CPA-Angriff auf ChaCha20

Bastian Buck and P. Wilhelm Volz

Albstadt-Sigmaringen University of Applied Sciences, Albstadt, Germany,
{buckbast,volzphil}@hs-albsig.de

Abstract. Die Stromchiffre ChaCha20 gilt als direkter Ersatz zur Blockchiffre AES und wird häufig bspw. in TLSv1.3 genutzt. Neben hoher Geschwindigkeit und meist geringer Code-Größe wird sie als sicher angesehen. Dies gilt jedoch nur für typische Angriffe ohne Beachtung von Power-Analysis Seitenkanal-Angriffen. Mit einem solchen Angriff ist es möglich den verwendeten Schlüssel zu extrahieren. Um dies zu unterbinden kommen zwei Gegenmaßnahmen in Frage, welche die Sicherheit erhöhen, die Performanz jedoch schwächen.

Keywords: ChaCha20 · CPA · Correlation Power Analysis · Side-Channel-Attack

1 Einleitung

ChaCha ist eine Familie von Stromchiffren, welche auf den Salsa20 Chiffren basiert und von Daniel J. Bernstein vorgestellt wurde [AFM17]. Sie ist von der IETF in RFC 8439 standardisiert. Dabei wird die Länge des Counters und der Nonce gegenüber der in [Ber08] vorgestellten Version abgeändert. ChaCha wird zunehmend in aktuellen kryptografischen Protokollen darunter TLSv1.2 und TLSv1.3 und verschiedenen Produkten wie dem als sicher geltenden VPN-Protokoll Wireguard verwendet. Durch geringe Komplexität und ausbleibende Angriffsmöglichkeiten blieb dieser Trend bis dato erhalten und wird es voraussichtlich weiter bleiben. Der hier gezeigte Angriff basiert auf der CPA-Methode, die zwar funktional den Schlüssel extrahiert, jedoch großen Aufwand benötigt. Der Correlation-Power-Analysis Angriff (CPA) gehört zu den Seitenkanalangriffen. Er kann dazu verwendet werden, geheime Schlüssel aus kryptografischen Geräten zu extrahieren. In diesem Artikel zeigen wir die Anwendung eines CPA-Angriffs auf den ChaCha20-Verschlüsselungsalgorithmus. Mit Hilfe des CPA kann ein Angreifer den Stromverbrauch oder die elektromagnetische Strahlung analysieren, die von einem kryptografischen Gerät während der Verschlüsselung ausgesendet wird, um den vom Gerät verwendeten geheimen Schlüssel zu extrahieren.

Zunächst geben wir einen Überblick über den ChaCha20-Algorithmus und wie er zur Verschlüsselung von Daten verwendet wird. Anschließend wird erläutert, wie unser CPA-Angriff funktioniert und welche Arten von Seitenkanalinformationen für einen solchen Angriff verwendet werden können. Wir werden auch die möglichen Auswirkungen eines erfolgreichen CPA-Angriffs auf die Sicherheit des ChaCha20-Algorithmus und die Maßnahmen erörtern, die zum Schutz vor solchen Angriffen getroffen werden können. Abschließend werden wir die teils implementierten Gegenmaßnahmen nach aktuellen statistischen Tests auf ihre Nützlichkeit evaluieren.

2 Die ChaCha20 Stromchiffre

ChaCha20 ist eine Stromchiffre, die auf der 20-Runden-Chiffre Salsa20/20 basiert. Die Änderungen von Salsa20/20 zu ChaCha20 zielen darauf ab, die Diffusion pro Runde zu verbessern, was vermutlich den Widerstand gegen Kryptoanalyse erhöht, während die Zeit pro Runde erhalten bleibt - und oft sogar verbessert wird. Neben der Version ChaCha20 gibt es auch ChaCha12 und ChaCha8 mit 12 bzw. 8 Runden. Diese sind je nach Gebrauch in Algorithmen bei denen die Performanz eine essentielle Rolle spielt nützlich. [Ber08] [JB17] Die ChaChaX Familie gilt als ARX-Chiffre, wodurch es bei der Ausführung ausschließlich zur Nutzung von AND, Rotation und XOR Gattern kommt [Ber08]. Wichtig hierbei ist die Differenzierung zwischen Rotate ($\lll$) und Shift ($\ll$). Beispielhaft sei das Byte mit Zustand [00 11 00 11] gegeben. Ein Shift um 3 Stellen nach links würde folgendes Resultat erzeugen [11 00 00 00]. Die Bits werden so aus dem Byte "herausgeshiftet". Ein Rotate hingegen stellt die ausgeshifteten Bits hinten an. Um das Beispiel fortzuführen wäre das Resultat bei Rotate das folgende: [11 00 11 00].

Der initiale Zustand von ChaCha20 besteht aus 16 32-Bit-Worten, die sich wie in Tabelle 1 dargestellt aus Konstanten (C), dem Schlüssel (K_{0-7}), einem Blockzähler (Counter, B) und einer zufälligen Zeichenfolge (Nonce, N) zusammensetzen. Der Kern des ChaCha-Algorithmus ist die sogenannte "Quarter Round". Sie arbeitet mit vier vorzeichenlosen 32-Bit-Ganzzahlen, die wir mit a, b, c und d gekennzeichnet haben. In Abbildung 1 wird der Ablauf einer ChaCha Quarter Round dargestellt. ChaCha20 besteht wie erwähnt aus 20 Runden, die abwechselnd auf einer Spalte und einer Diagonale des Zustands ausgeführt werden. In jeder Runde wird die Quarter-Round-Funktion vier mal hintereinander für jedes der 16 Worte ausgeführt, wobei jedes mal vier neue Werte aus dem Zustand erzeugt werden [NL15].

Nach Abschluss der 20 Runden wird der variable Klartext mittels XOR mit dem Schlüsselstrom verknüpft. Zwar wird bei jeder Rundengenerierung ein 64 Byte großer Block erzeugt, genutzt werden allerdings nur die Menge Bytes, welche auch für die Verschlüsselung notwendig sind. In Kürze bedeutet dies, dass ein 65 Byte Klartext zweimal 20 Runden benötigen würde. 64 Byte des ersten Schlüsselstroms werden für den ersten Teil, 1 Byte des nächsten Schlüsselstroms für das restliche Byte benötigt.

Tabelle 1: Initialer ChaCha20 Zustand

C	C	C	C
K_0	K_1	K_2	K_3
K_4	K_5	K_6	K_7
B	N	N	N

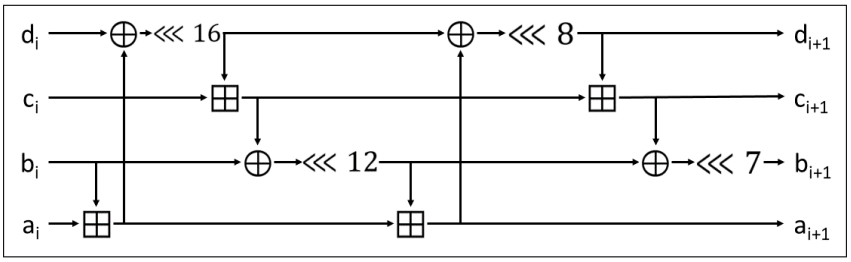

Abbildung 1: ChaCha Quarter Round

Für den in dieser Arbeit beschriebenen Angriff wurde eine in der Programmiersprache C geschriebene ChaCha20-Version aus dem öffentlichen GitHub Repository von shiffthq verwendet [isa16]. Der Code weist keine implementierten Gegenmaßnahmen auf.

3 CPA Angriffsmodell für ChaCha20

Für einen CPA Angriff auf ARX-Chiffren sind verschiedene Herangehensweisen bekannt. Einerseits ist bekannt, dass ARX-Chiffren resistent gegen Timing-Angriffe sind, weswegen solche bereits ausscheiden. In [AFM17] und [JB17] sind zwei Vorgehensweisen gezeigt. Beide Möglichkeiten nutzen dabei das Divide & Conquer Prinzip. Vereinfacht ausgedrückt sind bei 32-Bit Worten $2^{32} \approx 4,294$ Mrd Möglichkeiten für ein solches Schlüsselwort möglich. Dies erscheint zu rechenintensiv und nicht praktikabel. Stattdessen wird in den betrachteten Quellen Byte-für-Byte gearbeitet. Dies entspricht $2^8 = 256$ Möglichkeiten für ein Schlüsselbyte. Für einen 256 Bit Schlüssel entspräche dies $256/8 = 32$ Byteworte, die es herauszufinden gilt. [JB17] [Wu+08] [MAS15]

Um die Korrelation zwischen den Werten einzuordnen eignet sich der Pearson Correlation Coefficient. Dabei sind die benötigten Werte zum einen die Messungspunkte und zum anderen durch nachfolgende Gleichungen dargestellte Leistungsmodelle (Power-Model) der hypothetischen Keybytes. Der Rückgabewert des Pearson Correlation Coefficient ist immer im Bereich +1 und -1, wobei der absolut höchste Wert die beste Übereinstimmung verspricht. [JB17] [Jun22]

Für den hier dargestellten Angriff auf ChaCha20 wurde das Hamming-Gewicht-Modell (HW-Modell) ähnlich zu dem in [JB17] gewählt. Beim Hamming-Gewicht werden die in der Bit-Repräsentation dargestellten Bits gezählt. Beispielsweise wäre das Hamming-Gewicht von [00 11 00 11] = 4, da 4 Bits gesetzt sind. Im Vergleich zum Hamming-Distanz Modell ist nur der aktuelle Zustand wichtig. Bei einer Änderung des vorigen Bytes zu [01 00 01 00] wäre das Hamming-Gewicht 2, die Hamming-Distanz hingegen 6, da sich 6 Bits geändert haben. [MOP08] [JB17] [Jun22]

Um die Schlüsselbytes $K_{0,\cdots,3}$ anzugreifen wurde das folgende Modell (1) verwendet. Hierbei ist + equivalent zu $x + y \mod 2^{32}$. [JB17]

$$HW(d_1) = HW(d_0 \oplus a_1) = HW(d_0 \oplus (a_0 + b_0)) \tag{1}$$

Dabei ist Wert b_0 das anzugreifende Keybyte, welches hierbei die Werte 0-255 (Byte) annehmen kann und somit durch probieren ausfindig gemacht werden soll. Die Werte a_0 und d_0 sind dabei Werte aus den real verwendeten Initialwerten. a_0 ist ein Byte eines Wortes einer Konstanten und b_0 ein Byte des Blockzählers oder der Nonce (je nach Spalten/Diagonalwert). Folglich müssen diese bekannt sein. Die einzige Unbekannte ist demzufolge die Menge der Keybytes, die es zu finden gilt. Auch ist Big- und Little-Endian zu beachten. Das erste angegriffene Byte ist dabei das Least-Significant-Byte (LSB) des ersten Schlüsselwortes. Weiterhin ist zu beachten, dass durch das verwendete XOR zwei gespiegelt gleichwertige Werte beim Angriff aufkommen. Diese befinden sich dann sowohl im negativen als auch positiven Bereich mit dem gleichen absoluten Wert. [JB17]

Für die Schlüsselbytes $K_{4,\cdots,7}$ wurde sodann folgendes Modell (2) theoretisch verwendet.

$$HW(b_1) = HW(b_0 \oplus c_1) = HW(b_0 \oplus (c_0 + \underline{\mathbf{d_2}})) = HW(b_0 \oplus (c_0 + (d_1 \lll 16))) \tag{2}$$

Praktisch zeigte sich die Rotation auf den Bytewerten als erschwerend. Dennoch konnten Teile des Schlüssels mittels des folgenden alternativen Modells (3) genutzt werden. [JB17]

$$HW(b_1) = HW(b_0 \oplus c_1) = HW(b_0 \oplus (c_0 + \underline{\mathbf{d_1}})) \tag{3}$$

Zu beachten sei, dass die Rotation alternativlos entfällt. Da auf Byteebene gerechnet wird, hat dies zwar geringe Auswirkungen, diese sind jedoch weiterhin im Rahmen. Dieser Angriff benutzt sodann den Wert b_0 aus dem ersten Angriffsteil, sowie für den Wert von c_0 die jeweils 256 möglichen Keybytes. Der Wert d_1 ist aus dem ersten Angriffsteil zu übernehmen. Das Ergebnis zeigt nun nicht nur das korrekte Keybyte für c_i, sondern eliminiert auch die falsche Key-Hypothese aus dem ersten Angriffsteil. Folglich werden beide Key-Hypothesen so durch Prüfung der Korrelationskoeffizienten gefunden. [JB17]

3.1 Angriffsumgebung

Der Angriff wurde mit ChipWhisperer Hard- und Software der Organisation NewAE Technology Inc.[1] durchgeführt. Für den programmtechnischen Teil des Angriffs ist zum einen eine vorbereitete virtuelle Maschine mit Python 3.9, Numpy, Jupyter-Notebook, nötigen Compilern etc. installiert, zum anderen finden sich hier die einzelnen Verbindungen zur Messung mittels des genutzten ChipWhisperer-Lite 32-Bit Boards. Dieses besitzt u.a. das nötige Oszilloskop, um die Messpunkte zu erfassen.

Für die Berechnung verschiedener Werte wurde möglichst Numpy verwendet. Zwar sind die Angriffsmodelle auch in Python implementierbar, dennoch zeigen sich hierbei große Performanzeinbußen die durch Nutzung von optimierten Funktionen ausgemerzt werden können. Die Berechnung des Hamming-Gewichts geschieht durch ausschließliche Nutzung der Operationen Shift ($\ll$), AND ($+$), XOR ($\oplus$) und einer Multiplikation ($\cdot$). Die Evaluierung der Performanz findet in Abschnitt 6 statt.

Das Angriffsmodell und die Umgebung zeigen den theoretischen Ablauf, die praktische Ausführung ist mit konkreten Einzelheiten verbunden und wird im folgenden Kapitel erläutert.

4 Ausführung des CPA-Angriffs

Für die praktische Durchführung wurde die in Abschnitt 2 erwähnte Implementierung des ChaCha20-Algorithmus auf dem ChipWhisperer-Lite (32-Bit) über eine angepasste Firmware eingerichtet. Verbindungsspezifische Eigenschaften für die Kommunikation zwischen Rechner und Board wurden eingebracht. Anschließend wurden in verschiedenem Umfang Messungen durchgeführt, wobei jeweils ein neuer Wert für den Blockzähler und die Nonce verwendet wurde. Beide Werte wurden in einem an das Gerät übertragen und von der Firmware geparst. Auch hier ist das Endianness von großer Bedeutung. Die Anzahl der Messungen bewegt sich im Bereich 500 - 50 000. Wobei selbst 500 Messungen gute Ergebnisse erzielen und Werte von 50 000 das natürliche Rauschen mindern konnten. Die Messungen wurden in einem für Numpy optimierten Format zwischengespeichert, um bei Anpassungen in der Analyse nicht die gesamte Messung erneut durchführen zu müssen. Die Messungen dauerten je nach Anzahl zwischen wenigen Minuten bei 500 Messungen und $\sim 1,37$ Stunden bei 50 000 Messungen. Die Zeit ist von der Verbindung zum Board, im vorliegenden Fall USB 3.0, sowie der Hardware selbst abhängig.

Die Messung, wie auch das folgende Angriffsszenario wurden hauptsächlich je mittels eines Jupyter Notebooks in der Programmiersprach Python erzeugt und durchgeführt. Durch die Verwendung von Jupyter Notebooks könnte jedoch der Fall eintreten, dass standardmäßig konfigurierte Limits erreicht bzw. überschritten werden. Um diesen Fall abzufangen haben wird den Softwarecode zusätzlich in einer simpleren Python-Datei zusammengefasst.

[1] https://wiki.newae.com/Main_Page

Der eigentliche Angriff, wie in Abschnitt 3 gezeigt, wird mittels zweier unterschiedlicher Herangehensweisen geführt, wobei letztere vom Ergebnis des ersten Teils abhängig ist. Wie in Abbildung 1 und Gleichung 1 gezeigt, wird mit den Anfangswerten der Konstanten, dem ersten Keybyte und einem Teil des Counters bzw. der Nonce gearbeitet. In diesem Fall wurden folgende feste Werte genutzt:

$$Konstanten : 0x61707865, ...$$

siehe dazu [Ber08].

$$Key : 0x00, 0x01, 0x02, 0x03, 0x04, .., 0x1d, 0x1e, 0x1f$$

$$Counter/Nonce : 0x00, 0x00, 0x00, 0x01 - - - 0x00, ...$$

Der Zähler ist im Little-Endian Format equivalent der dezimalen 1. Die Nonce ist hier ausgenullt. Im Angriffsszenario sind Counter und Nonce pseudozufällig gewählt.

$$Klartext : 0x48, 0x65, 0x6c, 0x6c, ...$$

Der vollständig verwendete Klartext bildet den Satz "Hello from Chipwhisperer, Im only here to get encrypted and you?", was genau 64 Byte, also einem Verschlüsselungsblock entspricht.

Der Angriff für das erste Keybyte (b_0) sähe dann wie folgt aus:

$$HW(\text{0x64 (0b1100100)... }) = HW(0x01 \oplus (0x65 + b_0 \text{ (hier 0x00; für } b_0 = 0 - 255)))$$

Für den zweiten Angriff würde sodann die Hypothese b_0 (hier 0x00) und der Wert von d_1 (hier 0x64) nach Gleichung 3 wie folgt dargestellt:

$$HW(\text{0x74 (0b1110100)... }) = HW(0x00 \oplus (c_0 \text{ (hier 0x10; für } c_0 = 0 - 255) + 0x64))$$

Die Beispiele zeigen den Angriff für das jeweils erste Byte des ersten Schlüsselwortes (der ersten Column-Round).

Um einen automatisierten Angriff durchzuführen sind Schleifen notwendig, die zum einen 16 Iterationen nutzen, wobei jede Iteration dem Angriff auf ein Byte des Chiffrierschlüssels entspricht. In jeder Iteration wird für jede Messung (500 bis 50 000) und jeden möglichen Bytezustand (0-255) das Hamming-Gewicht des Verschlüsselungszwischenergebnisses berechnet. Anschließend wird die Korrelation zwischen diesem und dem der Messpunkte errechnet. Der beste Schätzwert für das aktuelle Byte des Chiffrierschlüssels wird als der Byte-Wert mit der maximalen/minimalen Korrelation ermittelt und in einer Liste gespeichert. Schließlich werden die besten Schätzungen für den Verschlüsselungsstrom (in hexadezimalen Format) ausgegeben.

Die Berechnung des Pearson Korrelationskoeffizienten lässt sich anschaulich an der Abbildung 2 und Abbildung 3 zeigen. Dabei ist bei Abbildung 2 die absolute Korrelation auf Grund des XORs im ersten Angriffsteil immer dieselbe, sodass die erwähnten Zwischenwerte zustande kommen. In Abbildung 3 ist dies nicht mehr der Fall und zeigt für jede Key-Hypothese aus Teil 1 ein eindeutiges Ergebnis.

Im ersten Teil des Angriffs konnten bis zu 10 (von 16) Keybytes erfolgreich gefunden werden. Im zweiten Fall sind durch das benötigte erste korrekte Keybyte deutlich weniger korrekte Ergebnisse zu erwarten. Hier konnten bis zu 4 Keybytes gefunden werden. Dies mag zwar wenig erfolgversprechend erscheinen, ist aber bei Betrachtung der Bitebene ein Irrtum. Demnach zeigt sich bei direktem Bit-Vergleich eine Übereinstimmung von teilweise über 80% (bzw. bis zu 208 von 256 korrekten Keybits). Meist sind Bitkipper dafür ausschlaggebend, dass das Keybyte nicht korrekt ist. Möglicherweise könnten umfangreichere Messungen dieses Problem jedoch beheben.

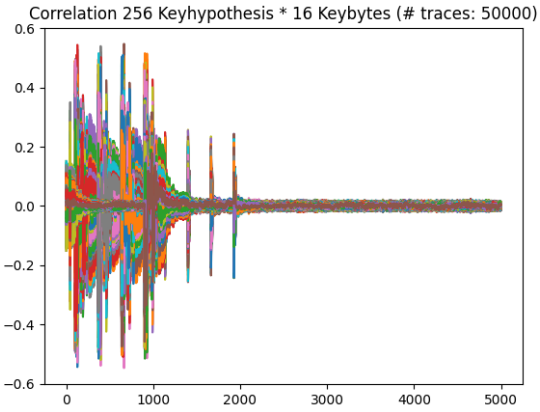

Abbildung 2: ChaCha CPA Attack Part1 (y = correlations; x = # trace-points per trace)

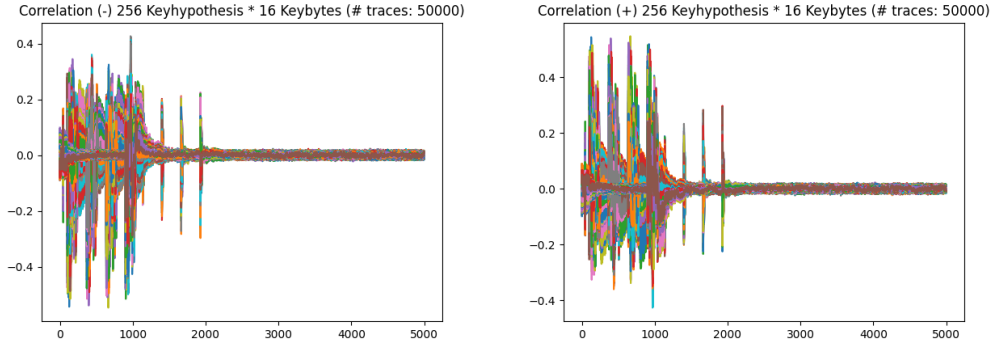

Abbildung 3: ChaCha CPA Attack Part2 (negative (<) & positive (>) b_0 key-hypothesis) (y = correlations; x = # trace-points per trace)

Somit konnte gezeigt werden, dass ein CPA-Angriff auf die Stromchiffre ChaCha20 selbst mit wenigen Messungen effizient innerhalb von Minuten möglich ist. Um die genutzte Variante sicher gegen einen solchen Angriff zu machen, werden wir im folgenden Kapitel Möglichkeiten für Gegenmaßnahmen erklären und eine praktische Implementierung vorstellen und evaluieren.

5 Härtung der ChaCha20 Implementierung

Prinzipiell sind zwei Arten von Gegenmaßnahmen möglich. Beim Hiding wird das Signal-to-Noise Verhältnis gestört. Das Rauschen wird erhöht und die Korrelation zwischen den Werten schwindet. Masking hingegen maskiert die Zwischenwerte im Idealfall vollständig, sodass ein Angriff nicht mehr möglich ist. Eine perfekte Variante ist jedoch praktisch nicht möglich. [JB17] Für jede der Gegenmaßnahmen wird Zufall benötigt. Dies wird in diesem Projekt mittels pseudozufälligen Zahlen durchgeführt. Es gibt neben der Funktion *rand*() in C auch Möglichkeiten wie den Mersenne Twister oder optimierte PRNG wie den in diesem Projekt verwendeten XorShift128+ (siehe [Vig16]). Dieser nutzt zwei 64 Bit Zahlen als Seed und erzeugt so eine pseudozufällige 64-Bit Zahl. Diese kann dann für die jeweiligen Zwecke genutzt werden. Im vorliegenden Fall entspricht der Seed den ersten beiden Worten der Nonce. Dies darf allerdings nicht auf eine praktische Implementierung angewendet werden.

5.1 Masking der Quarterround

Konzeptionell könnte eine effektive Gegenmaßnahme das Maskieren der internen Zustände der Quarterround Funktion mittels XOR darstellen. In dieser werden wie erwähnt ausschließlich Operationen wie ADD, Rotate und XOR verwendet. Eine Maskierung mittels ADD oder auch MUL wäre unnötig schwierig und hätte Einbußen in der Performanz. Für das Maskieren von XOR müsste ein Zufallsbit für ein zu maskierendes Bit erzeugt werden. Für Rotate ist kein seperater Zufall nötig. Für das Maskieren der Addition modulo 2^{32} ist jedoch keine triviale Möglichkeit vorhanden. Hierfür kann das Trichina AND-Gate genutzt werden, was jedoch ebenfalls ein weiteres Zufallsbit für jedes AND bedeutet und vier Zwischenwerte benötigt. [Jun22] [MOP08] Auf Grund dieser Hürde wurde in dieser Arbeit auf eine Implementierung verzichtet.

5.2 Hiding: Shuffling der Ausführungsreihenfolge der Quarterrounds & Dummy Instructions

Hiding-Maßnahmen können verschiedene Ausuferungen annehmen. Im vorliegenden Fall sind Shuffling und Dummy Instructions einfach implementiert und können großen Nutzen darstellen. Die in Abschnitt 2 erwähnte C-Implementierung von ChaCha20 gilt es nun mittels einer Kombination aus ebendiesen Hiding-Maßnahmen zu härten, sodass das Rauschen eine Auswertung bei einer geringen Anzahl Messungen nicht trivial möglich macht.

Shuffling bedeutet die zufällige Aneinanderreihung von Funktionsaufrufen, sodass der Endwert trotzdem dem korrekten Pendant entspricht. Dummy Instructions werden zwischen den jeweiligen Funktionsaufrufen so untergeschoben, sodass diese nicht trivial auszufiltern sind. Dabei haben diese keinerlei Funktionalität zum eigentlichen Kryptoalgorithmus. [Jun22]

Wir haben eine Verflechtung zwischen Shuffling und Dummy Instructions erreicht, sodass bei jedem Durchgang zufällig viele Dummy Instructions in verschiedenster Reihenfolge ausgeführt werden. Das Prinzip wird in der ChaCha20 Block Funktion eingebaut und nutzt den erwähnten XorShift128+ PRNG (siehe [Vig16]) für die Generierung pseudozufälliger Zahlen. Da der/die Counter/Nonce bei jedem Aufruf abgeändert wird, ist ein voriges automatisches Ausfiltern nicht vorhanden. Unser Algorithmus bedient sich folgendem Ablauf innerhalb der Block-Funktion sowohl bei den Spalten- als auch Diagonalrunden:

1. Erzeuge Array *done* mit Zwischenwerten und Zielwert y
2. Solange Ziel nicht erreicht:
   (a) Erzeuge Zufallswert x zwischen 0 und #-Runden.
   (b) Erzeuge solange bis x erreicht ist neue Zufallszahlen rnd_i im Bereich 0 und 3.
   (c) Prüfe ob Ausführung der Runde (Column(0-3) oder Diagonal(0-3)) bereits in *done* festgehalten wurde.
   (d) Falls bereits ausgeführt, beginne neu ab Punkt 2; ansonsten führe Runde aus.
3. Prüfe ob alle Runden ausgeführt; wenn ja setze Zielwert y, ansonsten fahre fort.

Trivial zu erkennen ist, dass je nach bereits aufgerufenen Rundenfunktionen die Wahrscheinlichkeit, die korrekte Zufallszahl für die nächste Rundenfunktion zu erhalten zunehmend gering wird. Folglich kommt es zu stark ansteigenden aufrufen an Dummy-Instructions, welche je nach Performanzanforderungen im Bereich Punkt 2a bei der Anzahl Runden angepasst werden können. Es hat sich gezeigt, dass selbst das Weglassen dieser zusätzlichen Dummy-Instructions die Verwertung niedriger Messpunkte stark erschwert. Dazu in Abschnitt 6 mehr dazu.

6 Evaluation der ChaCha20 Härtung

Nach der Implementierung der Hiding Maßnahmen wurde der selbe Angriff wie bei der ungeschützten Version durchgeführt. Es wurden ebenfalls wieder mehrere Messungen in unterschiedlicher Größe erstellt und analysiert. Die Darstellungen (Plots) des Angriffs auf die ungeschützte Implementierung (Abbildung 2 und Abbildung 3) verdeutlichen im Vergleich zu der Darstellung des Angriffs auf die gehärtete Implementierung (Abbildung 4) die Funktionalität der Gegenmaßnahme. Es ist sehr deutlich zu sehen, dass hier keine bzw. nur wenige Korrelationen (zu sehen in Form von "Peaks") gefunden werden können. Dies bestätigt sich weiter in der Ausgabe des Angriffs. Hier konnte keines der verwendeten Schlüsselbytes korrekt bestimmt werden. Eine Messung mit 50 000 Messungen zeigte das gleiche Ergebnis, konnte allerdings erste konkretere Peaks im Bereich der ersten 1000 Messpunkte verschiedener Messungen ausmachen.

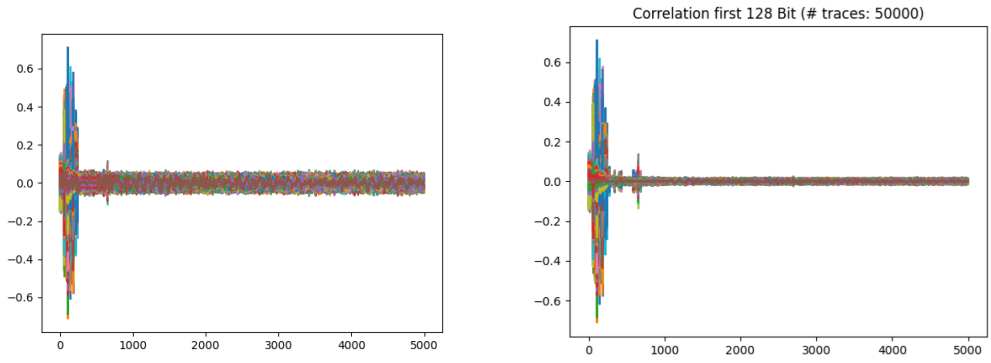

Abbildung 4: Shuffled CPA Attack 500 (<) and 50k (>) Traces Part1 (y = correlations; x = # trace-points per trace)

6.1 Angriff auf die gehärtete ChaCha20 Implementierung und Vergleich

Für die Durchführung des Angriffs auf die gehärtete Implementierung wurde der gleiche Softwarecode verwendet wie für die ungeschützte Implementierung. Der Ablauf des Angriffs ist also in beiden Fällen gleich. Durch das Einfügen der Dummy Instructions und die Mischung der relevanten Quarterround-Funktionsaufrufe des ChaCha20-Algorithmus wird das Signalrauschen in solchem Maße verstärkt, dass die Bestimmung des Erwartungswertes nicht mehr treffend durchgeführt werden kann. Deutlich wird dies ebenfalls bei der Betrachtung der grafischen Darstellung der Messungen, wie in Abbildung 5 zu sehen. Im linken Diagramm (ohne Gegenmaßnahmen) ist ein klares Muster erkennbar, während im rechten Diagramm die Identifizierung eines Musters deutlich schwerer fällt. Um den Angriff weiterhin erfolgreich durchführen zu können, muss die Anzahl der Messungen erhöht werden um das Rauschen ausreichend abzuschwächen. Ab einem bestimmten Punkt sind hierbei jedoch Probleme in der Verarbeitung der Messwerte, sowie die Verarbeitungsdauer zu erwarten. Diese Probleme können zwar behoben werden, benötigen jedoch einen neuen Angriffsansatz der diese mit einbezieht.

Ein weiteres Vorgehen könnte das Herausfiltern der Dummy Instructions und Vorhersage der Randomisierung der ChaCha20-Instruktionen darstellen. Das Filtern der Dummy Instructions ist vergleichsweise einfach zu erreichen. Hierzu können sich wiederholende immergleiche Messbereiche als Anhaltspunkte dienen. Bei der Vorhersage der zufällig ausgeführten Instruktionen ist die Auflösung schwieriger und nur realistisch umsetzbar, wenn für die Randomisierung ein schlechter Zufallserzeuger verwendet wurde. Im vorlie-

genden Fall ist der genutzte PRNG (XorShift128+) zwar ausreichend sicher, nutzt aber als Anfangsseed die ersten beiden 32-Bit Worte der Nonce. Folglich kann ein Angreifer, wenn er im Besitz der Nonce ist, den vollständigen Zufallsstrom voraussagen und somit ausgeführte Dummy-Instructions herausfiltern. Zudem kann sodann auch die Aufrufreihenfolge vorhergesagt und der Angriff entsprechend angepasst werden.

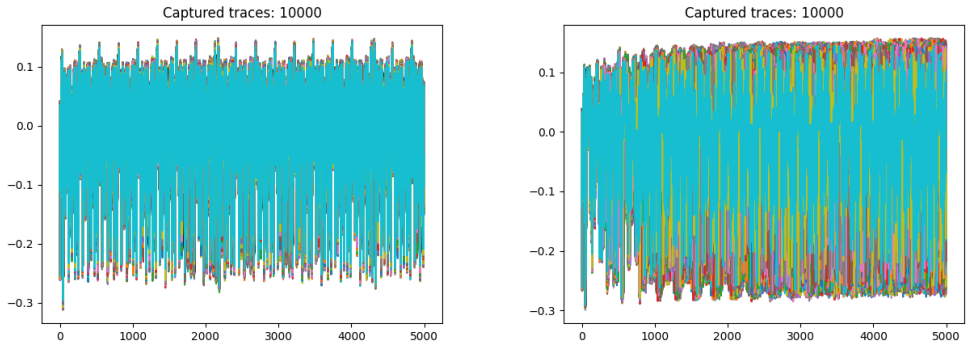

Abbildung 5: Measured 10 000 traces (no countermeasure (<) & hiding (>))

6.2 Statistischer Leakagetest

Um die Aussagekraft der Messungen zu prüfen, kann Welch's t-Test dienen. Hierzu werden Messungen mit dauerhaft festen Werten mit denen randomisierter Inputs verglichen. Dabei sei die Null-Hypothese zwischen $[-4.5, 4.5]$ akzeptiert (in diesem Bereich lassen sich tendenziell keine Informationen entnehmen). Werte darunter und darüber könnten somit Informationen liefern, die auf die Möglichkeit eines funktionierenden Angriffs hinweisen. [JB17] [Jun22] Die durchgeführte statistische Analyse mit Welch's t-Test zeigt, dass die normale Implementierung des ChaCha20-Algorithmus ein sehr deutliches gleichmäßiges Leakage aufweist (siehe Abbildung 6 links). Auch für die gehärtete Variante ist weiterhin ein Leakage zu erkennen (siehe Abbildung 6 rechts). Hier zeigen sich allerdings zu Beginn außerordentlich hohe Werte. Die nachfolgenden Werte zeigen wiederum vergrößert ähnliche Werte. Darin liegt die Begründung, warum der Angriff auf diese ChaCha20-Implementierungen im Allgemeinen funktioniert bzw. im letzteren Fall funktionieren kann.

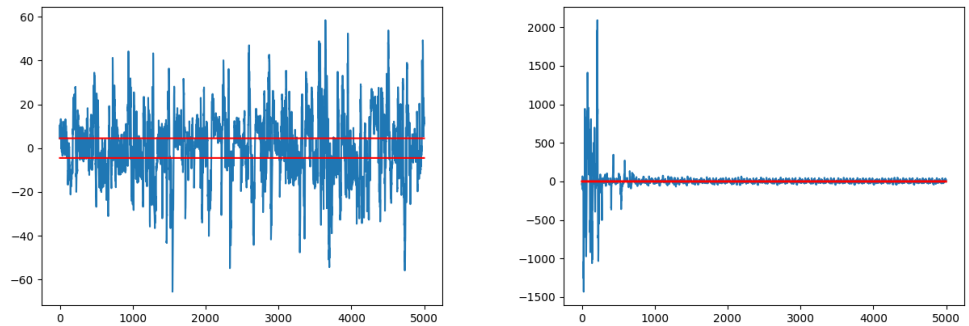

Abbildung 6: Welch't t-test with 500 traces (without countermeasure (<) & hiding (>)) (y = leakage; x = # trace-points per traces)

7 Fazit

Zusammenfassend lässt sich zum Angriff auf den ChaCha20-Algorithmus sagen, dass dieser durchaus möglich und sogar zu großen Teilen erfolgreich ist. Mit einer Extraktion von $\sim 83\%$ der Schlüsselbits ist bereits ein bedeutender Schritt gemacht und somit deutlich nachvollziehbar, dass eine Vulnerabilität gegenüber einem CPA Angriff besteht. Um die restlichen Schlüsselbits zu finden könnten weitere Ideen zum Zuge kommen, deren Umsetzung in einer weiterführenden Arbeit geprüft werden müssten. Eine weitere Option könnte die Verwendung eines anderen Angriffsmodells darstellen. Möglicherweise ist die Verwendung des Hamming-Distanz-Modell in diesem Fall vorteilhaft. Auch eine Analyse des erzeugten Assemblercodes führt möglicherweise zu weiteren Erkenntnissen die für die Optimierung des Angriffs genutzt werden können.

Um mit einer größeren Anzahl Messungen zu arbeiten, ist es nötig die Berechnung des Pearson-Korrelationskoeffizienten auf diesen Anwendungsfall hin auszulegen. Dafür kann der von Socha u. a. in [Soc+17] vorgeschlagene Algorithmus zur iterativen Berechnung des Pearson-Korrelationskoeffizienten verwendet werden.

Referenzen

[AFM17] Alexandre Adomnicai, Jacques JA Fournier und Laurent Masson. *Bricklayer attack: a side-channel analysis on the ChaCha quarter round*. Springer, 2017. URL: https://link.springer.com/chapter/10.1007/978-3-319-71667-1_4 (besucht am 07. 02. 2023).

[Ber08] Daniel J. Bernstein. *ChaCha, a variant of Salsa20*. 2008. URL: http://cr.yp.to/chacha/chacha-20080128.pdf (besucht am 04. 02. 2023).

[isa16] isayme. *ChaCha20 stream cipher in C*. 2016. URL: https://github.com/shiffthq/chacha20 (besucht am 06. 02. 2023).

[JB17] Bernhard Jungk und Shivam Bhasin. *Don't fall into a trap: Physical side-channel analysis of chacha20-poly1305*. IEEE, 2017. URL: https://ieeexplore.ieee.org/abstract/document/7927155 (besucht am 07. 02. 2023).

[Jun22] Bernhard Jungk. *Vorlesung: Implementation Attacks and Countermeasures (WiSe 22/23)*. 2022.

[MAS15] Bodhisatwa Mazumdar, Sk Subidh Ali und Ozgur Sinanoglu. "Power analysis attacks on ARX: an application to Salsa20". In: *2015 IEEE 21st International On-Line Testing Symposium (IOLTS)*. IEEE. 2015, S. 40–43.

[MOP08] Stefan Mangard, Elisabeth Oswald und Thomas Popp. *Power analysis attacks: Revealing the secrets of smart cards*. Bd. 31. Springer Science & Business Media, 2008.

[NL15] Y. Nir und A. Langley. *ChaCha20 and Poly1305 for IETF Protocols*. 2015. URL: https://www.rfc-editor.org/info/rfc7539 (besucht am 05. 02. 2023).

[Soc+17] Petr Socha u. a. "Optimization of Pearson correlation coefficient calculation for DPA and comparison of different approaches". In: *2017 IEEE 20th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*. IEEE. 2017, S. 184–189.

[Vig16] Sebastiano Vigna. *PRNG: XorShift128+*. 2016. URL: https://xoshiro.di.unimi.it/xorshift128plus.c (besucht am 08. 02. 2023).

[Wu+08] Keke Wu u. a. "Correlation power analysis attack against synchronous stream ciphers". In: *2008 The 9th International Conference for Young Computer Scientists*. IEEE. 2008, S. 2067–2072.