Hardware acceleration in RandomX algorithm of Monero: A feasibility study

門羅幣的 RandomX 演算法硬體加速可行性評估

李旻珊、廖宏儒

指導教授：黃之浩 教授

摘要

2008年，第一個運用區塊鍊的加密貨幣比特幣誕生，從此開啟加密貨幣百家爭鳴的時代。加密貨幣大多是依靠礦工使用礦機尋找隨機數（nounce）使產生的雜湊值（hash）符合加密貨幣白皮書規定來保證加密貨幣的公正性。

為了公平性，許多加密貨幣開始使用號稱在各平台運行速度都一樣快的雜湊函式（hash function）。而我們研究近年來開始熱門的 RandomX 演算法，瞭解使用 RandomX 作為 hash function 的門羅幣。門羅幣是新興加密貨幣中有效改良比特幣容易暴露用戶隱私的缺點的系統，除了基本的加密貨幣挖礦流程，也進一步學習門羅幣四大項隱私技術的特性與實作原理。最終我們嘗試加速 RandomX 的運行速度。

首先，我們在 RandomX 程式中尋找任何有可能可以進行優化的函數，或是沒有必要的多餘計算，但找到的不是已經用平行化來加速過了，就是運行時間就非常短，即使加速也沒辦法產生重大的效益。因此，我們認為從軟體上來加速是不可行的。

我們改嘗試用可程式邏輯陣列（FPGA）來加速。原先，想使用 Vivado HLS 將原本的 C++ 程式轉成 Verilog 的程式再做優化，但收集資料過後，發現 Vivado HLS 只能將特定寫法寫成的 C++ 程式轉成 Verilog 程式，因此我們覺得與其改寫 C++，還不如自己寫出 Verilog，也比較好處理後面優化的方式。

我們嘗試改寫 RandomX 常用到的函數 blake2b，希望可以透過硬體來加速重複運算的部分。但丟到 Vivado 後，電路合成（synthesis）一直沒有過，可能是接線的方式出錯，或是因為對 Verilog 不熟悉，因此我們無法得知加速後的結果。

總而言之，RandomX 演算法的開發者在軟體上已經優化很完善；而在硬體方面，在 FPGA 上將 RandomX 加速或許是可行的。

一、背景動機

2008年，第一個運用區塊鍊的加密貨幣比特幣誕生，從此開啟加密貨幣百家爭鳴的時代。加密貨幣大多是依靠礦工使用礦機尋找隨機數（nounce）使產生的雜湊值（hash）結果符合加密貨幣白皮書規定來保證加密貨幣的公正性。

而在眾多的加密貨幣中，門羅幣以特別注重隱私、防審查交易以及具有 Anti-ASIC 等種種特色吸引了我們的目光，其中對於它的 Anti-ASIC 特性最感興趣。而從 2019 年 11 月 30 日起，門羅幣開始採用 RandomX 演算法作為雜湊函式（hash function）。

我們的目標是研究 RandomX 演算法、瞭解使用 RandomX 作為 hash function 的門羅幣，最終嘗試加速 RandomX 的運行速度。

二、門羅幣

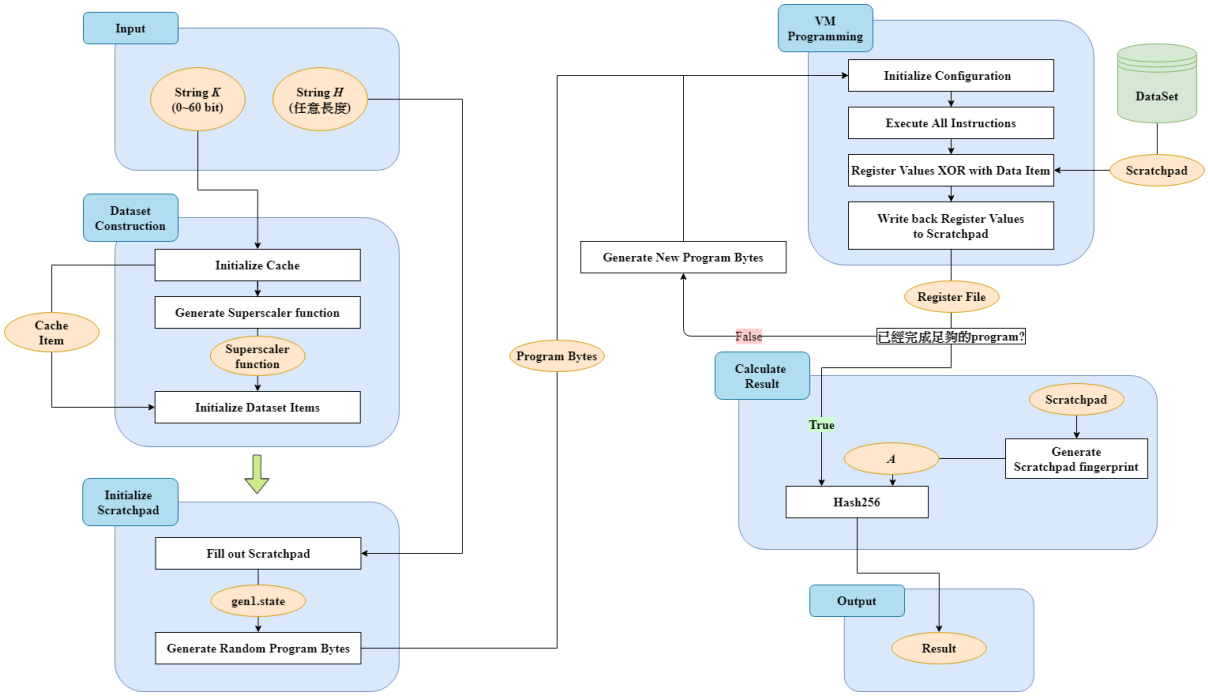
門羅幣是特別注重隱私、防審查交易的加密貨幣。當錢包設立後，錢包會從種子計算出兩組金鑰——公開金鑰（public keys）和私密金鑰（private keys）。將從公開金鑰計算出的地址交給對方後，等對方轉門羅幣到帳戶中，就能利用私密金鑰解鎖查看收到的門羅幣。

另外，門羅幣也有難度調節機制（difficulty adjustment）。門羅幣設置的理想速度是每兩分鐘就產生一個新的區塊，當速度偏快或偏慢時，系統就會調整尋找 Nonce 的難度以調節新區塊產生的速度。

門羅幣還有一項特色機制——Anti-ASIC。ASIC（application-specific integrated circuits）是指因應特殊需求客製化的積體電路。過去很多的加密貨幣發展至後期，很多大公司會刻意訂製「礦機」專門用來挖礦，因為有專門的機器和龐大記憶體容量等優勢，一旦礦機進入市場就代表著個人礦工失去競爭力，加密貨幣最後將被大公司壟斷。為了避免這樣的狀況，門羅幣自 2017 年起每六個月就會換一次挖礦的演算法，因為 ASIC 是客制化製作，沒辦法進行調整，這樣頻繁的更換演算法也就讓礦機失去了作用。

三、RandomX 演算法

RandomX 是個工作量證明演算法（proof-of-work algorithm, PoW algorithm），希望可以盡可能消除 CPU 和專用硬體間的差距，而其核心就在於它模擬出一個虛擬 CPU，並藉由資料的依存性和隨機性，讓它難以被平行加速。



(圖 1) RandomX 流程圖

RandomX 會接受兩個輸入值，0~60-bit 字串金鑰（K）和任意長度的字串（H），由 K 生成資料集、H 生成隨機程式的資訊，每個隨機程式都有自己的初始值和指令，以及每完成一個程式就與 dataset 中的特定數據進行 XOR，在完成所有的隨機程式後，就可以得到輸出值。

按照操作之間的獨立性，將其分為四個階段：

1. Dataset construction 利用 K 建立 dataset
2. Initialize Scratchpad 利用 H 填滿 Scratchpad 並生成第一個隨機程式
3. VM Programming 執行隨機程式並判斷是否需要生成新的隨機程式
4. Calculate Result 計算最終結果

四、研究過程與結果

4.1 RandomX Benchmark

我們首先使用 Intel® Core™ i7-7600U CPU、RAM 2GB、作業系統為 Ubuntu 18.04 的機器跑 Benchmark 測試 RandomX 各 function 的運行時間，測出來發現在 Cache Initialization 和運算 Hash 的 Hash-test 時間特別長：

- Cache Initialization：Execute Time = 1.385s
- Hash-test：Execute Time = 0.543 ~ 1.343s

4.2 演算法優化

因此我們在 RandomX 程式中尋找和上述功能有關的部分，希望能找到有可能可以進行優化的 function，其中覺得較有潛力的如下：

4.2.1 dataset.cpp

dataset.cpp 因為也有初始化的動作所以相對耗時，不過在 initDatasetItem 中，開發者實際上已將各個 Data item 之間產生的過程獨立，所以可以用平行化來同時進行。原始的程式中，特別將 Dataset 分一半讓兩條 thread 同時初始化 Dataset item，故現在所見相對耗時的結果已經經過加速，沒有再另外改程式優化的空間。

另外，在 initCache 裡的 generateSuperscalar 函式，雖然我們認為這個製造 Superscalar 的過程是可以平行的，但因為它本身的運行時間就非常短（從下圖使用到 generateSuperscalar 的 SuperscalarHash generator 可見，比起上述要用到0.5s 以上的功能還要快很多），即使加速也沒辦法產生重大的效益。

```
[ 2] SuperscalarHash generator                    ... PASSED
Execute_time = 0.002335
```

(圖2) Execute time of SuperscalarHash generator

4.2.2 randomX.cpp

randomX.cpp 中負責運算 Hash 的 randomx_calculate_hash 也是我們想加速的部分，因為從 Benchmark 觀察 Hash 計算是最為耗時的。在程式中有重複8次的迴圈，不過裡頭使用的 blake2b 這個 function 的 output 本身就是第一個參數，雖然迴圈中看起來 tmpHash 沒有被改變，但其實每次迴圈的 tmpHash 都是上一次的結果。因此迴圈之間並不獨立，沒有辦法改成平行計算來加速。

綜合以上，我們認為軟體上修改 code 來加速是不可行的，不過這也側面驗證了前面提到的 RandomX 的資料依存性和隨機性。

4.3 硬體加速

原本，我們想使用 Vivado HLS 將原本的 C++ 程式轉成 Verilog 的程式再做優化，花了許久時間才瞭解如何使用 Vivado HLS，但經過資料收集過後，我們 Vivado HLS 只能將特定寫法寫成的 C++ 程式轉成 Verilog 程式，而且與其改寫 C++，還不如自己寫出 Verilog，比較好處理後面優化的方式。

首先，為了讓 FPAG 板和電腦溝通，因此我們參考 UART 規格寫了 UART 界面。再來，我們嘗試用 Verilog 改寫 RandomX 常用到的函數 blake2b，希望可以透過硬體來加速重複運算的部分。

我們先把程式碼中，每個變數改變的時間和改變成的值記錄下來，分別對每個變數寫一個 always block 用來更新變數，然後寫一個 Finite State Machine 的 always block 記錄現在 State 位置。但丟到 Vivado 後，電路合成（synthesis）一直沒有過，可能是接線的方式出錯，或是因為對 Verilog 不熟悉，因此我們無法得知加速後的結果。

五、結論

總的來說，RandomX 的演算法本身就非常的複雜且開發者已做過部分的平行運算優化，剩下的部分沒有可以平行化或減除多餘的函式的地方，因此我們認為用軟體將 RandomX 加速是不可行的。而在硬體方面，我們還未能成功在 FPGA 上運行 RandomX，故無法測試在 FPGA 上操作對加速 RandomX 的效果，不過鑒於將小型範例程式從 C++ 轉成 Verilog 的成功經驗，我們認為如果能將可能錯誤原因一一排除，在 FPGA 上運行 RandomX 或許是可行的。

六、心得感想

李旻珊：

這次專題我們選擇的題目非常需要自主學習的能力，無論是演算法本身或是加速的方式都沒有很多的資料或完整的研究，真的只能自己 trace code 一點一點去理解、建構出整體架構再從中推測哪些部分可能是能著手的。這樣的過程比想像中還要困難許多，也會感到很挫折，因為演算法的加速很難看出階段性成果，久而久之會有自己好像什麼都沒完成的挫敗感。

不過當我們開始討論專題成果報告的內容，才意識到原來我們為了這個題目吸收了多大量的知識，也許成果和一開始預想的不同，但是所學到有關密碼學、加密貨幣、演算法基礎、RandomX 的知識和 trace code 的能力是真實、一輩子都受用無窮的。

最後要特別感謝我的指導教授和組員。黃之浩教授在每次的 meeting 都會給予我們後續方向的指引；組員廖宏儒更是補足了我在 Verilog 上能力的不足，沒有他我們在 FPGA 上運行 RandomX 的工作是不可能有進展的。

廖宏儒：

在這次的實作專題中，我學習到了以前沒有接觸過的區塊鍊、加密貨幣，並為了深入瞭解加密貨幣的運作原理，於三年級上學期修了與密碼學相關的課程。在實作的過程中，十分有幸可以使用到以前從未看過的 FPGA 礦機，此外，為了將 RandomX 重寫成 Verilog，我花了一個月複習了許久沒用過的 Verilog，學習如何使用 Vivado HLS 將 C++ 程式轉換成 Verilog。在這一年中，我逐漸提升團隊合作的精神、與組員溝通的技能以及與老師簡報的技巧。雖然最後沒有成功將重寫的 Verilog 燒到礦機裡，但經由這次的實作專題使我受益良多，並深刻體認到自己部份實力的不足。最後要謝謝黃之浩教授在我們迷惘的時候指引方向，以及組員李旻珊在實作過程中提出許多優化演算法的方法。

七、參考文獻

[1] Serhack, Mastering Monero. Accessed on May 1st, 2020. [Online] Available: https://github.com/monerobook/monerobook

[2] Riverninj4, Elliptic Curve Point Addition. Accessed on May 1st, 2020. [Online] Available: https://www.youtube.com/watch?v=XmygBPb7DPM

[3] tevador, RandomX, Accessed on May 1st, 2020. [Online] Available: https://github.com/tevador/RandomX