

FernUniversität in Hagen
Fakultät für Mathematik und Informatik



Hausarbeit zum Thema:

Balloon Hashing

Eine gute Wahl für Passwort Hashing?

Vorgelegt von:

Henry Weckermann
henry.weckermann@studium.fernuni-hagen.de

Matrikelnummer: 3667170
Studiengang: Master Informatik

Abgabedatum: 15.02.2023

Lehrgebiet: Parallelität und VLSI
Seminar: Security Protokolle und ihre Implementierung (01952)
Semester: Wintersemester 2022/2023
Dozent: Prof. Dr. Jörg Keller

Abstract

Durch weltweit zunehmenden Datendiebstahl mittels spezialisierter Hardware gewinnen Passwort Hashing Algorithmen zum Schutz von Nutzerdaten zunehmend an Bedeutung. Der Balloon Hashing Algorithmus spezifiziert eine Art Betriebsmodus für eine kryptographische Hash-Funktion. Die Funktion wurde mit den Entwicklungszielen beweisbarer Memory-Hardness, sowie einem passwort-unabhängigen Speicherzugriffsmuster entworfen. Dabei sollte die Performance mit anderen modernen Passwort Hashing Algorithmen vergleichbar sein. Der formale Beweis der Memory-Hardness wurde jedoch nur im sequenziellen Random-Oracle Model durchgeführt, sodass parallelisierte Angriffe nicht einbezogen wurden. Im parallel Random-Oracle Model (pROM) kann ein erfolgreicher Angriff auf Balloon und weitere Algorithmen gezeigt werden. Ebenso sind keine perfekten Memory-Hard Functions mit passwort-unabhängigem Speicherzugriffsmuster im pROM möglich. Ferner weist die Spezifikation und Referenzimplementierung von Balloon strukturelle Probleme auf. Ziel zukünftiger Forschung könnte die Umsetzbarkeit des gezeigten Angriffs sein. Die Entwicklung einer Funktion mit maximaler Memory-Hardness im pROM sollte zudem in Betracht gezogen werden.

Abstract	I
Abbildungsverzeichnis	III
Tabellenverzeichnis	IV
Abkürzungsverzeichnis	V
1. Einleitung	6
2. Theoretische Grundlagen	7
2.1 Hashfunktionen	7
2.2 Taxonomie und Abgrenzung	7
2.3 Kryptographisch-sicheres Hashing	8
2.3.1 Klassische Anforderungen an kryptographisch-sichere Hashfunktionen	9
2.3.2 Salt	9
2.4 Anforderungen an moderne Passwort-Hashing Algorithmen	10
2.4.1 Random-Oracle Model	10
2.4.2 Memory Hardness	11
2.4.3 Passwort-unabhängige Speicherzugriffsmuster	12
2.5 Graph Pebbling und Data Dependency Graphs	13
3. Balloon Hashing	14
3.1 Einführung und Ziele	14
3.2 Balloon Hashing Algorithmus	14
3.2.1 Implementierung	14
3.2.3 Untersuchung der Urbild-, 2. Urbild-, und Kollisionsresistenz von Balloon	16
3.3 Ziel 1 - Passwort-unabhängiges Speicherzugriffsmuster	17
3.4 Ziel 2 – Vergleichbare Geschwindigkeit	17
3.5 Ziel 3 – Beweis der Memory-Hardness von Balloon	18
4. Diskussion	20
5. Zusammenfassung und Ausblick	22
Literaturverzeichnis	XIII

Abbildungsverzeichnis

Abbildung 1	Hashing Taxonomie von Lianhua und Xingquan (2018)	8
Abbildung 2	Darstellung der Urbild-Resistenz Eigenschaft	9
Abbildung 3	Darstellung der 2. Urbild-Resistenz Eigenschaft	9
Abbildung 4	Darstellung der Kollisionsresistenz Eigenschaft	9
Abbildung 5	SHA-256 Hashrate pro Sekunde für verschiedene Computersysteme	11
Abbildung 6	Platine eines Avalon Bitcoin Mining ASIC	12
Abbildung 7	Beispielhafter Datenabhängigkeitsgraph zu einem gegebenen Algorithmus	13
Abbildung 8	Pseudocode-Implementierung des Balloon Hashing Algorithmus	15
Abbildung 9	Eine alternative Pseudocode Darstellung des zweiten Schrittes des Balloon Algorithmus	16
Abbildung 10	Die Einkern Hashrate pro Sekunde für Balloon, Argon2i, Catena DBG, Catena BRG, PBKDF2 und bcrypt unter Parametern für starke und normale Sicherheitsanforderungen	18
Abbildung 11	Beispielhafter DDG einer Balloon Funktion für den Beweis der Memory-Hardness Eigenschaft	19
Abbildung 12	Die Größe verschiedener Referenzimplementierungen von Passwort-Hashing-Algorithmen in KB	21
Abbildung 13	Nachricht, welche vor dem Einsatz der Referenzimplementierung des Balloon Algorithmus warnt	21

Tabellenverzeichnis

Tabelle 1	Darstellung und Erklärung der verschiedenen Eingabeparameter der Balloon Hashing Funktion	14
-----------	--	----

Abkürzungsverzeichnis

ASIC	Application-specific Integrated Circuit
Catena BRG	Catena Bit Reversalt Graph
Catena DBG	Catena Double Butterfly Graph
CPU	Central Processing Unit
DDG	Data Dependency Graph
DS-GVO	Datenschutz-Grundverordnung
FPGA	Field Programmable Gate Array
GB	Gigabyte
GPU	Graphics Processing Unit
IDS	Intrusion Detection System
KB	Kilobyte
KDF	Key Derivation Function
MD5	Message Digest 5
MHF	Memory-Hard Function
NIST	National Institute of Standards and Technology
PBKDF2	Password-Based Key Derivation Function 2
PHS	Password Hashing Scheme
PRNG	Pseudo-Random Number Generator
pROM	Parallel Random-Oracle Model
RIPEMD	RACE Integrity Primitives Evaluation Message Digest
ROM	Random-Oracle Model
SHA	Secure Hash Function
XOR	Exklusives Oder

1. Einleitung

Seit vielen Jahren nimmt die Anzahl der Datendiebstähle in allen Bereichen des digitalen Lebens zu [1–3]. Oft werden dabei große Datenbestände mit privaten Daten, wie E-Mail-Adressen, Zahlungsdaten und Passwörtern erbeutet. Der entstehende finanzielle Schaden ist - neben dem Vertrauensverlust der Kunden - für die betroffenen Unternehmen beträchtlich [1, 3]. Angreifer nutzen darüber hinaus aus, dass Nutzer ein Passwort oft bei mehreren Diensten verwenden und können sich so über nur einen Diebstahl Zugriff auf mehrere Konten eines Nutzers verschaffen [4–6]. Der Einsatz von einfachen kryptographisch-sicheren Hash-Funktionen (wie SHA-3 [7]) kann jedoch nicht ausreichend sein, um einen Angriff auf gespeicherte Passwörter zu verhindern. Durch die Verwendung von spezialisierter Hardware (ASICs, FPGAs, GPUs) können Angreifer auf diese Art gehashte Passwörter in einer hohen Geschwindigkeit durchprobieren [8–11]. Des Weiteren werden häufig einfache Phrasen als Passwort verwendet, sodass ein Angreifer einen sog. Wörterbuchangriff durchführen kann [12]. Daher werden spezielle Verfahren zum Hashen von Passwörtern, sog. Passwort Hashing Schemes (PHS) empfohlen [13]. Zu diesen Verfahren zählen unter anderem bcrypt oder PBKDF2, die jedoch in ihrem Entwurf keinen Schutz gegen stark parallelisierte Angriffe bieten [14, 15].

Um dieses Defizit zu überwinden, wurden Memory-Hard PHS entwickelt. Diese nutzen unter anderem eine variabel große Menge Arbeitsspeicher, sodass ein rechtmäßiger Aufruf der Funktion, z.B. durch einen Authentifizierungsserver, nicht wesentlich langsamer dauern sollte als der Aufruf eines Angreifers mit spezialisierter Hardware [16, 17]. Diese Eigenschaft erschwert den Einsatz spezieller Hardware, ohne den rechtmäßigen Einsatz zu bestrafen.

Balloon Hashing ist ein von Boneh, Corrigan-Gibbs und Schechter (2016) entwickeltes Memory-Hard PHS welches zusätzlich ein daten-unabhängiges Speicherzugriffsmuster implementiert [18]. Dadurch können Seitenkanalangriffe auf die Balloon Hashing-Funktion verhindert werden. Ein weiteres Ziel der Entwicklung des Algorithmus war eine hohe Leistungsfähigkeit, infolgedessen dieser unter Performancegesichtspunkten mit vergleichbaren Verfahren, wie Argon2 oder Catena, mithalten kann [19–21].

Im ersten Kapitel dieser Arbeit werden relevante theoretische Grundlagen des Hashens sowie Anforderungen an kryptographische Hashing-Funktionen erklärt. Im darauffolgenden Hauptteil wird insb. der Balloon Hashing Algorithmus präsentiert, die Implementierung und Entwicklungsentscheidungen erläutert und der formale Beweis der Memory-Hardness von Balloon dargestellt. Dann folgt eine Diskussion, welche zunächst die Schwachstellen des Algorithmus im parallelen Zufallsorakelmodell erörtert. Daraufhin werden verschiedene Angriffe auf Balloon dargelegt und die allgemeine Eignung von Memory-Hard-Functions (MHF) im Post-Quantum Kontext ermessend. Abschließend wird auf die Qualität der Referenzimplementierung eingegangen.

Das darauffolgende Kapitel gibt eine Zusammenfassung, an die wiederum ein Ausblick auf weitere relevante Forschungsfragen angeschlossen ist.

2. Theoretische Grundlagen

2.1 Hashfunktionen

Hashfunktionen spannen ein sehr großes und diverses Themenfeld auf und können unterschiedlichste Eigenschaften und Ziele verfolgen (siehe Kapitel 2.2). Zwei Eigenschaften haben diese Klassen an Funktionen jedoch gemein. Zum einen hat jede Hashfunktion eine Kompressionseigenschaft. Dies bedeutet, dass Eingaben variabler Länge eine Ausgabe fester Länge erzeugen [22]. So würde sowohl eine 80 Gb große Blu-ray Datei als auch eine kurze Textdatei eine 256 Bit lange Ausgabe erzeugen. Zum anderen haben Hashfunktionen die Eigenschaft „Ease of Computation“ (dt. Einfache Berechenbarkeit) gemeinsam. Dies bedeutet, dass die Berechnung der gegebenen Hashfunktion unter jeder Eingabe x „einfach“ sein soll [22]. Im Folgenden soll die folgende Definition einer Hashfunktion gelten:

$$h: X \rightarrow Y \ (|X| > |Y|, Y = \{0,1\}^n) \quad (1)$$

Die Ergebnismenge Y wird dabei Hash, Hash-Wert, Digest, oder in manchem Kontext auch Fingerprint genannt [22, 23]. Ebenfalls gelten die definitionsbedingten Eigenschaften einer mathematischen Funktion. So ist jede Hashfunktion linkstotal und rechtseindeutig [24]. Es ist daher leicht ersichtlich, dass die wiederholte Eingabe von x immer zu demselben Ergebnis y führen wird.

2.2 Taxonomie und Abgrenzung

Die Anwendungsgebiete von Hashing und Hashfunktionen erstrecken sich über viele Disziplinen der Informatik. Einige bekannte Beispiele sind Anwendungen in der Datenspeicherung und Datenbanktechnik, wo Hash-Werte unter anderem zum Erkennen von Duplikaten und Änderungen an Dateien genutzt werden, sowie zum Erstellen von Indexen und um effiziente Datenstrukturen (Hashmap, Hash Tabelle) zu realisieren [12]. Ebenfalls werden sie im Bereich der Computer- und Netzwerksicherheit eingesetzt, z.B. in Intrusion Detection Systems (IDS) um Änderungen an Datenbeständen zu erkennen oder Muster bekannter Malware in Netzwerkverkehr sichtbar zu machen [25]. Im Bereich der digitalen Forensik ist der Einsatz zur Sicherstellung der Integrität und Nichtabstreitbarkeit von gesicherten Beweismitteln notwendig [26].

Es ist nun wichtig zu differenzieren, für welches Einsatzgebiet die Hashfunktionen entwickelt wurden. So verfolgen Funktionen für effiziente Datenstrukturen unzweifelhaft andere Entwicklungsziele und Eigenschaften als sicherheits-orientierte Hashfunktionen, welche im Weiteren genauer betrachtet werden. Im Folgenden ist nur der Bereich „Unkeyed Cryptographic Hashing“ der Taxonomie von Lianhua und Xingquan (2018) von Relevanz (siehe Abb. 1) [27]. Aus diesem Grund werden im Weiteren alle Funktionen dieser Gruppe einfach als „Hash-Funktion“ o.Ä. beschrieben.

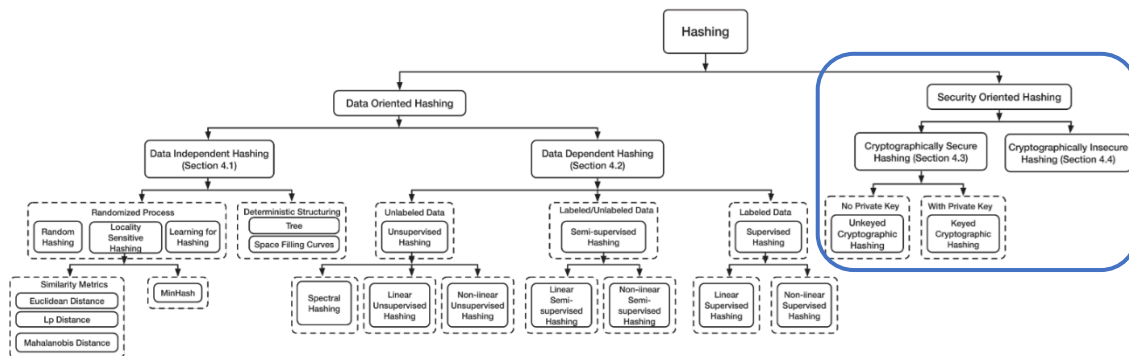


Abbildung 1: Hashing Taxonomie von Lianhua und Xingquan (2018). Es wird eine strikte Unterscheidung zwischen daten-orientierten Hashing und sicherheits-orientierten Hashing gemacht [27].

2.3 Kryptographisch-sicheres Hashing

Der Einsatz von kryptographischen Hashfunktionen ist in vielen täglichen Aufgaben wiederzufinden. So werden nicht nur Passwörter für Online-Dienste, sondern auch die Zugangsdaten von Windows, Linux oder MacOS als Hashwert gespeichert, um sie sicher speichern zu können. Bei der Anmeldung eines Nutzers wird dessen Eingabe nun durch die gleiche Hashfunktion berechnet und die Werte können verglichen werden, ohne dass das Gerät Passwörter im Klartext speichern muss [28]. Ebenfalls können Hashwerte von Nutzern verwendet werden, welche die Integrität einer Downloaddatei überprüfen wollen [29].

Der zusätzliche Aufwand lohnt sich jedoch. Viele Studien zeigen, dass sich die meisten Nutzer Passwörter mit niedriger Entropie ausdenken oder nur ein Passwort für viele Dienste nutzen [4, 5]. Wie die bekannte Passwortdatenbank „rockyou.txt“ zeigt sind z.B. „123456“ oder „password“ eine beliebte Wahl [30]. Um Nutzer sicher zu authentifizieren und zusätzlich im Falle eines Datenlecks Nutzerdaten adäquat zu schützen ist kryptographisch-sicheres Hashing unentbehrlich. Dies wird so seit 2018 in der Europäischen Union durch die DS-GVO sichergestellt [31].

Einige bekannte kryptographisch-sichere Hashfunktionen sind MD5, SHA-1, SHA-256, SHA-512, SHA-3, RIPEMD, BLAKE, PBKDF2 und Argon2 [7, 19, 32–37].

2.3.1 Klassische Anforderungen an kryptographisch-sichere Hashfunktionen

Eine Anforderung an Hashfunktionen ist die Urbild-Resistenz (preimage resistance). Dies bedeutet, dass für einen gegebenen Hash-Wert $y \in Y$ ein Urbild $x \in X$ praktisch nicht berechenbar sein soll [28]. Aufgrund dieser Eigenschaft werden Hash-Funktionen auch als Einwegfunktionen bezeichnet. Es soll also nicht möglich sein aus einem Hash-Wert wieder die Eingabedaten zu berechnen.

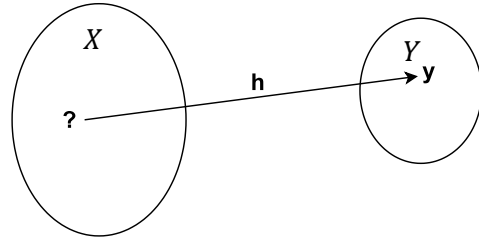


Abbildung 2: Darstellung der Urbild-Resistenz Eigenschaft.

Der Urbild Resistenz schließt sich die 2. Urbild Resistenz (2nd-preimage resistance) an. Diese Eigenschaft zielt darauf ab, dass es praktisch nicht möglich sein soll für eine gegebene Eingabe $x \in X$ eine zweite Eingabe $x' \in X$ zu finden, für welche $h(x') = h(x)$ gilt [22].

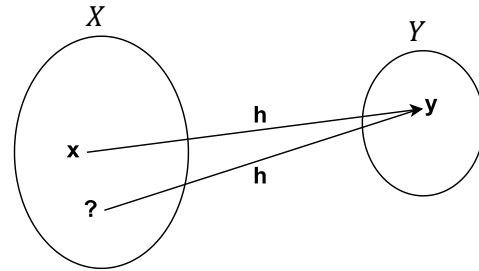


Abbildung 3: Darstellung der 2. Urbild-Resistenz Eigenschaft.

Die dritte Eigenschaft wird Kollisions-Resistenz (collision resistance) genannt. Diese Eigenschaft stellt sicher, dass es praktisch nicht möglich sein soll zwei unterschiedliche Nachrichten $x, x' \in X$ zu finden, welche denselben Hash-Wert ausgeben ($h(x') = h(x)$) [22, 23].

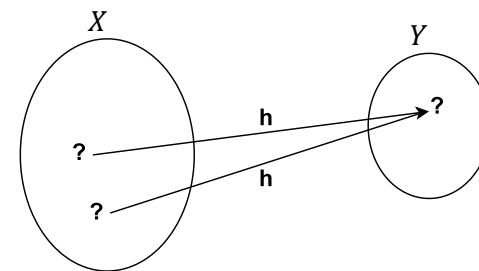


Abbildung 4: Darstellung der Kollisionsresistenz Eigenschaft.

Die Umsetzung dieser Eigenschaften bedeutet jedoch nicht, dass es unmöglich ist ein Urbild, ein 2.

Urbild oder eine Kollision zu finden. Durch das Iterieren aller Möglichkeiten (Bruteforce Angriff) ist es im Durchschnitt möglich einen Preimage oder Second-Preimage Angriff in $O(2^{n-1})$ durchzuführen [38]. Ein Kollisionsangriff ist mit 50% Wahrscheinlichkeit sogar in einer Laufzeit von $O(2^{\frac{N}{2}})$ möglich [28]. Diese Laufzeit lässt sich durch das Geburtstagsparadoxon erklären [39].

2.3.2 Salt

Ein häufiger Angriff auf gehashte Passwörter erfolgt mit sog. Rainbow-Tables. Dies sind große Datenbanken von bereits gehashten Passwörtern mit dem dazugehörigen Klartext [40]. Wenn ein vorbereiteter Angreifer in den Besitz einer Nutzerdatenbank mit Hashwerten kommt, ist es möglich in einer Laufzeit von $O(1)$ einen Klartext zu einem Hashwert zu finden, wenn der betroffene Nutzer ein häufige genutztes oder andernfalls bekanntes Passwort verwendet hat (siehe Abschnitt 2.3). Um diese Art Angriff zu verhindern, wird ein sog. Salt eingesetzt. Es handelt sich hierbei um einen einfachen Bitstring fester Länge (z.B. 256 Bit), welcher zusätzlich zu dem Passworthash gespeichert

wird. Das Salt muss für jedes Passwort einzigartig sein und aus einer kryptographischen Entropiequelle stammen [12]. Sind diese Voraussetzungen gegeben, kann es mit den Nutzdaten zusammen als Eingabe der Hash-Funktion verwendet werden. Statt $h(m)$ berechnet man nun $h(m||s)$, wobei h die Hashfunktion beschreibt, m die eigentlichen Nutzdaten und s das Salt [41]. Es ist ersichtlich, dass dieses Vorgehen die Vorberechnung eines Angreifers unmöglich macht, wodurch dieser die Berechnung erst durchführen kann, wenn er in den Besitz der Daten kommt. Darüber hinaus sorgt das Salt dafür, dass gleiche Passwörter auf verschiedene Hashwerte abgebildet werden.

2.4 Anforderungen an moderne Passwort-Hashing Algorithmen

Neben den bereits aufgeführten Anforderungen an kryptographisch sichere Hashing-Verfahren (siehe Kapitel 2.3.1) entstanden in den vergangenen Jahren einige Algorithmen, welche Entwicklungsziele für den expliziten Einsatz zum Hashen von Passwörtern verfolgen. Einige relevante Grundlagen und Eigenschaften sind im Folgenden aufgeführt.

2.4.1 Random-Oracle Model

Das Random-Oracle Model (dt. Zufallsorakel Modell) (ROM) ist ein Konstrukt, welches eine Brücke zwischen Theorie und Praxis für formelle Beweise darstellt. Das Modell kann für den formellen Sicherheitsbeweis von Hashfunktionen verwendet werden, welche in ihrer Konstruktion Pseudo-Zufallszahlen verwenden, um effizient Zufallszahlen zu approximieren [12, 23, 42]. Zwar ist ein wahres Zufallsorakel nach der Church-Turing These nicht möglich, doch kann so ein formeller Beweis einer konkreten Hashfunktion mit Pseudo-Zufallszahlen durchgeführt werden [43]. Es ist anzumerken, dass ein Beweis im ROM nicht so stark ist wie im kryptographischen Standardmodell. Die einschränkende Annahme ist somit, dass der Pseudo-Zufallszahlengenerator (PRNG) ausreichende Entropie bereitstellt und es keine Nebeneffekte oder strukturelle Sicherheitslücken in der Konstruktion gibt [12].

Ein Zufallsorakel stellt allen Parteien eine ideale kryptographische Hashfunktion zur Verfügung, welche den verwendeten PRNG oder andere Hashfunktionen in der Konstruktion ersetzt. Das Orakel gibt nun für eine arbiträre Eingabe eine Folge echter Zufallszahlen aus, welche in ihrer Länge der Ausgabelänge der ersetzten Funktion angepasst ist. Ebenso wird jede Kombination aus Eingabe und Ausgabe intern gespeichert, sodass die gleichen Eingabedaten immer zur gleichen Ausgabe führen [42, 44]. Das Orakel wird als ideale kryptographische Hashfunktion beschrieben, da es die Eigenschaften Kollisionsresistenz, Urbild-Resistenz und 2. Urbild Resistenz vollkommen umsetzt.

Der formelle Beweis von Balloon Hashing ist im ROM geführt, da in der Konstruktion auf vom Nutzer frei wählbare kryptographische Standard-Hashing-Verfahren wie SHA-2 oder BLAKE2

zurückgegriffen wird [18]. Das ROM formalisiert jedoch nur einen sequenziellen Ablauf der Aufrufe an das Orakel, sodass keine parallelisierte Ausführung einer Funktion betrachtet werden kann. Um dies zu modellieren, gibt es das erweiterte parallel Random-Oracle Model (pROM) (siehe Kapitel 4) [45].

2.4.2 Memory Hardness

Zunehmend wird spezielle Hardware, wie anwendungsspezifische integrierte Schaltungen (en. Application-specific integrated circuit (ASIC)) und Field Programmable Gate Array (FPGA), zur effizienteren Berechnung von Hashfunktionen eingesetzt [8, 46]. Diese Schaltungen sind nur für eine Aufgabe einsetzbar, können diese jedoch mit maximaler Effizienz ausführen. Wie die Namen dieser spezialisierten Hardware bereits vermitteln, kann dies dadurch erreicht werden, dass die Geräte lediglich die Schaltung für eine bestimmte Hashfunktion implementieren und nicht wie eine handelsübliche CPU ein großes Spektrum an Aufgaben abdecken [47]. Die hohe Geschwindigkeit und der vergleichsweise geringe Preis gegenüber handelsüblicher Server-Hardware machen diese Geräte beliebt bei Angreifern (vgl. Abb. 5) [48–50].

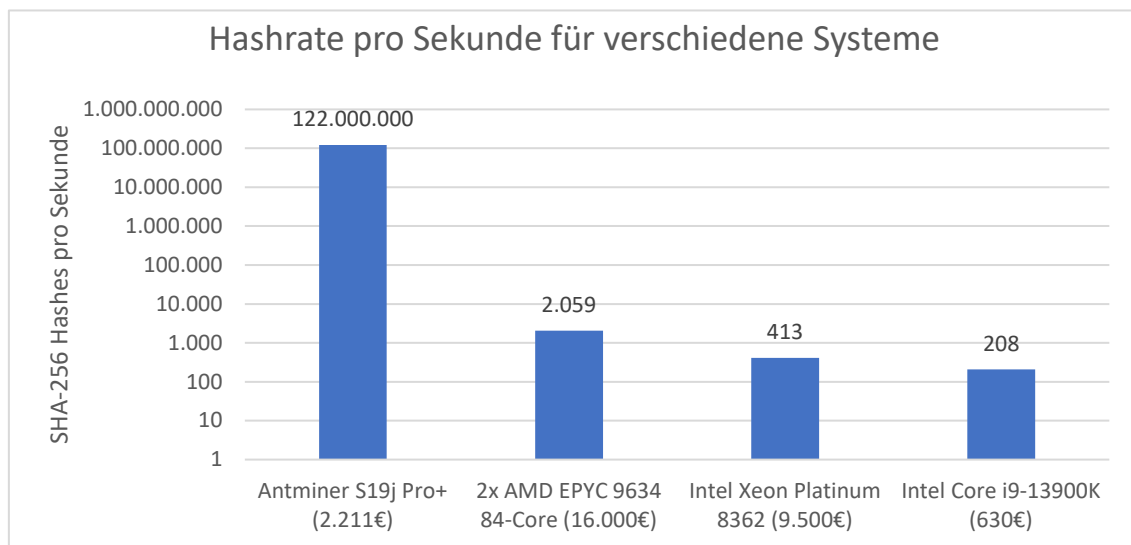


Abbildung 5: Die Abbildung zeigt die Hashrate pro Sekunde eines ASICs gegenüber Server- und Verbraucherhardware. Es werden Antminer S19j Pro+ (ein ASIC), 2x AMD EPYC 9634, ein Intel Xeon Platinum 8362 und ein Intel Core i9-13900k abgebildet. Die Hashrate des Antminer Systems ist ca. 60.000-mal höher als die des 2x AMD EPYC Systems. Die Preise wurden am 10.02.2023 erhoben.

Die Eigenschaft „Memory-Hardness“ (dt. Speicherschwierigkeit) verhindert die effiziente Berechnung einer Hashfunktion auf spezieller Hardware, ohne jedoch die Ausführung auf normaler Hardware substanziell zu beeinträchtigen. Auf diese Art kann der Server einer Webanwendung einen Nutzer effizient authentifizieren, ein Angreifer verliert aber seinen Performancevorteil. Um dieses Ziel zu erreichen, nutzen PHS den Aufbau von ASICs bzw. FPGAs aus. Diese Geräteklasse verfügen meist nicht über großen integrierten Speicher und Caches. Vergleichsweise hoher Speicherbedarf kann so nicht auf spezieller Hardware abgebildet werden und zwingt diese entweder

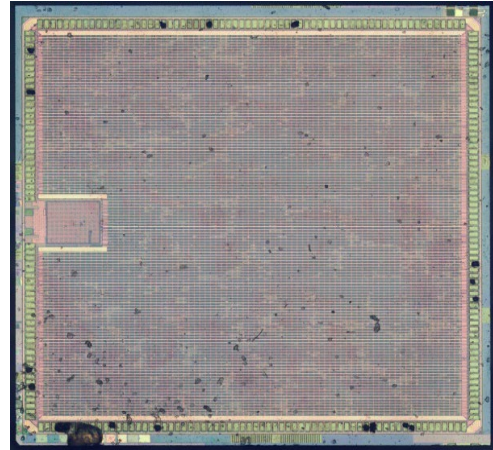


Abbildung 6: Das Bild zeigt die Platine eines Avalon Bitcoin ASIC. Die grünen Pakete am Rande implementieren den Speicher des Gerätes, das quadratische Konstrukt am linken Rand den Controller. Der gesamte Rest dient lediglich der Hash-Berechnung [69].

zur Auslagerung auf langsamen Sekundärspeicher oder zur ständigen Neuberechnung von Werten [45, 51–53]. Dadurch verliert diese Geräteklasse ihren inhärenten Vorteil (vgl. Abb. 6), während handelsübliche Hardware ohnehin über ausreichend Speicher verfügt. Eine handelsübliche CPU hingegen verfügt über genügend Speicher und Caches, da sie ein breites Spektrum an Anwendungsfällen abdecken muss [47]. Formell soll im ROM unabhängig der Berechnungsstrategie das Produkt des Größen-, und des Zeit-Parameters eine asymptotische untere Schranke für n^2 bilden [45]. Dies lässt sich wie folgt ausdrücken:

$$S \cdot T \in \Omega(n^2) \quad (2)$$

Die einfache Iteration kryptographischer Hashfunktionen (z.B. SHA-2) wie es in bcrypt oder PDKDF2 umgesetzt wurde, erhöht im Gegensatz zur Memory-Hardness Eigenschaft, sowohl bei Angreifer als auch Verteidiger die Kosten, ohne jedoch den relativen Vorteil zwischen den beiden zu verringern. Ebenfalls werden im hier betrachteten Kontext des ROM keine parallelisierten Angriffe betrachtet.

2.4.3 Passwort-unabhängige Speicherzugriffsmuster

Ein passwort-unabhängiges Speicherzugriffsmuster (en. password-independent memory access pattern) erschwert Seitenkanalangriffe auf Hashfunktionen. Ein Angreifer könnte das Zugriffsmuster auf den Cache oder die elektromagnetische Abstrahlung unter bestimmten Eingabedaten vorab für viele verschiedene Passwörter aufzeichnen. Während ein kompromittiertes Gerät einen Hash berechnet, kann das akut auftretende Speicherzugriffsmuster mit den bereits aufgezeichneten verglichen werden, um somit Informationen über das Passwort zu erlangen [18, 54, 55]. Dies ist vor allem im Cloud-Computing Umfeld relevant, wo sich viele Nutzer Ressourcen teilen [56].

2.5 Graph Pebbling und Data Dependency Graphs

Das Graphen Pebbling Spiel wird auf gerichtete azyklische Graphen $G = (V, E)$ angewendet. Die Menge V beschreibt die Knoten, E die Kanten. Das Tupel $(u, v) \in E$ stellt eine Kante da, wobei v der Nachfolger von u und u der Vorgänger von v genannt wird. Des Weiteren wird der Knoten, welcher keine eingehenden Kanten hat, Quelle (en. source) genannt und der Knoten, welcher keine ausgehenden Kanten hat, wird als Senke (en. sink) bezeichnet [57]. Im Pebbling Spiel ist es erlaubt einen Spielstein (en. pebble) auf eine Quelle zu setzen. Darüber hinaus kann ein Spieler einen Stein von einem beliebigen Knoten entfernen. Als letzte zulässige Option kann ein Spielstein auf einen Knoten gesetzt werden, aber nur wenn auf einem seiner Vorgänger bereits ein Stein liegt. Das Spiel endet mit dem Platzieren eines Spielsteins auf der Senke [18, 57].

Übertragen auf die Ausführung eines Algorithmus kann man das Spiel auch wie folgt interpretieren. Kanten sind Datenabhängigkeiten zwischen Knoten, welche wiederum Zwischenwerte darstellen. Die Quelle ist somit ein Eingabewert in die Funktion und die Senke stellt die Ausgabe dar. Die platzierten Steine symbolisieren dann Werte, die an einem Punkt in der Berechnung im Speicher vorliegen.

Es gibt verschiedene Strategien einen Graphen mit Spielsteinen zu belegen und mit möglichst wenigen Steinen zur Senke zu gelangen. Jede Strategie gibt ein sog. „space-time-trade-off“ an, wobei die Zeit (en. time) die Anzahl der Schritte und der Platz (en. space) die Anzahl der Spielsteine bedeutet [58, 59].

Ein möglicher Graph, der sich für die Untersuchung mittels eines Pebbling Spiels anbietet, sind die sog. Datenabhängigkeitsgraphen (en. data-dependency graph) (DDG) eines Algorithmus [18]. So kann man z.B. jeden Aufruf einer Hash Funktion als Knoten modellieren, während die Kanten die Datenabhängigkeiten der jeweiligen Eingabedaten darstellen (siehe Abb. 7).

```

$$\begin{aligned}v_1 &= \text{hash}(\text{input}, "0") \\v_2 &= \text{hash}(\text{input}, "1") \\v_3 &= \text{hash}(v_1, v_2) \\v_4 &= \text{hash}(v_2, v_3) \\v_5 &= \text{hash}(v_3, v_4) \\ \text{return } v_5\end{aligned}$$

```

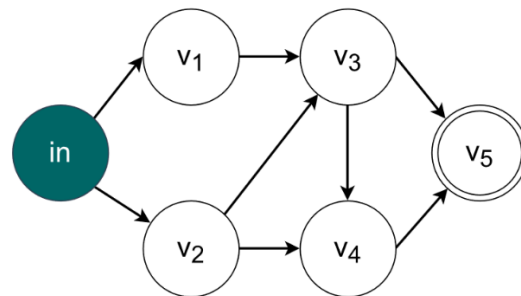


Abbildung 7: Ein beispielhafter Datenabhängigkeitsgraph (rechts) zu dem gegebenen Algorithmus (links) sind dargestellt [18].

3. Balloon Hashing

3.1 Einführung und Ziele

Balloon Hashing ist ein PHS bzw. eine „Key Derivation Function“ (KDF) von Boneh, Corrigan-Gibs und Schechter (2016), welcher eine Art Betriebsmodus für eine darunterliegende kryptographische Hashfunktion definiert [18]. Der Name „Balloon“ leitet sich daraus ab, dass der Speicher „aufgeblasen“ wird, um die Memory-Hardness Eigenschaft umzusetzen. Die Entwicklungsziele des Balloon Algorithmus sind ein passwort-unabhängiges Speicherzugriffsmuster (siehe Kapitel 2.4.3), beweisbare Memory-Hardness im ROM (siehe Kapitel 2.4.2) und mit gleichwertigen KDFs vergleichbare Performance (siehe Kapitel 3.4).

3.2 Balloon Hashing Algorithmus

3.2.1 Implementierung

Die Eingabedaten der Balloon Funktion sind ein Passwort bzw. Eingabedaten, das Salt, ein Zeit- und ein Platz-Parameter (vgl. Tabelle 1). Die Ausgabe der Funktion ist ein Bitstring fester Länge, welche von der Ausgabelänge der intern verwendeten Hashfunktion ist [18].

	Kurzbeschreibung	Variable im Beweis Pseudocode		Beschreibung
Passwort		<i>passwd</i>	<i>password</i>	Ein Passwort bzw. Eingabedaten
Salt		<i>salt</i>		Ein mit ausreichender Zufälligkeit generierter Salt
Time Parameter	Anzahl der Iterationen	<i>r</i>	<i>R</i>	Je größer dieser Parameter, desto länger dauert die Berechnung. Der Parameter passt die Laufzeit der Berechnung an, ohne den Speicherbedarf zu erhöhen
Space Parameter	Puffergröße	<i>n</i>	<i>N</i>	Anzahl von Speicherblöcken die die Funktion während der Berechnung benötigt. Mit <i>n</i> oder mehr Speicher sollte die Funktion einfach zu berechnen sein, sonst schwer

Tabelle 1: Darstellung und Erklärung der verschiedenen Eingabeparameter der Balloon Hashing Funktion. Die sind Passwort, Salt, Time Parameter und Space Parameter.

```

1 Balloon(password, salt, N = space_cost, R = num_rounds):
2
3     delta = 3          // A security parameter
4     B = Buffer[N]      // a buffer of N blocks
5
6     // Step 1: Fill the buffer
7     B[1] ← Hash(password, salt)
8     for i = 2, ..., M:
9         B[i] ← Hash(B[i-1])
10
11
12     // Step 2: Mix buffer
13     for r = 1, ..., R:
14         for i = 1, ..., N:
15             // Chosen pseudorandomly from salt
16             (v_1, ..., v_delta) ← Hash(salt, r, i)
17             B[i] = Hash(B[i-1 mod N], B[i], B[v_1], ..., B[v_delta])
18
19     // Step 3: Extract
20     return B[N]

```

Abbildung 8: Eine Pseudocode-Implementierung des Balloon Hashing Algorithmus. Der Funktion wird ein Passwort, Salt, die Platzkosten N und die Anzahl der Runden R übergeben. Der Aufbau des Algorithmus ist in drei Schritte unterteilt und es wird der letzte Block des Puffers als Ausgabe zurückgegeben. Der Parameter δ wird standardmäßig mit 3 initialisiert [18 p.7].

Der Balloon Algorithmus lässt sich der Ablauf in drei Schritte unterteilen (siehe Abb. 8). Nachdem der Sicherheitsparameter δ (δ) initialisiert wurde und ein Puffer in der Größe der Platzkosten ($space_cost$) angelegt wurde, wird die erste Stelle des Puffers mit der Berechnung einer Hashfunktion aus Passwort und Salt belegt (Abb. 8 Zeile 7). Der Aufruf von $Hash(\dots)$ kann in einer konkreten Implementierung durch den Aufruf einer dem Stand der Technik gerechten kryptographischen Standardfunktion (z.B. SHA-2, SHA-3, BLAKE2) erfolgen [7, 34, 36]. Die Ausgabe des Balloon Algorithmus hat dann die Länge der verwendeten Hashfunktion. In alle weiteren Blöcke des Puffers wird nun der Wert des vorherigen Blockes durch die gewählte Hashfunktion berechnet und abgelegt (Zeile 8 f.). Es entsteht also eine Abhängigkeitskette von Werten, da der Inhalt jedes Blockes vom Wert seines Vorgängers abhängig ist. Der erste Block bildet hier eine Ausnahme, der er keinen Vorgänger hat.

Im zweiten Schritt des Algorithmus wird der nun befüllte Puffer vermischt und überschrieben. Hierbei wird verschachtelt über die Anzahl der Runden (Zeile 13) und die Platzkosten iteriert (Zeile 14). In jeder Iteration wird der Wert des Vorgängers, der Wert des derzeitigen Blocks, und δ -viele pseudo-zufällig bestimmte weitere Blöcke mit der gewählten Hashfunktion berechnet und mit diesem Wert die derzeitige Stelle im Puffer überschrieben (Zeile 17). Die δ weiteren Blöcke werden hierbei pseudo-zufällig durch das Salt bestimmt (Zeile 16). Abbildung 9 verdeutlicht dieses Vorgehen durch eine ausführlichere Darstellung des relevanten Abschnittes.

Im letzten Schritt wird nun die letzte Stelle des Puffers nach sämtlichen Iterationen zurückgegeben. Die Größe des Blockes hängt von der Ausgabegröße der verwendeten Hashfunktion ab [18].

```

...
// Step 2. Mix buffer contents.
for t from 1 to t_cost:
    for m from 1 to s_cost:
        // Step 2a. Hash last and current blocks.
        block_t prev = buf[(m-1) mod s_cost]
        buf[m] = Hash(prev, buf[m])

        // Step 2b. Hash in pseudorandomly chosen blocks.
        for i from 1 to delta:
            block_t idx_block = ints_to_block(t, m, i)
            int other = to_int(Hash(salt, idx_block)) mod s_cost
            buf[m] = Hash(buf[m], buf[other])
...

```

Abbildung 9: Eine alternative Darstellung des zweiten Schrittes des Balloon Algorithmus dargestellt in Pseudocode. Die Auswahl der δ -vielen zufällig gewählten Blöcke soll hier ausführlicher dargestellt werden [18 p.7].

3.2.2 Parallelisierte Ausführung

Der vorgestellte Algorithmus unterstützt durch die starke Abhängigkeit zwischen den Blöcken des Puffers keine Parallelisierung. Es ist jedoch möglich eine Variante zu definieren, welche den Speicher einer M -Kern Maschine schneller füllt. Diese Funktion würde den ersten Schritt des Algorithmus (siehe Abb. 8 Zeile 7 ff.) ersetzen und kann wie folgt definiert werden:

$$\begin{aligned}
 \text{Balloon}_M(p, s) &:= \text{Balloon}(p, s || '1') \oplus \dots \oplus \text{Balloon}(p, s || 'M') \\
 p &= \text{Passwort} \\
 s &= \text{Salt}
 \end{aligned} \tag{3}$$

Die Aufrufe von $\text{Balloon}(\dots)$ können dann parallel stattfinden und die Ergebnisse werden per XOR-Operation verknüpft [18].

3.2.3 Untersuchung der Urbild-, 2. Urbild-, und Kollisionsresistenz von Balloon

Die Anforderungen Urbild-, 2. Urbild-, und Kollisions-Resistenz lassen sich einfach durch die Konstruktion

$$H_B(\text{password}, \text{salt}) := H(\text{password}, \text{salt}, B(\text{password}, \text{salt})) \tag{4}$$

erreichen. B beschreibt hier die Balloon Funktion und H eine kryptographische Hashing-Funktion [18 p.6]. Die Sicherheitsziele der Balloon Funktion werden durch die verschachtelte Verwendung nicht abgeschwächt, da der Algorithmus weiterhin berechnet werden muss. Durch die Verwendung der Ausgabe von Balloon als Parameter in einer kryptographischen Hashfunktion, werden die

klassischen Anforderungen der Urbild-, 2. Urbild-, und Kollisions-Resistenz auf diese ausgelagert. Man kann also sagen, dass jede Eingabe die eine Kollision in B ist keine Kollision in H auslösen kann, da die Kollisionsresistenz für H bewiesen ist.

3.3 Ziel 1 - Passwort-unabhängiges Speicherzugriffsmuster

Wie leicht aus dem Algorithmus (siehe Abb. 8 / Abb. 9) ersichtlich ist, hängt das Muster der Speicherzugriffe pseudo-zufällig vom Salt ab und würde sich auf diese Art auch nicht wiederholen, wenn das gleiche Passwort erneut mit dem Balloon Algorithmus gehasht wird. So kann ein Angreifer nur sehr ineffizient Vorberechnungen anhand der Speicherzugriffe machen. Die Eigenschaft des passwort-unabhängigen Speicherzugriffsmusters (vgl. Kapitel 2.4.3) ist somit erfüllt.

3.4 Ziel 2 – Vergleichbare Geschwindigkeit

Bezogen auf die Performance von Balloon gibt es sowohl theoretische als auch experimentelle Evaluationen. Boneh et al. (2016) entwickelten für den Vergleich von MHF im sequenziellen ROM die Metrik

$$Q[A_S, f] = \frac{ST_f(A_S)}{T_f(Honest)} \quad (5)$$

A_S beschreibt hier einen Angreifer, S den benötigten Speicherplatz, T die Zeit, und f beschreibt die betrachtete Funktion [18 p.17 ff.]. Die Metrik Q soll mit einbeziehen, dass eine gute MHF Performance für legitime Nutzer erhält und Angriffe erschwert. Die folgenden Ergebnisse wurden für einen Angreifer errechnet, welcher ca. 64-mal weniger Speicherplatz verwendet.

$$Q[A_S, Balloon_{(r=1)}] \geq \frac{n}{64} \quad (6) \quad Q[A, Balloon_{(r>1)}] \geq \frac{(2^r - 1)n}{32(r + 1)} \quad (7)$$

$$Q[A_S, Catena-BRG] \geq \frac{n}{32} \quad (8) \quad Q[A_S, Catena-DBG] \geq \frac{r^r}{2r \log_2 n} \quad (9)$$

$$Q[A_S, Argon2i] \geq \frac{n}{1536} \quad (10)$$

Balloon ($r = 1$; siehe Formel 6) ist stets besser als Argon2i (siehe Formel 10) unter Gesichtspunkten der Memory-Hardness in der beschriebenen Metrik Q . Catena-BRG (siehe Formel 8) hingegen schneidet besser ab als Balloon ($r = 1$). Bei einer wachsenden Puffergröße n und einer stabilen Anzahl Runden r kann Balloon besser abschneiden als Catena-BDG (siehe Formel 9). Wenn hingegen r wächst und n statisch bleibt, bietet Catena-DBG die stärkere Memory-Hardness im ROM [18 p.17].

Der experimentelle Geschwindigkeitsvergleich stellt Balloon, Catena (DBG und BRG) und Argon2i gegenüber (siehe Abb. 10). Alle Funktionen wurden in C implementiert und mit Blake2b instanziiert. Ebenfalls sind PBKDF2 und bcrypt in der Grafik als nicht MHF aufgeführt [18 p.19]. Balloon und Argon2i sind mit Parametern für hoher Sicherheitsansprüche gleichauf, während Catena DBG, bcrypt und PBKDF2 nicht dieser Performance entsprechen können. Instanziiert mit Parametern für ein normales Sicherheitsniveau zeigt Balloon einen leichten Vorteil gegenüber Argon2i, fällt jedoch hinter Catena-BRG (siehe Abb. 10).

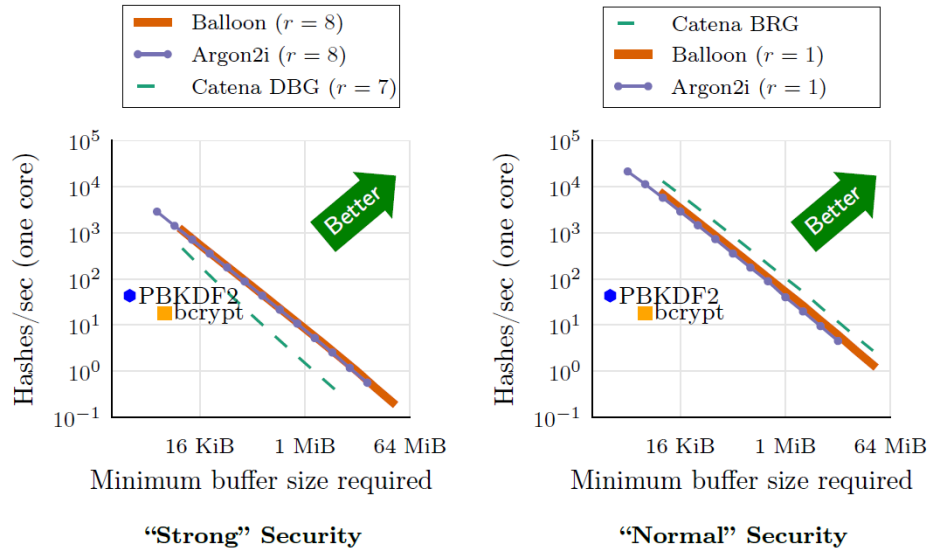


Abbildung 10: Die Einkern Hashrate pro Sekunde für Balloon, Argon2i, Catena DBG, Catena BRG, PBKDF2 und bcrypt. Alle Funktionen sind mit Blake2b instanziiert. Auf der X-Achse ist zudem die minimale Puffergröße angegeben, bei welcher der Algorithmus ohne Einschränkungen ausgeführt werden kann. Die Geschwindigkeiten für "starke" (links) und "normale" (rechts) Sicherheitsanforderungen werden verglichen [18 p.19].

3.5 Ziel 3 – Beweis der Memory-Hardness von Balloon

Um die Memory-Hardness Eigenschaft des Balloon Algorithmus zu untersuchen, wird zunächst der DDG der Funktion (vgl. Kapitel 2.5) betrachtet (siehe Abb. 11). Boneh et al. (2016) zeigten, dass dieser DDG zwei bestimmte Eigenschaften erfüllt. Zum einen ist dies die "Well-Spreadedness" Eigenschaft und zum anderen "Expansion". Das Kriterium der "Well-Spreadedness" wird dadurch erfüllt, dass für jede Menge von k aufeinander erfolgenden Knoten auf einem Level L_x ($x \in \mathbb{N}_0$) mindestens ein Viertel der Knoten des vorherigen Levels auf nicht mit Spielsteinen belegten Pfaden zu diesen k Knoten sind. Die Eigenschaft "Expansion" ist dadurch charakterisiert, dass jede Menge an k Knoten auf einem Level L_x ($x \in \mathbb{N}_0$) nicht mit Steinen belegte Pfade zu mindestens $2k$ Knoten auf dem vorherigen Level hat. Dabei hängt k von dem gewählten Sicherheitsparameter δ ab [18 p.9].

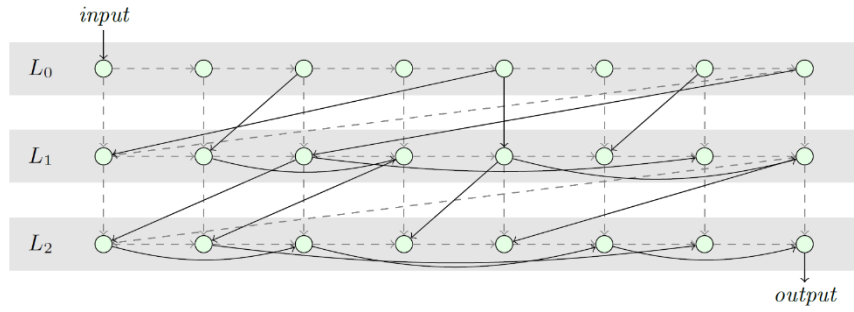


Abbildung 11: Beispielhafter DDG einer Balloon Funktion. Die Funktion wurde mit $n = 8$ Blöcken, $r = 2$ Runden und $\delta = 1$ instanziiert. Die gestrichelten Kanten stellen fest vorgegebene Abhängigkeiten dar, die durchgezogenen Linien sind Abhängigkeiten durch pseudozufällig ausgewählte Blöcke [18 p.10].

Im weiteren Verlaufe des Beweises wird nun gezeigt, dass jeder Angreifer S viel Platz und T viel Zeit verwenden muss, um die Ungleichung $S \cdot T \geq \frac{n^2}{32}$ mit hoher Wahrscheinlichkeit zu erfüllen. Die Idee hinter diesem Vorgehen basiert auf einer Menge mit k vielen Werten, die viele Abhängigkeiten im DDG haben. Wenn ein Angreifer nun weniger Platz (S) verwendet kann er nicht alle diese Abhängigkeiten im Speicher halten, wodurch viele Werte im weiteren Verlauf des Algorithmus wieder berechnet werden müssen. Dies kostet den Angreifer viel Zeit [18 p.10].

Im letzten Schritt wird auf eine frühere Veröffentlichung von Dwork, Naor und Wee (2005) zurückgegriffen, welche eine Methode beschreibt einen gerichteten, azyklischen Graphen in eine Funktion f_G zu transformieren [60]. Diese zeigten ebenfalls, dass wenn ein Graph G für einen zeit-, und platzsparenden Angreifer nicht effizient berechenbar ist, auch die Berechnung der transformierten Funktion f_G dieses Graphen für diesen Angreifer nicht effizient stattfinden kann. Ebenfalls kann für eine derartige Funktion f_G eine obere Wahrscheinlichkeitsschranke angegeben werden, dass ein Angreifer diese mit weniger Zeit und Platz berechnen kann. Boneh et al. [18 p.8] berechnen das Space-Time-Tradeoff für den Balloon Algorithmus mit dem beschriebenen Vorgehen (siehe Theorem 1) [18 p.9 ff.]. So kann man bestimmen, dass ein Angreifer, welcher ein Achtel des Speicherplatzes verwendet, eine exponentiell zu der Anzahl der Runden wachsende Verlangsamung hinnehmen muss. Bei 20 Runden würde der Angreifer ca. das 60.000-fache an Zeit verwenden.

Theorem 1

A ist ein Algorithmus, der eine n -Block, r -Runden Balloon Hashing Funktion mit Sicherheitsparameter $\delta = 3$ berechnet. Die kryptographische Hash-Funktion H ist dabei ein Zufallsorakel. Wenn A nun maximal S Blöcke Puffer nutzt dann wird die Berechnung (mit sehr hoher Wahrscheinlichkeit) so viel Zeit T kosten, dass

$$S \cdot T \geq \frac{r \cdot n^2}{32} \quad (11)$$

erfüllt ist. Bei $\delta = 7$ und $S < \frac{n}{8}$ erhält man

$$S \cdot T \geq \frac{(2^r - 1)}{8 \cdot n^2} \quad (12)$$

4. Diskussion

Da der vorgestellte Balloon Passwort-Hashing Algorithmus viele beweisbare Eigenschaften implementiert (vgl. Kapitel 3), welche Angriffe erschweren, wird dieser vom National Institute of Standards and Technology (NIST) empfohlen [61]. Jedoch müssen für den Einsatz von Balloon einige neuere Erkenntnisse und Probleme einbezogen werden. Die Eigenschaft der Memory-Hardness von Balloon ist lediglich im sequenziellen ROM formalisiert. In diesem Beweismodell wird davon ausgegangen, dass ein Angreifer zu jedem Zeitpunkt nur eine Anfrage an das Orakel stellen kann (siehe Kapitel 2.4.1). Reale Angriffe setzen jedoch schon seit einigen Jahren vermehrt auf extreme Parallelisierung für die Durchführung von Wörterbuchangriffen [8]. Dies liegt unter anderem an der stetig wachsenden Anzahl an Prozessorkernen auf Hauptprozessoren und vor allem Grafikprozessoren [10, 11]. Ebenfalls lässt sich die Aufgabe eines Wörterbuchangriffes trivial parallelisieren. Mit dem parallel Random-Oracle Model (pROM) führten Alwen und Serbinenko (2015) eine Erweiterung des ROM ein, welches die parallele Berechnung ebenfalls mit in den formellen Beweis aufnimmt [45]. So zeigten Alwen und Blocki (2016), dass in diesem Modell keine Funktion mit einem passwort-unabhängigen Speicherzugriffsmuster die Memory-Hardness Eigenschaft perfekt umsetzen kann [62, 63]. Die niedrigste asymptotische untere Schranke die Funktionen wie Balloon erreichen können ist $\Omega(\frac{N^2}{\log n})$. Konkret werden für Balloon amortisierte Kosten von $O(n^{\frac{7}{4}})$ für das Produkt aus Platz und Zeit angegeben [62]. Alwen et al. (2016) konnten jedoch auch zeigen, dass es eine Grenze für die Parallelisierbarkeit von Angriffen auf Balloon gibt [64]. Die von Boneh et al. (2016) bewiesene asymptotische untere Schranke hält zwar in einem realistischeren Szenario nicht vollständig stand und wurde durch Alwen und Blocki (2016) abgeschwächt, doch bietet Balloon weiterhin gute Resistenz gegen Angriffe. Ebenso lassen sich Angriffe nicht unendlich gut parallelisieren, wodurch ein Angreifer weiter abgeschwächt wird. Darüber hinaus schlagen Boneh et al. (2016) eine Kombination aus Balloon und scrypt vor, welche sowohl die Vorteile einer MHF mit passwort-unabhängigem Speicherzugriffsmuster als auch die überlegenen Sicherheitseigenschaften von scrypt im pROM besitzt [18, 65]. So kann der Balloon Algorithmus sicher in einer Produktionsumgebung eingesetzt werden. Eine Überarbeitung des Balloon Algorithmus unter diesen Gesichtspunkten könnte dennoch angemessen sein.

Blocki et al. (2021) entwickelten eine Abwandlung des Pebbling Spiels, welches im Kontext von Quantencomputing eingesetzt werden kann [66]. So untersuchten sie die Sicherheit von MHF in einem Post-Quantum Umfeld und konnten erneut geringere Schranken für die Platz-Zeit Komplexität feststellen. Diese Schranken liegen jedoch immer noch nicht in einem Bereich effizienter Berechenbarkeit. Somit lässt sich festhalten, dass es effizientere Wege gibt, Algorithmen, welche auf DAGs basieren, auf Quantencomputern zu berechnen, diese jedoch die sichere Verwendung des Algorithmus nicht ausschließen. So könnte Balloon nach heutigen Sicherheitsanforderungen

Kritik am Balloon Algorithmus kann weiterhin in Form der Spezifikation und der Referenzimplementierung geübt werden. Der Aufbau der Arbeit von Boneh et al. (2016) verschachtelt die Definition von Balloon mit einem Angriff auf Argon2i [18]. Die zur Verfügung gestellte Referenzimplementierung des Balloon Hashing Algorithmus ist im Vergleich mit vielen anderen PHS eine der längsten (siehe Abb. 12) und schon seit 2020 nicht mehr aktiv durch die Autoren unterstützt [67, 68]. Durch veraltete Abhängigkeiten im Quellcode lässt die Implementierung sich nicht ohne Weiteres auf modernen Systemen kompilieren. Das Projekt scheint vollständig von den Autoren aufgegeben. Dies zeigt sich auch daran, dass eine Warnung der Autoren dargestellt wird, welche vor dem Einsatz außerhalb eines Performance-Tests warnt (vgl. Abb. 13) [67]. Des Weiteren veröffentlichten die Autoren keine weiteren Forschungsergebnisse oder Weiterentwicklungen des Algorithmus und meldeten diesen auch nie für einen Wettbewerb an, sodass er durch die höhere Reichweite weiter untersucht hätte werden können. Der Einsatz von Balloon in einer Produktionsumgebung ist unter diesen Gesichtspunkten weniger empfehlenswert als die Verwendung von Argon2id, zumal die Argon Algorithmen eine höhere Adaption erfahren.



Abbildung 13: Nachricht, welche vor dem Einsatz der Referenzimplementierung des Balloon Algorithmus warnt [68].

5. Zusammenfassung und Ausblick

Balloon Hashing hat beweisbare Memory-Hardness (siehe Kapitel 3.5) und passwort-unabhängiges Speicherzugriffsmuster (siehe Kapitel 3.3) Eigenschaften, während es mit der Performance weiterer renommierter PHS mithalten kann bzw. diese in einigen Fällen übertrumpft (siehe Kapitel 3.4). Diese Charakteristika konnten im ROM (vgl. Kapitel 2.4.1) bewiesen werden. Die Anforderungen an Kollisionsresistenz, Urbild-, und 2. Urbild-Resistenz können mit einer einfachen Erweiterungskonstruktion erreicht werden (siehe Kapitel 3.5). Der Aufbau des Balloon Algorithmus ist dabei sehr einfach und erlaubt den freien Einsatz kryptographischer Hashfunktionen, für welche er quasi einen Betriebsmodus bereitstellt (vgl. Kapitel 3.2). Der Ablauf lässt sich in Befüllen des Puffers, Mischen des Puffers und Extraktion des Ergebnisses unterteilen. Durch die verfolgten Sicherheitsziele kann Balloon effizient von einem Anwender, wie z.B. einem Web-Authentifikationsserver, verwendet werden, jedoch verlangsamt sich die Berechnung je mehr Speicherplatz ein Angreifer mit spezialisierter Hardware (z.B. ASIC) spart. Des Weiteren werden Seitenkanalangriffe durch das passwort-unabhängige Speicherzugriffsmuster (siehe Kapitel 2.4.3) erschwert.

Spätere Erkenntnisse weiterer Autoren konnten die Schranke für die effiziente Berechnung eines Angreifers in einem parallelen Kontext zwar erniedrigen, jedoch nicht auf ein praxisrelevantes Level bringen. Ein Angriff auf den Balloon Algorithmus ist auch mit Quantencomputern nicht in polynomieller Zeit möglich. Letztlich scheinen die Weiterentwicklung und Anpassung der Funktion an die neuen Erkenntnisse dennoch eingestellt. Die Referenzimplementierung wurde in einem unsicheren und unleserlichen Zustand aufgegeben und es wurden keine weiteren Veröffentlichungen bzgl. Balloon vorgestellt. Somit wurde Balloon nicht für das pROM überarbeitet (vgl. Kapitel 4). Die vorgestellte Erweiterungskonstruktion mit scrypt kann jedoch genutzt werden, um Balloon Hashing auch in einem parallelen Kontext sicher zu verwenden.

Im Fokus zukünftiger Forschung könnte die Frage stehen, ob spezielle Hardware für Funktionen wie Balloon oder Argon2 einen parallelen Angriff effizient umsetzen kann. Ebenfalls können diese Ergebnisse mit einem Angriff in sequenzieller Hardware verglichen werden. Basierend auf der Arbeit von Alwen und Blocki (2016) stellt sich darüber hinaus die Frage, ob es quasi perfekte MHF im pROM gibt, welche ein passwort-unabhängiges Speichermuster aufweisen. Wenn es eine solche Funktion gibt, wäre es angebracht die praktische Umsetzbarkeit sowie die Leistungsfähigkeit zu untersuchen [18].

Literaturverzeichnis

- [1] Achim Berg, “Wirtschaftsschutz 2022,” Berlin, 2022. Accessed: Feb. 13 2023. [Online]. Available: https://www.bitkom.org/sites/main/files/2022-08/Bitkom-Charts_Wirtschaftsschutz_Cybercrime_31.08.2022.pdf
- [2] Federal Bureau of Investigation, “Internet Crime Report 2021,” 2021. [Online]. Available: https://www.ic3.gov/Media/PDF/AnnualReport/2021_IC3Report.pdf
- [3] IBM Security, “Cost of a Data Breach Report 2022,” 2022. [Online]. Available: <https://www.ibm.com/downloads/cas/3R8N1DZJ>
- [4] A. Das, J. Bonneau, M. Caesar, N. Borisov, and X. Wang, “The Tangled Web of Password Reuse,” in *Proceedings 2014 Network and Distributed System Security Symposium*, Reston, VA, 2014.
- [5] A. Adams and M. A. Sasse, “Users are not the enemy,” *Commun. ACM*, vol. 42, no. 12, pp. 40–46, 1999, doi: 10.1145/322796.322806.
- [6] S. Gaw and E. W. Felten, “Password management strategies for online accounts,” in *Proceedings of the second symposium on Usable privacy and security - SOUPS '06*, Pittsburgh, Pennsylvania, 2006, p. 44.
- [7] M. J. Dworkin, “SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions,” 2015.
- [8] Electronic Frontier Foundation, *Cracking DES: Secrets of encryption research, wiretap politics & chip design*, 1st ed. Sebastopol, Calif.: O'Reilly, 1998.
- [9] N. G. Einspruch and J. L. Hilbert, *Application specific integrated circuit (ASIC) technology*: Academic Press, 2012.
- [10] *GPU-based password cracking*, 2010. [Online]. Available: <https://rp.os3.nl/2009-2010/p34/report.pdf>
- [11] *GPU-Based High Performance Password Recovery Technique for Hash Functions*, 2016. [Online]. Available: <https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=acc7d3c97e7059df40bd92f768c8fed7ea1c3c10>
- [12] H. C. A. van Tilborg and S. Jajodia, Eds., *Encyclopedia of Cryptography and Security*, 2nd ed. Boston, MA: Springer US, 2011.
- [13] George Hatzivasilis, Ioannis Papaefstathiou, and Charalampos Manifavas, “Password Hashing Competition - Survey and Benchmark,” *Cryptology ePrint Archive*, 2015. [Online]. Available: <https://eprint.iacr.org/2015/265>

- [14] D. M. N. Provos, *Bcrypt algorithm*: USENIX, 1999. [Online]. Available: https://www.usenix.org/events/usenix99/provos/provos_html/node5.html
- [15] K. Moriarty, B. Kaliski, and A. Rusch, “PKCS #5: Password-Based Cryptography Specification Version 2.1,” 2017.
- [16] J. Alwen *et al.*, “On the Memory-Hardness of Data-Independent Password-Hashing Functions,” in *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, Incheon Republic of Korea, 2018, pp. 51–65.
- [17] Niels Kornerup, “Memory hard functions and persistent memory hardness,” 2022. [Online]. Available: <https://www.cs.utexas.edu/~dwu4/courses/sp22/static/projects/Kornerup.pdf>
- [18] D. Boneh, H. Corrigan-Gibbs, and S. Schechter, “Balloon Hashing: A Memory-Hard Function Providing Provable Protection Against Sequential Attacks,” in *Lecture Notes in Computer Science, Advances in Cryptology – ASIACRYPT 2016*, J. H. Cheon and T. Takagi, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 220–248.
- [19] A. Biryukov, D. Dinu, D. Khovratovich, and S. Josefsson, “Argon2 Memory-Hard Function for Password Hashing and Proof-of-Work Applications,” 2021.
- [20] *The Catena Password-Scrambling Framework*, 2015. [Online]. Available: <https://www.password-hashing.net/submissions/specs/catena-v5.pdf>
- [21] Christian Forler, Stefan Lucks, and Jakob Wenzel, “Catena: A Memory-Consuming Password-Scrambling Framework,” *Cryptology ePrint Archive*, 2013. [Online]. Available: <https://eprint.iacr.org/2013/525>
- [22] A. J. Menezes, *Handbook of Applied Cryptography*. Hoboken: CRC Press, 1996. [Online]. Available: <https://ebookcentral.proquest.com/lib/kxp/detail.action?docID=1605633>
- [23] D. R. Stinson and M. B. Paterson, *Cryptography: Theory and practice*. Boca Raton, London, New York: CRC Press, 2019.
- [24] P. R. Halmos, *Naive Mengenlehre*, 5th ed. Göttingen: Vandenhoeck & Ruprecht, 1994.
- [25] H.-J. Liao, C.-H. Richard Lin, Y.-C. Lin, and K.-Y. Tung, “Intrusion detection system: A comprehensive review,” *Journal of Network and Computer Applications*, vol. 36, no. 1, pp. 16–24, 2013, doi: 10.1016/j.jnca.2012.09.004.
- [26] Joachim Metz, *EWFF 2 specification*. [Online]. Available: <https://github.com/libyal/libewff/blob/main/documentation> (accessed: Feb. 13 2023).
- [27] L. Chi and X. Zhu, “Hashing Techniques,” *ACM Comput. Surv.*, vol. 50, no. 1, pp. 1–36, 2018, doi: 10.1145/3047307.

- [28] J.-P. Aumasson, *Serious Cryptography: A Practical Introduction to Modern Encryption*. San Francisco, CA: No Starch Press Incorporated, 2018. [Online]. Available: <https://ebookcentral.proquest.com/lib/kxp/detail.action?docID=5331332>
- [29] *An overview of cryptographic hash functions and their uses*, 2003. [Online]. Available: <http://target0.be/madchat/crypto/papers/paper879.pdf>
- [30] Brannon Dorsey, *naive-hashcat*. [Online]. Available: <https://github.com/brannondorsey/naive-hashcat/releases> (accessed: Feb. 14 2023).
- [31] “Datenschutz-Grundverordnung: DSGVO,” in *Amtsblatt der Europäischen Union (ABl. L 119, 04.05.2016)*, 2016. Accessed: Jan. 21 2023. [Online]. Available: <https://eur-lex.europa.eu/legal-content/DE/TXT/PDF/?uri=CELEX:32016R0679>
- [32] R. Rivest, “The MD5 Message-Digest Algorithm,” 1992.
- [33] D. Eastlake and P. Jones, “US Secure Hash Algorithm 1 (SHA1),” 2001.
- [34] Q. H. Dang, “Secure Hash Standard,” 2015.
- [35] H. Dobbertin, A. Bosselaers, and B. Preneel, “RIPEMD-160: A strengthened version of RIPEMD,” in 1996, pp. 71–82. [Online]. Available: https://link.springer.com/chapter/10.1007/3-540-60865-6_44
- [36] Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O’Hearn, and Christian Winnerlein, “BLAKE2: simpler, smaller, fast as MD5,” in *Applied cryptography and network security: 11th international conference, ACNS 2013, Banff, AB, Canada, June 25-28, 2013 ; proceedings*, 2013, pp. 119–135. [Online]. Available: https://www.researchgate.net/publication/262288030_BLAKE2_simpler_smaller_fast_as_MD5
- [37] B. Kaliski, “PKCS #5: Password-Based Cryptography Specification Version 2.0,” 2000.
- [38] N. P. Smart, *Cryptography: An introduction*. London: McGraw-Hill, 2003.
- [39] K. Suzuki, D. Tonien, K. Kurosawa, and K. Toyota, “Birthday Paradox for Multi-collisions,” in *Lecture Notes in Computer Science*, vol. 4296, *Information security and cryptology - ICISC 2006: 9th international conference, Busan, Korea, November 30 - December 1, 2006 ; proceedings*, M. S. Rhee and B. Lee, Eds., Berlin, Heidelberg: Springer, 2006, pp. 29–40.
- [40] H. Kumar *et al.*, “Rainbow table to crack password using MD5 hashing algorithm,” in *2013 IEEE Conference on Information & Communication Technologies (ICT 2013): Thuckalay, Tamil Nadu, India, 11 - 13 April 2013*, Thuckalay, Tamil Nadu, India, 2013, pp. 433–439.
- [41] P. Gauravaram, “Security Analysis of salt||password Hashes,” in *2012 International Conference on Advanced Computer Science Applications and Technologies (ACSAT 2012): Kuala Lumpur, Malaysia, 26 - 28 November 2012 ; [including papers of the Workshop on Computer Assisted*

Surgery and Diagnostic and the Workshop on Islamic Applications in Computer Science and Technology, Kuala Lumpur, Malaysia, 2012, pp. 25–30.

- [42] M. Bellare and P. Rogaway, “Random oracles are practical,” in *Proceedings of the 1st ACM conference on Computer and communications security - CCS '93*, Fairfax, Virginia, United States, 1993, pp. 62–73.
- [43] B. J. Copeland, *The Church-Turing Thesis*, 1997. Accessed: Feb. 14 2023. [Online]. Available: <https://plato.stanford.edu/entries/church-turing/>
- [44] R. Canetti, O. Goldreich, and S. Halevi, “The random oracle methodology, revisited,” *J. ACM*, vol. 51, no. 4, pp. 557–594, 2004, doi: 10.1145/1008731.1008734.
- [45] J. Alwen and V. Serbinenko, “High Parallel Complexity Graphs and Memory-Hard Functions,” in *Proceedings of the forty-seventh annual ACM symposium on Theory of Computing*, Portland Oregon USA, 2015, pp. 595–603.
- [46] P. Rogaway and T. Shrimpton, “Cryptographic Hash-Function Basics: Definitions, Implications, and Separations for Preimage Resistance, Second-Preimage Resistance, and Collision Resistance,” in *Lecture Notes in Computer Science, Fast Software Encryption*, T. Kanade et al., Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 371–388.
- [47] J. L. Hennessy and D. A. Patterson, *Computer architecture: A quantitative approach*, 3rd ed. San Francisco, CA: Morgan Kaufmann Publishers, 2003.
- [48] L. Dadda, M. Macchetti, and J. Owen, “The design of a high speed ASIC unit for the hash function SHA-256 (384, 512),” in *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, Paris, France, 2004, pp. 70–75.
- [49] A. Satoh and T. Inoue, “ASIC-hardware-focused comparison for hash functions MD5, RIPEMD-160, and SHS,” *Integration*, vol. 40, no. 1, pp. 3–10, 2007, doi: 10.1016/j.vlsi.2005.12.006.
- [50] X. Zhang, R. WU, M. Wang, and L. Wang, “A High-Performance Parallel Computation Hardware Architecture in ASIC of SHA-256 Hash,” in *2019 21st International Conference on Advanced Communication Technology (ICACT)*, PyeongChang Kwangwoon_Do, Korea (South), 2019, pp. 52–55.
- [51] Colin Percival, “STRONGER KEY DERIVATION VIA SEQUENTIAL MEMORY-HARD FUNCTIONS,” 2009. [Online]. Available: <https://www.semanticscholar.org/paper/STRONGER-KEY-DERIVATION-VIA-SEQUENTIAL-MEMORY-HARD-Percival/7c74956d21f0466c9771bb583e2fdf854c2aedbf>

- [52] Niels Kornerup, *Memory hard functions and persistent memory hardness*, 2022. [Online]. Available: <https://www.cs.utexas.edu/~dwu4/courses/sp22/static/projects/kornerup.pdf>
- [53] M. Abadi, M. Burrows, M. Manasse, and T. Wobber, “Moderately hard, memory-bound functions,” *ACM Trans. Internet Technol.*, vol. 5, no. 2, pp. 299–327, 2005, doi: 10.1145/1064340.1064341.
- [54] E. Tromer, D. A. Osvik, and A. Shamir, “Efficient Cache Attacks on AES, and Countermeasures,” (in En;en), *J Cryptol*, vol. 23, no. 1, pp. 37–71, 2010, doi: 10.1007/s00145-009-9049-y.
- [55] M. Zohner, M. Kasper, M. Stottinger, and S. A. Huss, “Side channel analysis of the SHA-3 finalists,” in *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2012.
- [56] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, “Hey, you, get off of my cloud,” in *Proceedings of the 16th ACM conference on Computer and communications security*, Chicago Illinois USA, 2009, pp. 199–212.
- [57] W. J. Paul, R. E. Tarjan, and J. R. Celoni, “Space bounds for a game on graphs,” *Math. Systems Theory*, vol. 10, no. 1, pp. 239–251, 1976, doi: 10.1007/BF01683275.
- [58] J. Hopcroft, W. Paul, and L. Valiant, “On Time Versus Space,” *J. ACM*, vol. 24, no. 2, pp. 332–337, 1977, doi: 10.1145/322003.322015.
- [59] T. Lengauer and R. E. Tarjan, “Asymptotically tight bounds on time-space trade-offs in a pebble game,” *J. ACM*, vol. 29, no. 4, pp. 1087–1130, 1982, doi: 10.1145/322344.322354.
- [60] C. Dwork, M. Naor, and H. Wee, “Pebbling and Proofs of Work,” in *Lecture Notes in Computer Science, Advances in Cryptology – CRYPTO 2005*, D. Hutchison et al., Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 37–54.
- [61] P. A. Grassi *et al.*, “Digital identity guidelines: authentication and lifecycle management,” Gaithersburg, MD, 2017.
- [62] J. Alwen and J. Blocki, “Towards Practical Attacks on Argon2i and Balloon Hashing,” in *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, 2017.
- [63] J. Alwen and J. Blocki, “Efficiently Computing Data-Independent Memory-Hard Functions,” in *Lecture Notes in Computer Science, Advances in Cryptology – CRYPTO 2016*, M. Robshaw and J. Katz, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 241–271.
- [64] J. Alwen, B. Chen, C. Kamath, V. Kolmogorov, K. Pietrzak, and S. Tessaro, “On the Complexity of Scrypt and Proofs of Space in the Parallel Random Oracle Model,” in *Lecture notes in computer science Security and cryptology*, vol. 9666, *Advances in cryptology -*

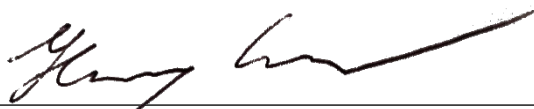
EUROCRYPT 2016: 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016 : proceedings, M. Fischlin and J.-S. Coron, Eds., Berlin, Heidelberg: Springer, 2016, pp. 358–387.

- [65] C. Percival and S. Josefsson, “The scrypt Password-Based Key Derivation Function,” 2016.
- [66] J. Blocki, B. Holman, and S. Lee, “The Parallel Reversible Pebbling Game: Analyzing the Post-Quantum Security of iMHFs,” Oct. 2021. [Online]. Available: <https://arxiv.org/pdf/2110.04191>
- [67] Henry Corrigan-Gibbs, *balloon*. [Online]. Available: <https://github.com/henrycg/balloon> (accessed: Feb. 1 2023).
- [68] G. Hatzivasilis, “Password-Hashing Status,” *Cryptography*, vol. 1, no. 2, p. 10, 2017, doi: 10.3390/cryptography1020010.
- [69] Zeptobars (2013): Avalon - specialized Bitcoin processor. Available online at <https://zeptobars.com/en/read/avalon-bitcoin-mining-unit-rig>, updated on 6/11/2013, checked on 2/14/2023.

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides Statt, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt bzw. nicht veröffentlicht.

Tübingen, den 14.02.2023



Henry Weckermann