruppo di manager gestisce il tempo.
    - Non fornisce alcun incentivo per cui i partecipanti si comportino correttamente.
    - I partecipanti che non si comportano correttamente vengono subito scoperti e isolati.
    - La maggioranza dei manager deve essere fidata.
    - Non adatto a garantire il time-lock encryption.
- Conditional Oblivious Transfer
    - Il tempo è gestito da un Time Server.
    - Gli unici partecipanti sono il mittente, il server e il destinatario.
    - Il messaggio, il tempo corrente e il release time rimangono nascosti al server e destinatario.
    - Adatto a garantire il time-lock encryption a patto che il Time Server sia di fiducia.
- RC5 con chiave modulabile
    - Puzzle semplice con chiavi troppo corte.
    - Non adatto a garantire il time-lock encryption.
- Quadrati ripetuti
    - Puzzle semplice.
    - Non parallelizzabile.
    - Non adatto a garantire il time-lock encryption.
- Witness encryption
    - Tempo calcolato in base alla lunghezza della Blockchain.
    - I partecipanti sono incentivati a comportarsi correttamente.
    - Non richiede una computazione continua.
    - L'unica implementazione esistente richiede di conoscere il witness in cifratura e in decifratura.
    - Adatto a garantire il time-lock encryption, a patto che venga sviluppata un'implementazione in cui il witness venga utilizzato solo in decifratura.
- Curve ellittiche con ElGamal
    - Tempo calcolato in base alla complessità della curve ellittica scelta.
    - I partecipanti sono incentivati a comportarsi correttamente.
    - Richiede una computazione continua.
    - Adatto a garantire il time-lock encryption, a patto che venga scelta la curva ellittica più adatta al tempo di risoluzione desiderato.

Bibliografia

[1] R. L. Rivest, A. Shamir, D. A. Wagner. Time-lock puzzles and timed-release Crypto. March 1996.

[2] Shamir, Adi. How to share a secret. Communications of the ACM 22.11 (1979): 612-613.

[3] M. O. Rabin, C. Thorpe. Time-Lapse Cryptography. Harward University, Cambridge, December 2006.

[4] Di Crescenzo G., Ostrovsky R., Rajagopalan S. (1999) Conditional Oblivious Transfer and Timed-Release Encryption. In: Stern J. (eds) Advances in Cryptology — EUROCRYPT '99. EUROCRYPT 1999. Lecture Notes in Computer Science, vol 1592. Springer, Berlin, Heidelberg. May 1999.

[5] T. Elgamal, A public key cryptosystem and a signature scheme based on discrete logarithms, in *IEEE Transactions on Information Theory*, vol. 31, no. 4, pp. 469-472, July 1985.

[6] Rabin, Michael O. "How To Exchange Secrets with Oblivious Transfer." IACR Cryptol. ePrint Arch. 2005.187 (2005).

[7] De Santis, Alfredo, and Giuseppe Persiano. "Public-randomness in public-key cryptography." Workshop on the Theory and Application of of Cryptographic Techniques. Springer, Berlin, Heidelberg, 1990.

[8] Liu, Jia, et al. "How to build time-lock encryption." Designs, Codes and Cryptography 86.11 (2018): 2549-2586.

[9] Nakamoto, Satoshi, and A. Bitcoin. "A peer-to-peer electronic cash system." Bitcoin.–URL: https://bitcoin. org/bitcoin. pdf 4 (2008).

[10] Garg, S., Gentry, C., Sahai, A., & Waters, B. (2013, June). Witness encryption and its applications. In Proceedings of the forty-fifth annual ACM symposium on Theory of computing (pp. 467-476).

[11] W. Smith, M. Arapinis. Towards Timelock Encryption. 4th Year Project Report, The University of Edinburgh. April 2019.

[12] Lai, W. J., Hsueh, C. W., & Wu, J. L. (2019, July). A Fully Decentralized Time-Lock Encryption System on Blockchain. In 2019 IEEE International Conference on Blockchain (Blockchain) (pp. 302-307). IEEE.

[13] Etherem, https://ethereum.org/en/developers/docs/

[14] Icart, T. (2009, August). How to hash into elliptic curves. In Annual International Cryptology Conference (pp. 303-316). Springer, Berlin, Heidelberg.

[15] Simmons, T. (2015). A Computational Study of Icart's Function.

[16] J. Falk, P. Svensson, M. Nilsson. On Pollard's rho method for solving the elliptic curve discrete logarithm problem. Linnaeus University, Sweden. 2019.