Hashcash
A Denial of Service Counter-Measure

Adam Back 1st August 2002

2022/03/31

組員：吳冠廷、李威辰、張允瀚

Introduction

- Hashcash是一種計算工作量證明的演算法，最一開始用來監測、限制互聯網資源的系統
- 如：Dos攻擊、垃圾郵件
- Hashcash設計了一個cost-function，在郵件標題中加入一個時間戳(stamp)。如果發送的是有意義的郵件，就需要花費一定的CPU時間來生成Hashcash戳(stamp)並發送，以此來證明發送的不是垃圾郵件
- 成本還是也可被用來計算工作量證明機制
- 後來也被用來計算比特幣的獎勵機制

Cost-function（成本函數）

- 成本函數本應該是可以有效驗證的，但由於用參數運算的計算成本太高，因此用符號來定義
- C - Challenge
- CHAL - Challenge：發布一個challenge給client(客戶端)
- s:service-name　　w:work
- T - token：challenge得到的獎勵
- MINT：計算解決challenge得到的token
- V - VALUE：計算出token的價值

$$\begin{cases} \mathcal{C} \leftarrow \mathrm{CHAL}(s, w) \\ \mathcal{T} \leftarrow \mathrm{MINT}(\mathcal{C}) \\ \mathcal{V} \leftarrow \mathrm{VALUE}(\mathcal{T}) \end{cases}$$

interactive cost-function

Publicly Auditable,Fixed,Probabilistic Cost

- Publicly Auditable:成本函數可以被任何第三方驗證，且不用經過任何走後門(trapdoor)或是加密訊息的方式
- Fixed cost:成本函示花費固定的資源來計算，如果要製造固定的token，fixed cost是最快的演算法
- Probabilistic cost:客戶端製造token的時間是可預期的，但如果隨機起始值，有時候因為運氣好抽到接近解答的起始點，就會更有效率的解決

→ unbounded probabilistic cost:儘管花費的時間明顯比預期的時間少的機率為趨近為0，但理論上會花費永遠的時間來計算

→ bounded probabilistic cost:不論客戶端再怎麼不幸運都有一個上限會找到解答

interactive cost-function與non-interactive cost-function

- 在交互成本函數中，伺服器會提供一個challenge給客戶端，可防使預先計算的攻擊，如:每封郵件送出都必須付費，就有可能可以限制垃圾郵件濫用電子郵件的規模
- 在非交互成本函數中，則是客戶端來選擇challenge或隨機產生起始值，用於伺服器沒有通道來發送challenge的時候，如:垃圾郵件
- 但如果有個對手花了很多時間計算tokens並在同一天全部生效，還是可以短暫癱瘓系統一天
- 在非交互成本函數中可以套用交互成本函數的公式，反之則不行

$$\begin{cases} \mathcal{C} \leftarrow \mathrm{CHAL}(s, w) \\ \mathcal{T} \leftarrow \mathrm{MINT}(\mathcal{C}) \\ \mathcal{V} \leftarrow \mathrm{VALUE}(\mathcal{T}) \end{cases}$$

interactive cost-function

$$\begin{cases} \mathcal{T} \leftarrow \mathrm{MINT}(s, w) \\ \mathcal{V} \leftarrow \mathrm{VALUE}(\mathcal{T}) \end{cases}$$

non-interactive cost-function

Trapdoor-free

- Trapdoor-free是已知成本函數中的一個缺點，就是challenger可以廉價的製造任意價值的token
- Trapdoor-free成本函數可以說在製造token上完全沒有優點。
- EX:在計算網頁點擊率時，可能會將點擊率膨脹，以此來跟廣告投放商拿取更高的利潤

Hashcash cost-function

- Hashcash的計算只看service-name(s)
- 由於伺服器只接受自己的service-name製造的token，為了避免伺服器製造的token被其他伺服器拿走，service-name可以由一個獨特的二元字串辨認

Hashcash cost-function

s為二元字串 s={0,1}*

$[s]_i$ 代表s字串的第i項，$[s]_1$ 為第$[s]_{|s|}$ 為最後一項

因此 $s = [s]_{1\ldots|s|}$

定義運算元 $\overset{\text{left}}{=}_b$ 等號左邊字串的第b項跟等號右邊的字串第b項是否相等

$$x \overset{\text{left}}{=}_0 y \quad [x]_1 \neq [y]_1$$

$$x \overset{\text{left}}{=}_b y \quad \forall_{i=1\ldots b}\, [x]_i = [y]_i$$

Hashcash cost-function

- Hashcash cost-function有non-interactive, publicly auditable, trapdoor-free cost-function unbounded probabilistic cost幾種
- Hashcash成本函數是基於在一個全0字串找出partical hash collisions，最快的計算演算法就是暴力解法(brute force)
- 因為每個客戶端都可以安全的選擇自己的隨機challenge，因此Hashcash cost-function是一種non-interactive和trapdoor-free的cost-function，也因為Hashcash cost-function可以publicly auditable因此每個人都可以有效率的驗證已發行的tokens
- 為了避免資料庫無限的成長，可以在service-name的字串中加入服務的時間，這樣就可以讓定時將過期的資料，只是這些合理的丟棄資料應該要考慮到時鐘、計算時間的不準確和計算延遲

Interactive Hashcash

由伺服器選擇用於對TCP、IP、SSH、IPSEC等連線的challenge。

交互hashcash的目的是防止伺服器資源過早耗盡，並且在面對DoS攻擊時提供優雅降級的服務以及對使用者的公平分配。

在TLS、SSH、IPSEC等安全協定中，連線建立階段的成本很高，涉及公鑰加密。被保護的伺服器資源是伺服器可用CPU時間。

Interactive Hashcash cost function

$$
\begin{cases}
\mathcal{C} \leftarrow \mathsf{CHAL}(s, w) & \textbf{choose } c \in_R \{0,1\}^k \\
 & \textbf{return } (s, w, c) \\
\mathcal{T} \leftarrow \mathsf{MINT}(C) & \textbf{find } x \in_R \{0,1\}^\star \textbf{ st } \mathcal{H}(s\|c\|x) \overset{\text{left}}{=}_w 0^k \\
 & \textbf{return } (s, x) \\
\mathcal{V} \leftarrow \mathsf{VALUE}(T) & \mathcal{H}(s\|c\|x) \overset{\text{left}}{=}_v 0^k \\
 & \textbf{return } v
\end{cases}
$$

Cost-function

這裡列出了Cost-function的特徵類別，使用以下這些符號來表示Cost-function

$$([e = \{1, \tfrac{1}{2}, 0\}], [\sigma = \{0, \tfrac{1}{2}, 1\}], [\{i, \bar{i}\}], [\{a, \bar{a}\}], [\{t, \bar{t}\}], [\{p, \bar{p}\}])$$

e代表效率，1效率最好，1/2效率低於1但足以應付，0效率非常低。

σ代表標準差的表徵(characterization of the standard-deviation)，0最快且固定成本，1/2代表bounded probabilistic cost，1代表unbounded probabilistic cost。

Cost-function

$$([e = \{1, \tfrac{1}{2}, 0\}], [\sigma = \{0, \tfrac{1}{2}, 1\}], [\{i, \bar{i}\}], [\{a, \bar{a}\}], [\{t, \bar{t}\}], [\{p, \bar{p}\}])$$

i表示成本函數是可互動的(interactive)，Ī表示成本函數不可互動的(non-interactive)。

α表示成本函數是可被第三方驗證的(publicly auditable)，ā表示成本函數是不可被第三方驗證的(not publicly auditable)。

t表示伺服器在計算成本函數時有後門(trapdoor)， $\bar{t}$表示伺服器在計算成本函數時沒有後門。

P表示成本函數是可平行化的(parallelizable)， $\bar{p}$表示成本函數是不可平行化的(non-parallelizable)。

Cost-function

	Trapdoor-free	Trapdoor
Interactive	Hashcash (e=1,σ=1,i,a, $\bar{t}$,p)	Client-puzzles (e=1,σ=1/2,i,a,t,p) Time-lock (e=1/2,σ=0,i,ā,t, $\bar{p}$)
non-interactive	Hashcash (e=1,σ=1, $\bar{i}$ a, $\bar{t}$,p)	Time-lock (e=1/2,σ=0, $\bar{i}$,ā,t, $\bar{p}$)

challenge-response

每次進行身分認證時，伺服器會發送給客戶端一個不同的challenge，客戶端必須對challenge做出對應的response。

例如:密碼身分驗證，challenge是要求輸入密碼，respons是正確的密碼。

challenge若需要大量執行需要付出很大的努力，這能有效過濾掉垃圾郵件。

Dynamic throttling(動態限流)

使用交互hashcash可以根據伺服器的CPU負載進行動態調整客戶端所需的工作因子。

此方法承認交互hashcash challenge-response只在高負載期間的可能性。

這使得在不破壞舊客戶端軟體向下兼容的情況下逐步抵抗DoS協定是可能的。

在高負載期間，非hashcash客戶端會無法連線，或是被放置在有限的連線池中，受制於較舊、較差的DoS對策，像是random connection dropping。

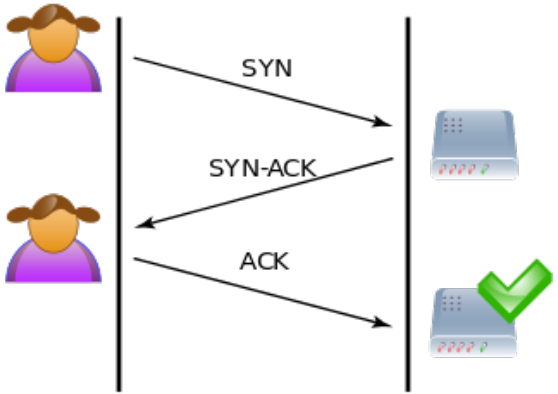
MAC function

訊息鑑別碼 (Message authentication code , 簡稱 MAC) ，訊息鑑別碼 MAC可以實現『身分識別』以及『訊息完整性』

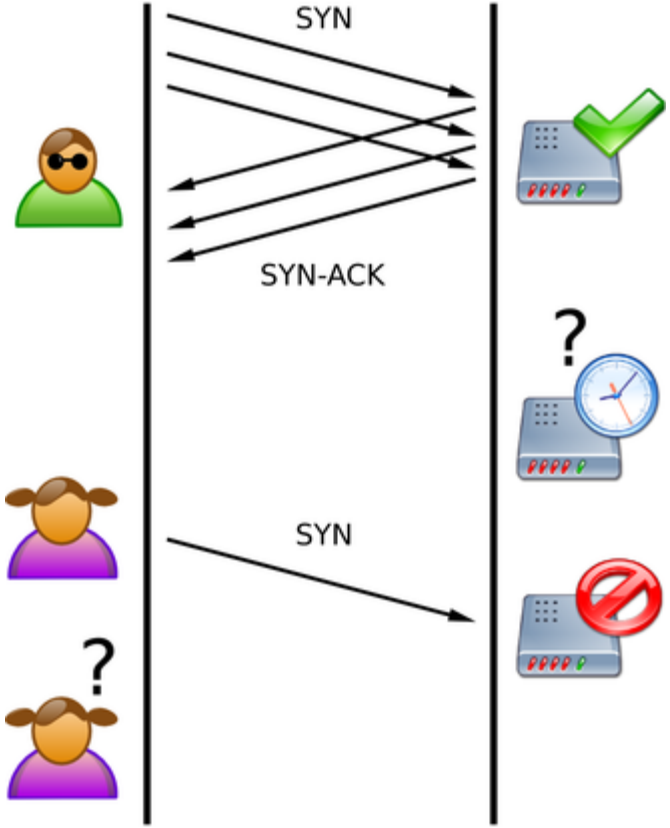
SYN-flood attack

一種DoS攻擊。

目的在透過消耗所有可用的伺服器資源使伺服器無法用於合法流量。



正常TCP三項交握



SYN flood attack

SYN cookie

SYN cookie 是一種用於阻止 SYN flood 攻擊的技術。這項技術的主要發明人 Daniel J. Bernstein 將 SYN cookies 定義為「TCP 伺服器進行的對開始TCP數據包序列數字的特定選擇」。

TCP伺服器收到TCP SYN封包並返回TCP SYN+ACK封包時，會根據這個SYN封包計算出cookie，收到TCP ACK封包時，TCP伺服器根據cookie檢查TCP ACK封包的合法性。

確認合法後才放入connection-slot處理TCP連線。

MAC(訊息鑑別碼 Message authentication code)

因hash無法對應第三方偽造的攻擊，所以使用MAC，MAC可實現身分識別與訊息完整(integrity)。

MAC與hash相同，但會要求一把key，共同持有這把key的雙方能驗證訊息完整，也能確保無法被未持有key的第三方偽造。

MAC不提供不可否認性。

MAC 又稱為Keyed Hash Function。

hashcash-cookie

當遇到SYN-flood攻擊時，TCP connection-slot會儲存在TCP stack中，伺服器資源會因此被耗盡。

在這種情況下會希望避免保持連線狀態，直到客戶端使用交互hashcash成本函數計算出token。

這種防禦類似SYN-cookie對於SYN-flood攻擊的防禦，但作者建議在連線機器上額外增加CPU成本來儲存TCP connection-slot。

Client puzzle

目的是防止濫用伺服器資源，是工作量證明的實作。

若伺服器受到攻擊，則要求連線到伺服器的所有客戶端在連線建立之前解出數學難題，解決難題後客戶端會將答案送回伺服器，伺服器根據驗證決定是否連線。

hashcash-cookie

為了避免將challenge儲存在connection state，伺服器可以選擇計算本來要儲存的訊息的MAC key，並將其作為challenge的一部分發送給客戶端，以便驗證客戶端回傳的challenge跟token的真實性。

(將一條記錄與MAC一起儲存到訊息所涉及的實體的技術被稱為對稱密鑰證書symmetric key certificate)

這種類似SYN-cookie的技術，Juels和Brainard在他們的client-puzzles論文中提出應用程式協定等級的相關方法。

hashcash-cookie

使用伺服器金鑰K作為MAC函數M的輸入，challenge MAC可用下列計算。

$$
\begin{cases}
\text{PUBLIC:} & \text{MAC function } \mathcal{M}(\cdot,\cdot) \\
\\
\mathcal{C} \leftarrow \text{CHAL}(w) & \textbf{choose } c \in_R \{0,1\}^k \\
& \textbf{compute } m \leftarrow \mathcal{M}(K, t\|s\|p\|w\|c) \\
& \textbf{return } (t,s,p,w,c,m)
\end{cases}
$$

hashcash-cookie

客戶端必須發送MAC跟帶有response oken的challenge和challenge參數，以便伺服器可以驗證challenge和response。

伺服器還應該在MAC中包含連線參數，至少足以識別connection-slot跟一些時間大小或遞增計數器，以便在conection-slot空閒後舊challenge-response不會被收集跟重用。

challenge跟MAC會在TCP SYN-ACK response訊息中發送，並且客戶端會在TCP ACK訊息中包含交互hashcash token(challenge-response)。

跟SYN-cookie相同，伺服器在接收到TCP ACK之前不需要儲存每個連線的任何狀態。

hashcash-cookie

為了向下兼容SYN-cookie感知TCP stack，hashcash-cookie感知TCP stack只會在偵測到被TCP 連線耗盡攻擊時才會開啟hashcash-cookie。

Dan Bernstein提出的類似論點可用於表示向下兼容被保留了，也就是Berstein的論點顯示在SYN-flood攻擊下如何提供non SYN-cookie感知實現的向下兼容。

在連線耗盡攻擊下，hashcash-cookie只會在non hashcash-cookie感知TCP stack不可用時開啟。

hashcash-cookie

隨著flood增加，hashcash-cookie演算法會增加TCP ACK訊息中所需的碰撞大小。

假設DoSer的CPU資源有限，hashcash-cookie感知客戶端依然可以用更公平的機會連線對抗DoS攻擊者。

DoS攻擊者將有效的用他的CPU對抗其他所有嘗試連線的客戶端。

如果沒有hashcash-cookie防禦，DoSer可以用連線建立flood伺服器以及藉由每個空閒連現超時完成N個連線(N是連線表大小)來更簡單的占用所有slot，或一次ping連線每個空閒連線超時來說服伺服器他們還活著。

hashcash-cookie

連線會以使用者CPU資源粗略比例集體分發給使用者，因此公平性是基於CPU資源。

所以結果會偏向處理器快的客戶端，他們每秒可以計算更多交互hashcash challenge-response。

Hashcash improvements

在最初發布的hahscash構想中，使用服務名稱的雜湊來公平地選擇目標字串找到雜湊碰撞。

Hal Finney和Thomas Boschloo提出對hashcash的後續改進是找到一個對抗碰撞的固定輸出字串。

它們觀察到固定的碰撞目標是公平、更簡單且可以將驗證成本降低兩倍。

方便比較試驗抵抗碰撞的固定目標字串是K-bit字串0^k，K是雜湊輸出大小。

Low Variance

理想情況下，成本函數tokens應該占用可預測數量的計算資源來計算。

Juels和Brainard的client-puzzle構造通過使用已知解法發出challenge來提供probabilistic bounded-cost。

然而，雖然這限制理論上的最差情況執行時間，但對 variance和典型經驗執行時間的實際差異有限。

使用已知解法的技術也不適用non interactive setting。關於是否存在與hashcash具有相同量級驗證成本的probabilistic bounded-cost或fixed-cost non-interactive成本函數是一個開放問題。

Juels和Brainard帶來的另一個更顯著的改善是建議使用具有相同預期成本但成本差異較小的多個sub-puzzle，該技術應該適用於hashcash的non-interacive跟interactive變體。

Non-Parallelizability and Distributed DoS

Roger Dingledine、Michael Freedman 和 David Molnar 在 [8] 的第 16 章中提出了non-parallelizability成本函數不太容易受到DDoS攻擊的論點。

他們的論點是，non-parallelizability成本函數會阻擋DDoS，因為攻擊者無法分割和移交計算單個令牌的工作。

作者在 [9] 中使用 Rivest、Shamir 和 Wagner 的time-lock puzzle [10] 描述了一個fixed-cost cost-function，它也是non-parallelizable。

time-lock puzzle cost-function可用於non-interactive和interactive setting，因為使用者可以安全地選擇自己的challenge。 Rivest 等人的time-lock puzzle作為cost-function的適用性也被 Dingledine 等人在 [8] 中注意到。

Non-Parallelizability and Distributed DoS

[9] 中基於time-lock puzzle的

fixed-cost和non-parallelizable cost-function：

PUBLIC: $n = pq$

PRIVATE: primes p and q, $\phi(n) = (p-1)(q-1)$

$\mathcal{C} \leftarrow \mathsf{CHAL}(s, w)$
choose $c \in_R [0, n)$
return (s, c, w)

$\mathcal{T} \leftarrow \mathsf{MINT}(\mathcal{C})$
compute $x \leftarrow \mathcal{H}(s \| c)$
compute $y \leftarrow x^{x^w} \pmod{n}$
return (s, c, w, y)

$\mathcal{V} \leftarrow \mathsf{VALUE}(\mathcal{T})$
compute $x \leftarrow \mathcal{H}(s \| c)$
compute $z \leftarrow x^w \bmod \phi(n)$
if $x^z = y \pmod{n}$ **return** w
else return 0

Non-Parallelizability and Distributed DoS

客戶端不知道ϕ(n)，因此客戶端計算MINT()

最有效的方法是重複取冪。

challenger知道ϕ(n)，可以通過減少指數modϕ(n)

以更高效的計算。

因此challenger可以使用2個模冪來執行VALUE()

作為副作用的challenger在計算成本函數時有暗門

因為它可以使用相同算法有效計算MINT()

PUBLIC: $n = pq$

PRIVATE: primes p and q, $\phi(n) = (p-1)(q-1)$

$\mathcal{C} \leftarrow \mathrm{CHAL}(s, w)$
choose $c \in_R [0, n)$
return (s, c, w)

$\mathcal{T} \leftarrow \mathrm{MINT}(\mathcal{C})$
compute $x \leftarrow \mathcal{H}(s||c)$
compute $y \leftarrow x^{x^w} \pmod{n}$
return (s, c, w, y)

$\mathcal{V} \leftarrow \mathrm{VALUE}(\mathcal{T})$
compute $x \leftarrow \mathcal{H}(s||c)$
compute $z \leftarrow x^w \bmod \phi(n)$
if $x^z = y \pmod{n}$ **return** w
else return 0

Non-Parallelizability and Distributed DoS

但作者認為non-parallelizability cost-function提供的額外DDoS保護是微不足道的，除非伺服器限制他向可識別的唯一客戶端發出的challenge數量，否則DDoS攻擊者可以像分配一個分割的challenge一樣輕鬆分配多個challenge，並用與以前相同的速率消耗伺服器資源。此外，單個客戶端偽裝成伺服器的多個客戶端並不難。

還要考慮到DDoS攻擊者通常由於他徵用節點的方法而擁有與處理器相同數量的網路連接節點可供使用。因此他在任何情況下都可以讓每個攻擊結點直接參與正常協定，與其他合法使用沒有區別。

但這種攻擊策略是最佳的，因為攻擊節點提供一組不同的來源地址，這會阻止嘗試每個連接的公平節流策略和基於路由器DDoS策略的基於跨IP地址規範的流量。

因此對於自然攻擊節點marshalling模式，non-parallizable cost-function提供有限的附加抵抗。

Non-Parallelizability and Distributed DoS

除了抵抗non-parallelizability cost-function的實際效能和價值的論點之外，迄今為止，non-parallelizability cost-function的驗證函數比non-parallelizability cost-function慢幾個量級。

這是因為迄今為止在文獻中討論的non-parallelizability cost-function與本質上效率較低的後門公鑰密碼結構有關。是否存在一個未解的問題存在基於對稱密鑰結構的non-parallelizability cost-function。其驗證函數與基於對稱加密的成本函數具有相同量級的驗證函數。

雖然對於將time-lock puzzles應用於成本函數，可以減小公鑰大小來加速驗證函數，但這種方法引入了模數被分解的風險，導致攻擊者在mint token獲得巨大優勢。 （注意：因式分解本身就是一種大部分上是 parallelizable 計算。）

為了解決這個問題，伺服器應該定期更改公共參數，但在time-lock puzzles使用的公共參數的特殊情況下，此操作本身的成本適中，因此不會頻繁執行。為這個應用程式部屬基於低於768位的密鑰大小的軟體可能不太好，此外，它有助於定期更改密鑰，比如每小時。

Non-Parallelizability and Distributed DoS

time-lock puzzle cost-function一定有後門，因為伺服器需要一個私有驗證密鑰來允許他有效驗證token。驗證密鑰的存在增加了密鑰洩漏的風險，讓攻擊者能繞過成本函數的保護(相較之下hashcash的成本函數沒有後門，因此沒有密鑰可以讓攻擊者在計算token時獲得捷徑)。

若驗證密鑰被洩漏，可以被替換，但這需要增加複雜度與管理開銷，因為需要檢測此事件並實施手動干預或一些自動檢測觸發密鑰替換。

time-lock puzzle cost-function也將傾向於具有更大的訊息，因為需要傳達計畫跟緊急的重新加密公共參數。

對某些應用程式而言，SYN-cookie跟hashcash-cookie協定，由於網路基礎設施的向下兼容性與資料大小限制，空間非常珍貴。

Non-Parallelizability and Distributed DoS

因此作者認為non-parallelizability成本函數在防禦DDoS攻擊方面的實用性值得懷疑。更貴的驗證函數會產生驗證密鑰洩漏跟隨之而來的密鑰管理複雜性的風險，更多的訊息實現起要更複雜。

因此作者推薦使用更簡單的hashcash協定。