CryptoNight & RandomX

Alessia Angelone, Francesco Baccaro, Simona Bertè,
Emilio Cassaro, Christian Coduri, Giovanni Nicosia

4 giugno 2024

Sommario

Your abstract.

Capitolo 1

RandomX

1.1 Design di Randomx

Per minimizzare il vantaggio di hardware specializzati come gli ASIC, già precedentemente discussi, un algoritmo di **Proof of Work** (**PoW**) deve potersi legare ai dispositivi esistenti il cui uso risulti essere ampiamente diffuso. Non a caso, si focalizza sull'utilizzo delle **CPU** per i seguenti motivi:

- **Accessibilità**: Le CPU, essendo meno specializzate, sono più diffuse e accessibili. Un algoritmo basato su CPU è più **egalitario** e permette a più partecipanti di unirsi alla rete, contrariamente a ciò che accadeva con gli ASIC in quanto il loro notevole costo permetteva una centralizzazione del mining nelle mani di pochi;
- **Istruzioni Hardware Comuni**: CPU diverse **condividono** un grande *subset* di istruzioni hardware native;
- **Documentazione e Compilatori**: Tutti i principali set di istruzioni CPU sono ben **documentati** con diversi compilatori **open-source** disponibili.

1.1.1 Prova di Lavoro Dinamica

L'idea alla base di una **Proof of Work** basata su CPU è l'impiego di un *lavoro dinamico* sfruttando il fatto che queste accettano come input non solo **dati**, come le tipiche funzioni di hash crittografiche, ma anche il **codice**. Ciò evita che le sequenza delle operazioni sia fissa e quindi più facilmente eseguibile da un circuito integrato specializzato. Una **Proof of Work dinamica** consiste in 4 steps:

1. Generazione di un programma casuale;
2. Traduzione nel codice macchina nativo della CPU;
3. Esecuzione del programma;
4. Trasformazione dell'output del programma in un valore crittograficamente sicuro.

Generazione di un programma casuale

Inizialmente la progettazione di una Proof of Work si basava sulla generazione di un programma in linguaggi ad alto livello, come C o Javascript, ma questi per via della loro sintassi complessa, implicavano notevoli costi in termini di tempo.

Il modo più veloce per generare un programma casuale è utilizzare un generatore senza logica, riempiendo semplicemente un **buffer con dati casuali**. Questo richiede la progettazione di un linguaggio di programmazione senza sintassi, in cui tutte le stringhe di bit casuali rappresentano programmi validi.

Traduzione del Programma in Codice Macchina

Per generare il codice macchina il più velocemente possibile, il nostro set di istruzioni deve essere il più vicino possibile all'hardware nativo, ma allo stesso tempo deve risultare abbastanza generico in modo da non limitare l'algoritmo a una specifica architettura CPU.

Esecuzione del Programma

L'esecuzione del programma dovrebbe utilizzare il **maggior numero** possibile di **componenti della CPU**. Alcune delle caratteristiche che dovrebbero essere sfruttate sono:

- Cache multi-livello (L1, L2, L3);
- Cache µop;
- Unità logica aritmetica (ALU);
- Unità a virgola mobile (FPU);
- Controller di memoria;
- Parallelismo a livello di istruzione;
- Esecuzione fuori ordine;
- Esecuzione speculativa;
- Rinominazione dei registri.

Calcolo del Risultato Finale

Blake2b è una funzione di hashing crittograficamente sicura, progettata per essere veloce nel software, soprattutto sui moderni processori a 64 bit, dove è circa tre volte più veloce di SHA-3. Proprio per questo è ideale per essere utilizzata in una **Proof of Work** basata su **CPU**.

Per quanto riguarda, invece, l'elaborazione di grandi quantità di dati in modo crittograficamente sicuro, l'**Advanced Encryption Standard** (**AES**) può fornire la massima velocità di elaborazione perché molte CPU moderne supportano l'accelerazione hardware di queste operazioni.

1.1.2 "Easy program problem"

Il problema dell'*easy program* si basa sul fatto che quando viene generato un programma casuale, si potrebbe decidere di eseguirlo solo in caso sia favorevole. Questa strategia è fattibile per due motivi principali:

- **Distribuzione del Tempo di Esecuzione**: I tempi di esecuzione dei programmi generati casualmente seguono tipicamente una *distribuzione log-normale*. Questi possono rapidamente analizzati e in caso di tempo di esecuzione superiore alla media, l'esecuzione può essere saltata e può essere generato un nuovo programma. Se quest'ultima operazione risulta essere economica, può migliorare significativamente le prestazioni.
- **Ottimizzazione delle Caratteristiche**: Un'implementazione potrebbe scegliere di ottimizzare un sottoinsieme delle caratteristiche necessarie per l'esecuzione del programma. Ad esempio, si può decidere di eliminare il supporto per alcune operazioni (come la divisione) o di implementare alcune sequenze di istruzioni in modo più efficiente. In seguito, i programmi generati verrebbero analizzati ed eseguiti solo se soddisfano i requisiti specifici dell'implementazione ottimizzata.

Queste strategie di ricerca di programmi con particolari proprietà vanno in **contrasto** con gli obiettivi di questa **Proof of Work**, quindi devono essere eliminate:

- **Soluzione**: Esecuzione di una sequenza di **N programmi casuali**, in modo che ogni programma sia generato dall'**output del precedente**. L'output del programma finale viene quindi utilizzato come risultato.
- **Principio**: Una volta eseguito il primo programma, un miner deve decidere **se impegnarsi a terminare l'intera catena** (che può includere programmi sfavorevoli) o ricominciare da capo e sprecare lo sforzo impiegato sulla catena non completata;
- **Vantaggio**: Uniformare il tempo di esecuzione per l'intera catena, poiché la deviazione relativa di una somma di tempi di esecuzione distribuiti identicamente è ridotta.

1.1.3 Tempo di verifica

Poiché lo scopo del **Proof of Work** è di essere utilizzato in una **rete peer-to-peer** senza fiducia, i partecipanti alla rete devono essere in grado di verificare rapidamente se una prova è valida o meno. Ciò pone un **limite** superiore alla **complessità dell'algoritmo** di **Proof of Work**. In particolare, abbiamo fissato l'obiettivo per RandomX di essere almeno altrettanto veloce da verificare quanto la funzione di hash CryptoNight, che mira a sostituire.

1.1.4 Memory-hardness

Oltre alle risorse computazionali pure, come **ALU** e **FPU**, le **CPU** di solito hanno accesso a una grande quantità di memoria sotto forma di DRAM. Le prestazioni del sottosistema di memoria sono tipicamente ottimizzate per adattarsi alle capacità di calcolo.

Per utilizzare la **memoria esterna** così come i controller di memoria on-chip, l'**algoritmo di Proof of Work** dovrebbe accedere a un grande **buffer** di memoria (chiamato "**Dataset**"). Il Dataset deve essere:

- **più grande** di quanto possa essere memorizzato on-chip (per richiedere memoria esterna);
- **dinamico** (per richiedere memoria scrivibile)

Idealmente, la dimensione del Dataset dovrebbe essere di almeno 4 GiB. Tuttavia, a causa dei vincoli sul tempo di verifica la dimensione utilizzata da RandomX è stata selezionata a 2080 MiB.

Bibliografia

[a]

[alf]

[ASC]

[bib]

[bit]

[egn]

[ene]

[exc]

[gnu]

[int]

[nun]

[pi]

[zet]