Effiziente Auslagerung der IPsec-Verschlüsselung auf Grafikkarten

Georg Schönberger, BSc

MASTERARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Sichere Informationssysteme

in Hagenberg

im Mai 2011

Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und die aus anderen Quellen entnommenen Stellen als solche gekennzeichnet habe.

Hagenberg, am 26. Mai 2011

Georg Schönberger, BSc

Inhaltsverzeichnis

Kurzfassung

Virtuelle private Netzwerke (VPNs) transportieren private Daten über öffentliche Netzwerke, wobei *Internet Protocol Security* (IPsec) als Set von Protokollen dazu verwendet wird, um Sicherheitsanforderungen wie Integrität, Vertraulichkeit und Authentizität zu gewährleisten. In puncto Vertraulichkeit wird hierbei vor allem auf den *Advanced Encryption Standard* (AES), einen symmetrischen Blockverschlüsselungs-Algorithmus, gesetzt.

Werden die im Netzwerkbereich geforderten Datenraten von 10 Gigabit/s betrachtet, so stellen diese Transferraten beträchtliche Anforderungen an die Prozessoren eines Systems. Maßnahmen, zur Beschleunigung und zur gleichzeitigen Entlastung der Prozessoren, sind einerseits die Verwendung von neuen kryptografischen Algorithmen und andererseits die Auslagerung der Berechnungen auf spezielle Coprozessoren.

Hinsichtlich neuer Algorithmen zeichnet sich z. B. der AES im Galois/-Counter Mode (AES-GCM) dadurch aus, dass er Vertraulichkeit, Authentizität und Integrität gewährleisten kann und als „Combined Cipher“ extra durchzuführende Hash-Verfahren überflüssig macht. Im Hinblick auf Coprozessoren ist die Berechnung der kryptografischen Operationen unter Zuhilfenahme von Grafikkarten (GPUs) eine Möglichkeit, die Performance zu steigern. Deren Attraktivität ist vor allem mit der Entwicklung der Nvidia CUDA-Plattform gestiegen.

In dieser Arbeit übernimmt eine GPU die benötigte AES-Verschlüsselung bei der Umsetzung von AES-GCM. Die Verschlüsselung wird jedoch nicht nur ausgelagert, sondern auch im Vorhinein berechnet. Eine Controller-Komponente auf der CPU vervollständigt den AES-GCM und verwaltet parallel dazu die GPU sowie deren Ergebnisse in Double-Buffer. Ein auf Performance abgestimmtes Zusammenspiel zwischen der GPU und dem Controller am Host-System hat hierbei oberste Priorität.

Zur Demonstration der Funktionalitäten und der Performance des Systems werden eigene Applikationen mit virtuellen Netzwerk-Devices und einem Patch der Bibliothek „LibTomCrypt“ realisiert. Dabei werden die Anforderungen an Komponenten für IPsec analysiert und die rechenintensiven Operationen eruiert.

Abstract

A virtual private network (VPN) is a secure way to transport private data over a public network. Internet Protocol Security (IPsec) is a package of protocols to create VPNs and ensure confidentiality, authentication and integrity. Concerning confidentiality the Advanced Encryption Standard (AES) has become one of the favorite encryption schemes. As the most compute intensive parts of network applications with Gigabit/s throughput often belong to cryptographic algorithms—such as AES—new ways for acceleration are needed. Some possibilities to accelerate applications can be individual coprocessors or the usage of novel algorithms.

In terms of AES the Galois/Counter Mode (AES-GCM) has the potential to enhance an application's performance as it is a so called "combined cipher". AES-GCM assures confidentiality, authentication, and integrity and does not need extra hashing or message authentication codes. Concerning coprocessors graphics processing units (GPUs) are widely spread today. Especially with the "Compute Unified Device Architecture" (CUDA) and the programming language "C for CUDA" GPUs gained a boost in popularity.

In this master thesis the AES part of AES-GCM is precomputed on GPUs whereas the results can later on be used to en- or decrypt information. Only the authentication part is left to be computed on the Central Processing Unit (CPU). A controller with a smart devised design is responsible for assuring maximum performance and a smooth cooperation of GPU and CPU. This is realized through a double buffer system where the GPU has the job to refill empty buffers with encrypted counters while the controller can continue processing information with AES-GCM.

For testing functionality of controller and GPU an application using virtual network devices is created. In order to evaluate the performance a patch of the library "LibTomCrypt" is implemented. In the context of these two applications the requirements for components realizing IPsec are analyzed as well as the most compute intensive parts are revealed.

Kapitel 1

Einleitung

1.1 Problemstellung

Die Paketverarbeitungs- und Kryptografie-Routinen von IPsec-Verbindungen, bei Geschwindigkeiten im Gigabit/s-Bereich, stellen rechenintensive Anforderungen an heutige Prozessoren. Die Tatsache, dass kryptografische Algorithmen auf Pakete von mehreren Verbindungen zugleich angewendet werden müssen, erfordert neue Wege für die Erfüllung der gewünschten Datenraten. Dabei liegt der Schwerpunkt nicht nur auf der Entwicklung von Hardware-Beschleunigungskarten oder hardware-optimierten Implementierungen, sondern auch auf der Evaluierung von neuen kryptografischen Algorithmen.

In Bezug auf neue Coprozessoren haben sich in den letzten Jahren Grafikkarten als preisgünstige und effiziente Lösung etabliert. Mit der Entwicklung der „*Compute Unified Device Architecture*“ (CUDA) durch Nvidia existiert außerdem ein einfaches aber auch umfassendes Werkzeug für die Programmierung von Grafikkarten. Da deren Performance jedoch stark von der Art des zu lösenden Problems abhängig ist, kann nicht verallgemeinert werden, dass sie sich für die Beschleunigung von kryptografischen Algorithmen eignen. Es ist daher die Frage zu klären, in welchem Ausmaß und zu welchem Zweck Grafikkarten im Kontext von IPsec-Verbindungen eingesetzt und mit CUDA integriert werden können.

1.2 Zielsetzung

Diese Arbeit hat sich die Analyse neuer Wege zur Beschleunigung von IPsec-Verbindungen zum Ziel gemacht. Diese Analyse inkludiert das Verständnis der Verwendung von AES im Galois/Counter Mode für IPsec, die grundsätzlichen Aspekte bei der Programmierung von Applikationen für Grafikkarten sowie die auftretenden Probleme bei der Konzeption von Architekturen im Gigabit/s-Netzwerkbereich. Wichtige Themen sind unter anderem

ein ganzheitlicher Überblick über den Einsatz des Krypto-Algorithmus (z. B. bezüglich Parametrisierung), die Auslagerung von geeigneten Teilen auf die Grafikkarte, die Realisierung eines effizienten Zusammenspiels von CPU und Grafikkarte oder auch die Analyse der Performance des entwickelten Systems.

1.3 Aufbau der Arbeit

Zu Beginn dieser Arbeit werden zunächst die theoretischen Aspekte des AES im Galois/Counter Mode erläutert. Darauf folgt eine Einleitung in die CUDA-Programmierung sowie die Erklärung der Umsetzung des AES-Algorithmus. Im nächsten Schritt wird detailliert beschrieben, wie der AES im Galois/Counter Mode in IPsec für die Ver- und Entschlüsselung von Daten eingesetzt wird. Im Zuge dieses Abschnitts werden auch die Änderungen der Standards vorgestellt, die nötig sind, um verschlüsselte Counter vorausberechnen zu können. Nach dem Abschluss der theoretischen Grundlagen wird näher auf die praktische Implementierung eingegangen. In diesem Zusammenhang wird geschildert, wie ein GPU- und ein CPU-Modul den AES im Galois/Counter Mode gemeinsam umsetzen, wobei die Grafikkarte für die Verschlüsselung eingesetzt wird. Bei der näheren Darstellung des CPU-Moduls wird auch gezeigt, wie mit Hilfe eines Patches der LibTomCrypt-Bibliothek der kryptografische Algorithmus vervollständigt wird. Als ein weiterer wichtiger Teil folgt eine Analyse der Performance der umgesetzten Applikation. Abschließend wird ein Resümee gezogen und auf weiterführende Arbeiten und Themen eingegangen.

Kapitel 2

AES-GCM und CUDA

Dieses Kapitel bietet einen Überblick über den *Advanced Encryption Standard* (AES) und seine Verwendung im Galois/Counter Modus (GCM). Da dieser Modus Teil der späteren Implementierung ist, werden im Detail seine Abläufe und Komponenten erläutert. Der zweite Teil von Kapitel 2 geht näher auf die CUDA-Architektur ein und schildert die Grund-Techniken und -Begriffe im Zusammenhang mit der Umsetzung eines AES mit CUDA. In diesem Zusammenhang ist es das Ziel die wichtigsten Begriffe und Abläufe kennen zu lernen, um die in den darauf folgenden Kapiteln umgesetzte Architektur zu verstehen.

2.1 AES

Der AES wurde vom *National Institute of Standards and Technology*[1] als Standard FIPS PUB 197 [25] veröffentlicht und basiert auf dem von Joan Daemen und Vincent Rijmen eingereichten Algorithmus „Rijndael“ [5]. Im folgenden Abschnitt sind nur die wichtigsten Begriffe im Zusammenhang mit AES enthalten. Für eine genaue und vollständige Analyse des Algorithmus empfehlen sich zum Beispiel [25], [5] oder [4].

AES ist ein symmetrischer Blockverschlüsselungs-Algorithmus – im Folgenden auch als „Block-Cipher“ bezeichnet – zur Ver- und Entschlüsselung von Informationen [25, S. 1]. Er ermöglicht die Verwendung von verschiedenen Schlüssellängen (128, 192 und 256 Bit) bei einer gleich bleibenden Blocklänge von 128 Bit [5, S. 8].

Der Algorithmus arbeitet einen Block des Klartextes (Plaintext) bzw. der verschlüsselten Information (Ciphertext) in mehreren Runden ab. Die Anzahl der Runden ist dabei je nach Schlüssellänge verschieden. Tabelle 2.1 zeigt die Abhängigkeit der Rundenanzahl als Funktion von Schlüssellänge und Blocklänge (Nk und Nb geben die Anzahl an 32-Bit-Wörtern an).

[1] http://www.nist.gov

	Schlüssellänge (Nk)	Blocklänge (Nb)	Runden (Nr)
AES-128	4	4	10
AES-192	6	4	12
AES-256	8	4	14

Tabelle 2.1: Anzahl der AES-Runden abhängig von Schlüssel- und Blocklänge (aus [25, S. 14]).

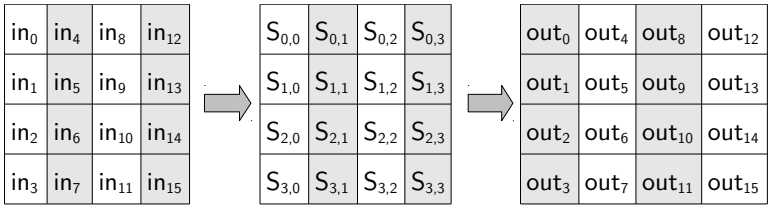

Abbildung 2.1: Mapping des Input-Blocks in das State Array (aus [25, S. 9]).

Jeweils für den Verschlüsselungs-Algorithmus (Cipher) und den Entschlüsselungs-Algorithmus (Inverse Cipher) verwendet AES vier verschiedene, byte-orientierte Transformationen:

1. Byte-Substitution mit einer S-Box – `SubBytes`,
2. Verschiebung der Zeilen mit verschiedenen Offsets – `ShiftRows`,
3. Durchmischung der Spalten – `MixColumns` und
4. Hinzufügen eines Rundenschlüssels – `AddRoundKey`.

Diese Runden-Funktionen werden auf eine interne Struktur (State Array bzw. State) angewendet. Der State setzt sich aus einer 4×4-Matrix zusammen, deren Werte aus den 16 Byte eines Input-Blocks bestehen. Abbildung 2.1 stellt dar, wie die Bytes des Input-Blocks den State befüllen. Für `AddRoundKey` werden außerdem Rundenschlüssel benötigt, die aus dem geheimen Schlüssel (Secret Key) erzeugt werden – auch bekannt als „`Key Expansion`“. Für eine detaillierte Beschreibung der soeben genannten Runden-Funktionen wird auf [25, Abschnitt 5.1 und 5.2] verwiesen.

2.2 AES im Galois/Counter Mode

2.2.1 Einführung

Der AES-Algorithmus kann in verschiedenen Modi betrieben werden. Grundsätzlich sind die Modi unabhängig von den darunter liegenden symmetrischen Block-Ciphern, dieses Dokument bezieht sich aber stets auf die Verwendung von AES.

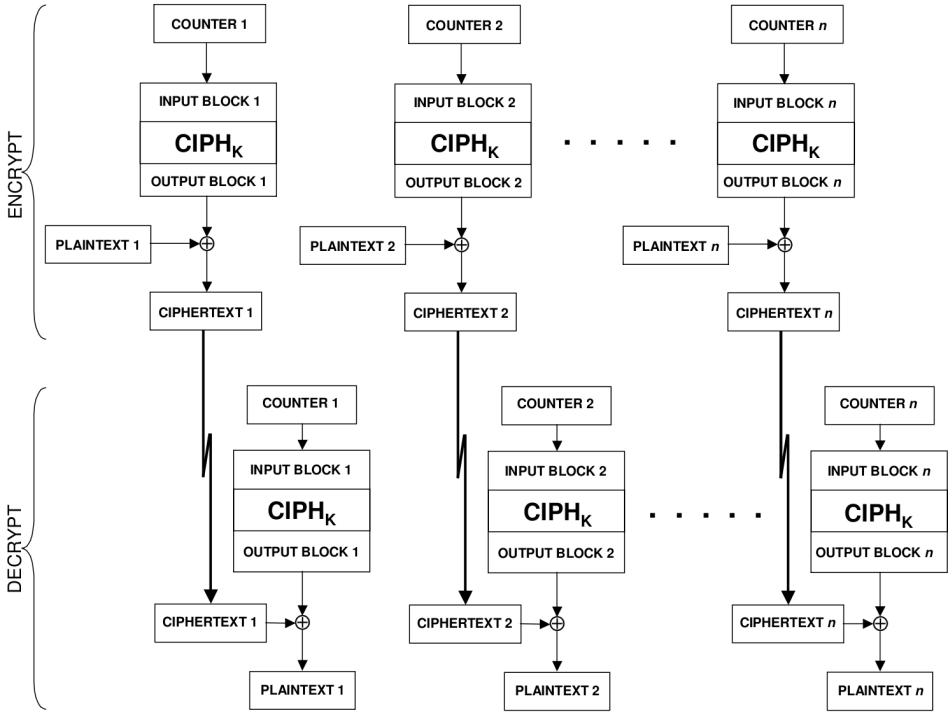

Abbildung 2.2: Verwendung des CTR-Modus, aus [9, S. 16].

Der Galois/Counter Mode (GCM) ist ein Operations-Modus, der nicht nur Vertraulichkeit, sondern auch Authentizität und Integrität für Nachrichten gewährleisten kann. In diesem Zusammenhang wird auch oft von Authenticated En- bzw. Decryption gesprochen. Idealerweise besitzt AES eine Blocklänge von 128 Bit, da diese für die Verwendung von GCM als Modus benötigt wird [10, S. 1]. Von nun an wird einzig und allein der GCM mit AES als Block-Cipher beschrieben, was des Weiteren auch als AES-GCM bezeichnet wird.

AES-GCM verwendet eine Variation des Counter-Modus (CTR-Modus). Hierbei bilden fortlaufende Zähler (sogenannte „Counter“) die zu verschlüsselnden Input-Blöcke, um dadurch eine Sequenz von Output-Blöcken zu erzeugen, die mit dem Plain- bzw. Ciphertext exklusiv-oder-verknüpft (XOR) werden [9, S. 15]. Die verwendeten Counter müssen für einen gegebenen Secret-Key einzigartig sein, d. h. sie dürfen sich für denselben Key nicht wiederholen, da ansonsten derselbe Plaintext den gleichen Ciphertext ergibt. Abbildung 2.2 zeigt, wie Counter verschlüsselt und mit dem Cipher- bzw. Plaintext XOR gerechnet werden und somit Ver- und Entschlüsselung umsetzen. Ein Nachteil des CTR-Modus ist, dass er alleine nur die Vertraulichkeit der verschlüsselten Daten sicherstellt. Für die Gewährleistung von Authentizität und Integrität müssten nachträglich Hash-Funktionen bzw.

Hash-based Message Authentication Codes (HMACs)[2] angewandt werden.

Diese nachträgliche Anwendung wird beim AES-GCM nicht benötigt, da bereits eine universelle Hash-Funktion (GHASH, s. auch Abschn. 2.2.2) im Algorithmus integriert ist. GHASH arbeitet mit einem binären Galois-Feld, zusammen mit dem CTR-Modus ergibt sich daher der Name des Algorithmus „Galois/Counter Modus“. Durch die Kombination der oben genannten Techniken ergeben sich bei einem Einsatz von AES-GCM für eine Applikation folgende Vorteile [23, S. 1]:

1. Der CTR-Modus erlaubt Pipelining bzw. Parallelisierung bei der Verarbeitung von Input-Blöcken und ermöglicht daher hohe Datenraten.
2. Ist bekannt wie der Initialisierungsvektor aussehen wird und ist die Länge der Nachricht bekannt, so können die verschlüsselten Counter vorausberechnet werden. Diese Eigenschaft ist vor allem für die Erreichung von hohen Durchsatz-Raten interessant, da für die rechenintensivste Operation – die Verschlüsselung – Ergebnisse vorausberechnet werden können.
3. Die benötigten Initialisierungsvektoren können von beliebiger Länge sein und müssen nicht der Block-Länge des darunter liegenden Block-Ciphers entsprechen.
4. AES-GCM benötigt nicht die inversen Funktionen (Inverse Cipher) des darunter liegenden Block-Ciphers.
5. Durch GHASH wird die Authentizität und Integrität der Nachrichten garantiert. Dazu können außerdem weitere Daten in die Authentifizierung mit einfließen, die nicht verschlüsselt werden. Sind diese zusätzlichen, nicht verschlüsselten Authentifizierungsdaten schon vor der eigentlichen Verschlüsselung vorhanden, kann der GCM-Authentifizierungsteil vorausberechnet werden.
6. Die Multiplikation im binären Galois-Feld, die bei GHASH benötigt wird, lässt sich einfach und effizient implementieren.
7. AES-GCM kann auch als alleinstehende MAC-Funktion benutzt werden, wenn keine Daten verschlüsselt werden. Diese Funktion wird auch als GMAC bezeichnet.

2.2.2 Mathematische Komponenten

Die folgenden elementaren Teile werden für die detaillierte Beschreibung des AES-GCM benötigt. Vor allem die genaue Schreibweise von Bit-Operationen und Längen-Angaben ist bei der Erklärung des Algorithmus wichtig.

[2] Für eine Erklärung von Hash-Funktionen und HMACs kann [24, Kap. 9] zur Hilfe genommen werden.

Bitfolgen

Darstellung: Die Bits „0“ und „1“ werden durch eine Typewriter-Formatierung hervorgehoben. Des Weiteren bedeutet der Ausdruck $\mathtt{0}^s$ für eine Bitfolge, dass sie aus s Nullen besteht. Dazu ein Beispiel bei dem die Bitfolge sich aus acht Nullen zusammensetzt:

$$\mathtt{0}^8 = \mathtt{00000000}$$

Konkatenation: Das Aneinanderreihen zweier Bitfolgen wird gekennzeichnet durch $\|$, z. B. $\mathtt{00} \parallel \mathtt{11} = \mathtt{0011}$.

Länge: Die Länge einer Bitfolge X wird angegeben durch $\mathrm{len}(X)$:

$$\mathrm{len}(\mathtt{00110}) = 5$$

Für eine gegebene Zahl s bedeutet der Ausdruck $[\mathrm{len}(X)]_s$ die s Bit lange Binärdarstellung von $\mathrm{len}(X)$.

MSB und LSB: Die Funktionen MSB_s und LSB_s geben für eine positive ganze Zahl s, die s höchstwertigen (most significant) und die s niedrigstwertigen (least significant) Bits zurück:

$$\mathrm{MSB}_3(\mathtt{00110}) = \mathtt{001}$$
$$\mathrm{LSB}_2(\mathtt{00110}) = \mathtt{10}$$

Inkrement: Die Funktion $\mathrm{inc}_s(X)$ erhöht für eine gegebene positive Zahl s, die dezimale Repräsentation der s niedrigstwertigen Bits von X, modulo 2^s. Die höchstwertigen $\mathrm{len}(X) - s$ Bits von X bleiben unverändert.

Operationen in $GF(2^{128})$

Die folgenden Ausdrücke beschreiben die Addition und die Multiplikation zweier Elemente in einem Galois-Feld. Wie die Algorithmen aussehen, die die Addition und Multiplikation für Elemente des Feldes in Form von Bitfolgen umsetzen, kann z. B. in [23, Kap. 3] oder [10, Abschn. 6.3] nachgelesen werden. Als elementare Operationen ist es für die Performance des Algorithmus wichtig, dass sie einfach und effizient zu implementieren sind.

Addition: Die Addition von zwei Elementen $X, Y \in GF(2^{128})$ notiert der Ausdruck $X \oplus Y$.

Multiplikation: Das Multiplizieren von zwei Elementen $X, Y \in GF(2^{128})$ $X \bullet Y$.

GHASH

Der Authentifizierungsmechanismus des GCM basiert auf der Hash-Funktion GHASH (vgl. Abb. 2.3). GHASH multipliziert die gegebenen Daten mit einem fixen Parameter – dem „Hash-Subkey“ (H) – in $GF(2^{128})$ ([22, App. B]

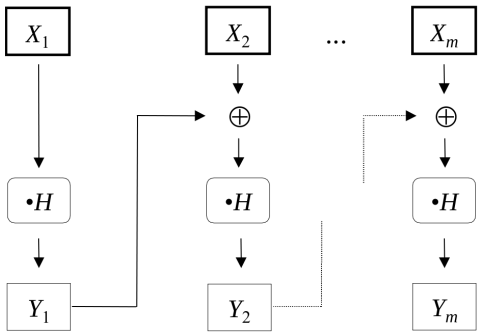

Abbildung 2.3: $\text{GHASH}_H(X_1 \parallel X_2 \parallel \ldots \parallel X_m) = Y_m$ [10, S. 13].

bzw. [23, S. 8]). Der Hash-Subkey entsteht durch die Anwendung des Block-Ciphers auf eine Null-Bitfolge: $H = E(K, 0^{128})$ (s. auch [10, S. 10] und [23, S. 5]).

Voraussetzungen: H (Hash-Subkey).

Input: Bitfolge X mit $\text{len}(X) = 128 * m$, wobei m eine positive, ganze Zahl ist.

Schritte: $(((X_1 \bullet H \oplus X_2) \bullet H \oplus X_3) \bullet H \ldots \oplus X_m) \bullet H$

GCTR

Die GCTR-Funktion lehnt sich, wie schon in Abschnitt 2.2 erwähnt, an den normalen CTR-Modus an.

Voraussetzungen: Block-Cipher CIPH – 128 Bit Blocklänge; Key K.

Input: Initialisierungsvektor (ICB); Bitfolge X von beliebiger Länge.

Output: Bitfolge Y mit $\text{len}(Y) = \text{len}(X)$.

Schritte: Abb. 2.4 stellt die Operationen grafisch dar. In Schritt 1 wird der Input in die größtmögliche Anzahl an 128-Bit-Blöcken aufgeteilt, sodass nur der letzte Bit-Block kein vollständiger Block ist. Dann wird der ICB inkrementiert, um eine Folge von Counter-Blöcken zu generieren. Daraufhin wird auf diese Counter-Blöcke der Block-Cipher angewandt und die Ergebnisse mit den Input-Blöcken XOR-verknüpft. Abschließend bilden die Konkatenation dieser Blöcke die Output-Blöcke.

2.2.3 Ver- und Entschlüsselung

Authenticated Encryption

Die Authenticated-Encryption-Funktion verschlüsselt die vertraulichen Daten und berechnet einen Authentication Tag. Für die Berechnung des Tags

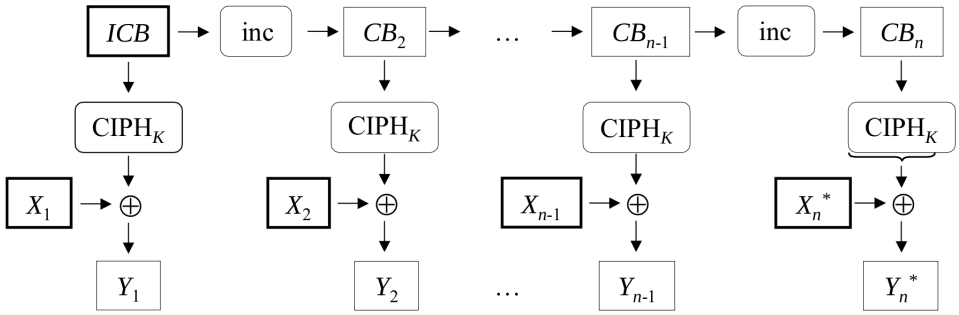

Abbildung 2.4: $\mathrm{GCTR}_K(\mathrm{ICB}, X_1 \parallel X_2 \parallel \ldots \parallel X_n^*) = Y_1 \parallel Y_2 \parallel \ldots \parallel Y_n^*$ [10, S. 14].

können zusätzliche, nicht zu verschlüsselnde Daten, hinzugefügt werden (Additional Authenticated Data (AAD)). Der Tag garantiert daraufhin die Authentizität und Integrität für die verschlüsselten Daten als auch für die AAD.

Voraussetzungen: Block-Cipher CIPH mit 128 Bit Blocklänge; Key K; Unterstützte Input- und Output-Längen; Unterstützte Tag-Längen t.

Input: Initialisierungsvektor (IV); Klartext (P); AAD (A).

Output: Ciphertext (C); Authentication Tag (T).

Schritte: Abb. 2.5 stellt die nachfolgend beschriebenen Schritte grafisch dar.

1. $H = \mathrm{CIPH}_K(0^{128})$.
2. Wenn $\mathrm{len(IV)} = 96$, dann bilde $J_0 = \mathrm{IV} \parallel \mathtt{0}^{31} \parallel \mathtt{1}$. Ist $\mathrm{len(IV)} \neq 96$, dann ist $J_0 = \mathrm{GHASH}_H(\mathrm{IV} \parallel \mathtt{0}^{s+64} \parallel [\mathrm{len(IV)}]_{64})$ und s jene Anzahl an Bits, sodass die Bitlänge von $\mathrm{IV} \parallel \mathtt{0}^s$ ein Vielfaches von 128 Bit ist.
3. $C = \mathrm{GCTR}_K(\mathrm{inc}_{32}(J_0), P)$.
4. u ist jene Anzahl an Bits, sodass die Bitlänge von $C \parallel \mathtt{0}^u$ ein Vielfaches von 128 Bit ist. v ist jene Anzahl an Bits, sodass die Bitlänge von $A \parallel \mathtt{0}^v$ ein Vielfaches von 128 Bit ist.
5. Bilde den Block S durch

$$S = \mathrm{GHASH}_H(A \parallel \mathtt{0}^v \parallel C \parallel \mathtt{0}^u \parallel [\mathrm{len}(A)]_{64} \parallel [\mathrm{len}(C)]_{64})$$

6. $T = \mathrm{MSB}_t(\mathrm{GCTR}_K(J_0, S))$.
7. Rückgabewert ist (C, T).

Alternativ zeigt Abbildung 2.6 wie der Verschlüsselungs-Ablauf für zwei Datenblöcke und einen Block AAD aussieht. In dieser Abbildung wird der IV als „Counter 0“ bezeichnet, „mult_H“ steht für die Multiplikation mit H in $GF(2^{128})$ und „Auth Data 1“ für die AAD.

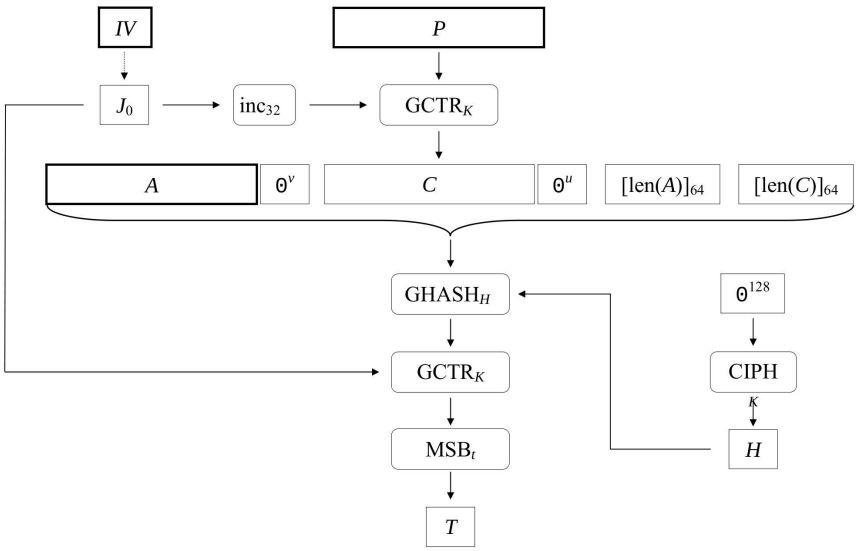

Abbildung 2.5: Authenticated Encryption mit den Parametern IV, P, A und C, T als Output [10, S. 16].

Authenticated Decryption

Die Authenticated-Decryption-Funktion entschlüsselt den Ciphertext und berechnet für diesen, als auch für die vorhandenen AAD, den Authentication Tag. Des Weiteren wird der errechnete mit dem gegebenen Authentication Tag verglichen und so die Authentizität bzw. Integrität überprüft.

Voraussetzungen: Block-Cipher CIPH mit 128 Bit Blocklänge; Key K; Unterstützte Input- und Outputlängen; Unterstützte Tag-Längen t.

Input: Initialisierungsvektor (IV); Ciphertext (C); AAD (A); Authentication Tag (T).

Output: Klartext (P) oder der Hinweis, dass die Überprüfung mit T fehlgeschlagen ist (`FAIL`).

Schritte: Abb. 2.7 stellt die nachfolgend beschriebenen Schritte grafisch dar.

1. Werden die Längen von IV, A oder C nicht unterstützt bzw. ist $\text{len}(T) \neq t$, dann wird `FAIL` zurückgegeben.
2. $H = \text{CIPH}_K(0^{128})$.
3. Wenn $\text{len}(\text{IV}) = 96$, dann $J_0 = \text{IV} \parallel 0^{31} \parallel 1$. Ist $\text{len}(\text{IV}) \neq 96$, dann ist $J_0 = \text{GHASH}_H(\text{IV} \parallel 0^{s+64} \parallel [\text{len}(\text{IV})]_{64})$ und s jene Anzahl an Bits, sodass die Bitlänge von IV $\parallel 0^s$ ein Vielfaches von 128 Bit ist.
4. $P = \text{GCTR}_K(\text{inc}_{32}(J_0), C)$.

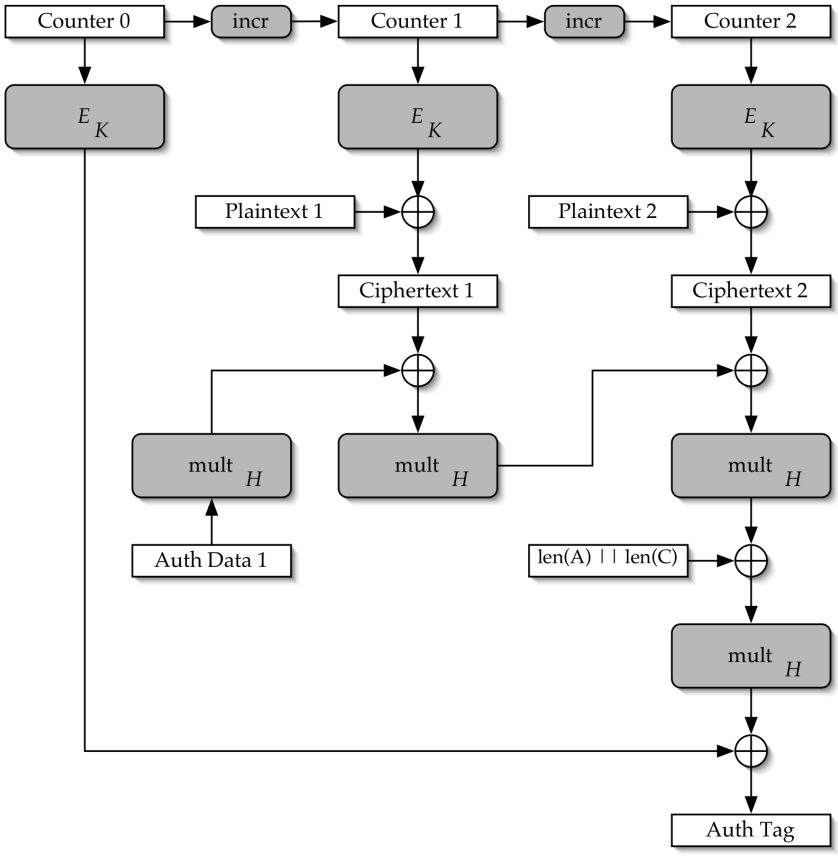

Abbildung 2.6: Authenticated Encryption mit zwei Datenblöcken und einem Block AAD [23, S. 6].

5. u ist jene Anzahl an Bits, sodass die Bitlänge von $C \parallel \mathtt{0}^u$ ein Vielfaches von 128 Bit ist. v ist jene Anzahl an Bits, sodass die Bitlänge von $A \parallel \mathtt{0}^v$ ein Vielfaches von 128 Bit ist.
6. Bilde den Block S durch

$$S = \mathrm{GHASH}_H(A \parallel \mathtt{0}^v \parallel C \parallel \mathtt{0}^u \parallel [\mathrm{len}(A)]_{64} \parallel [\mathrm{len}(C)]_{64})$$

7. $T' = \mathrm{MSB}_t(\mathrm{GCTR}_K(J_0, S))$.
8. Wenn $T' = T$, dann gib P zurück, sonst `FAIL`.

Nach dem soeben abgehandelten Abschnitt ist nun verständlich, welche Komponenten bei AES-GCM wie zusammenspielen, um Daten zu ver- und entschlüsseln und Authentication Tags zu berechnen.

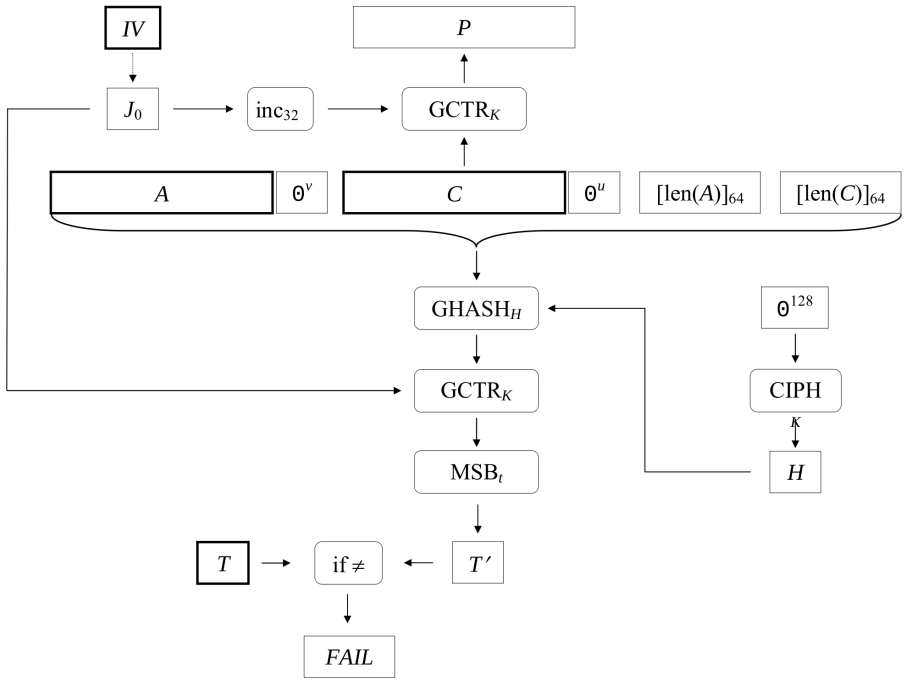

Abbildung 2.7: Authenticated Decryption mit den Parametern IV, C, A und P oder `FAIL` als Output [10, S. 18].

2.3 CUDA

Die „*Compute Unified Device Architecture*“ (CUDA)[3] ist eine von Nvidia erzeugte Architektur für die Entwicklung von General Purpose Applications auf Grafikkarten (GPUs oder im Folgenden auch als Devices bezeichnet) [18, S. 7]. Die Entwicklung der Architektur beinhaltete nicht nur das Hervorbringen von Software-Werkzeugen für die Programmierung, sondern auch entscheidende Änderungen der GPU-Hardware. Seit dem G80-Chip[4] ist es erstmals möglich, ohne Benutzung der Grafik-Interfaces, Programme auf einer GPU auszuführen. Bis zu diesem Zeitpunkt mussten Programmierer Wege finden, ihre Berechnungen mittels grafische Operationen umzusetzen, um die Rechenkapazitäten der GPUs nutzen zu können. Dies schloss auch die Programmierung mit Programmierschnittstellen (APIs) wie OpenGL oder DirectX mit ein, deren Komplexität und Beschränktheit für General Purpose Programming durch CUDA kompensiert wurde.

Die CUDA-Softwarewerkzeuge basieren auf dem C-Standard [1], somit können GPU-Applikationen mit Grundkenntnissen in C und einer Hand voll C-Erweiterungen – für den Device-Code – implementiert werden. Dazu

[3] http://developer.nvidia.com/category/zone/cuda-zone.
[4] http://www.nvidia.com/page/8800_tech_briefs.html.

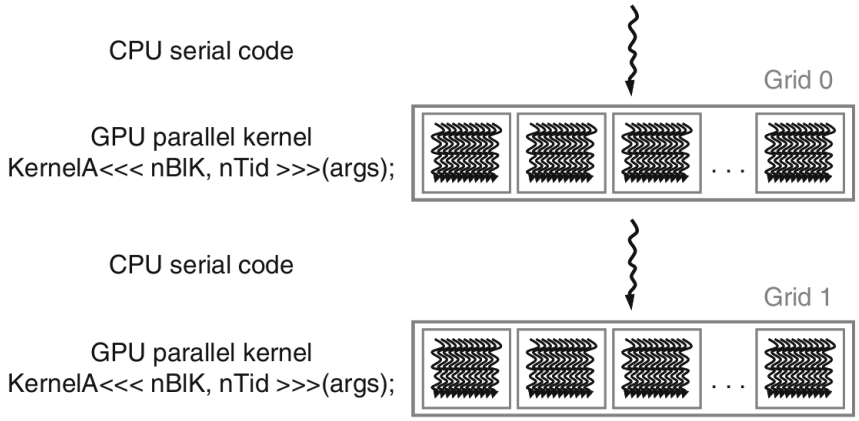

Abbildung 2.8: Ablauf eines CUDA-Programms [18, S. 42].

werden APIs für das Management von Devices, zum Starten von Kerneln und auch zur Verwaltung von GPU-Speicher zur Verfügung gestellt. Alle Quelldateien, die die spezifischen CUDA-Erweiterungen beinhalten, werden mit dem eigens zur Verfügung gestellten Compiler „nvcc“ kompiliert [27, Kap. 3]. Dieser übersetzt den Device-Code und übergibt den Host-Code an den systemspezifischen Compiler (unter Linux z. B. der gcc[5]).

In den nächsten beiden Abschnitten werden nur die wichtigsten Begriffe bezüglich CUDA erläutert, die zum besseren Verständnis der weiteren Kapitel dienen. Als Literatur für ein tieferes Verständnis der Materie empfiehlt sich vor allem „*Programming Massively Parallel Processors: A Hands-On Approach*“ von David Kirk und Wen-Mei W. Hwu, „*CUDA by Example: An Introduction to General-Purpose GPU Programming*“ von Jason Sanders und Edward Kandrot oder auch die Webseite http://developer.nvidia.com/object/gpucomputing.html.

CUDA-Programmstruktur

Dem CUDA-Programmierer stehen bei der Entwicklung von Applikationen üblicherweise ein oder mehrere CPUs und GPUs zur Verfügung. Dabei stellt sich die Frage, welche Teile von der CPU und welche von der GPU ausgeführt werden. Für die GPU eignen sich jene Abschnitte, die ein hohes Maß an Daten-Parallelität (Eng.: *Data Parallelism*) aufweisen [18, Abschn. 3.1]. In diesen Abschnitten können viele arithmetische Operationen auf unterschiedliche Datensätze zugleich ausgeführt werden, was des Öfteren auch als *Single-Program, Multiple-Data* bezeichnet wird [18, S. 51] . Zusammengefasst lässt sich folgender Ablauf verallgemeinern (s. auch Abb. 2.8):

1. Serieller Code wird auf der CPU ausgeführt – z. B. normaler C-Code.

[5] http://gcc.gnu.org/.

2. Für die parallelen Abschnitte werden sogenannte „Kernels“ auf der GPU gestartet. Vor dem Start eines Kernels legt der Programmierer die Anzahl der Threads fest, die denselben Kernel parallel ausführen.
3. Nach dem Kernel-Start werden die Threads parallel von der GPU abgearbeitet.
4. Haben alle Threads eines Kernels terminiert, führt die CPU die Abarbeitung fort.

CUDA-Thread-Hierarchie

Die Gesamtheit an Threads, die einen Kernel ausführen, wird als Grid bezeichnet. Dieses Grid kann aus mehreren Tausend bis zu Millionen Threads bestehen, die ihre Tätigkeit parallel auf verschiedene Daten ausüben [27, Kap. 2]. Die Anzahl der zu startenden Threads wird dabei beim Kernel-Start festgelegt (aus [27, S. 8]):

```
1  // Kernel definition
2  __global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N]){
3    int i = threadIdx.x;
4    int j = threadIdx.y;
5    C[i][j] = A[i][j] + B[i][j];
6  }
7  int main(){
8  ...
9    // Kernel invocation with one block of N * N * 1 threads
10   int numBlocks = 1;
11   dim3 threadsPerBlock(N, N);
12   MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
13 }
```

In diesem Code-Beispiel wird ein Block mit $N \times N$ Threads gestartet. Hierbei lässt sich erkennen, dass ein Grid aus einer räumlich angeordneten Menge von Threads und Blöcken bestehen kann (Abb. 2.9). Blöcke werden dabei als eine Gruppierung von Threads aufgefasst, die bis zu drei Dimensionen besitzen können. Ein Grid setzt sich aus mehreren Blöcken zusammen, die maximal zwei Dimensionen aufweisen können.

Um Threads und Blöcke untereinander unterscheiden zu können, stehen zur Laufzeit System-Variablen zur Verfügung . Zum Beispiel besitzt bei einem zweidimensionalen Block jeder Thread die Variablen `threadIdx.x` und `threadIdx.y`. Wurde dazu das Grid mit Blöcken einer Dimension erzeugt, so kann auf den Block-Index mittels `blockIdx.x` zugegriffen werden. Die Anzahl der Threads pro Block kann über die Variable `blockDim` herausgefunden werden. Die maximale mögliche Anzahl an Blöcken und Threads ist von GPU zu GPU unterschiedlich und muss jeweils spezifisch eruiert werden.

Diese Unterscheidung der Threads untereinander als auch deren Gruppierung in Blöcken wird auch dazu genutzt, um die Threads auf verschiedenen Datensätzen operieren zu lassen. Angenommen die GPU hätte die

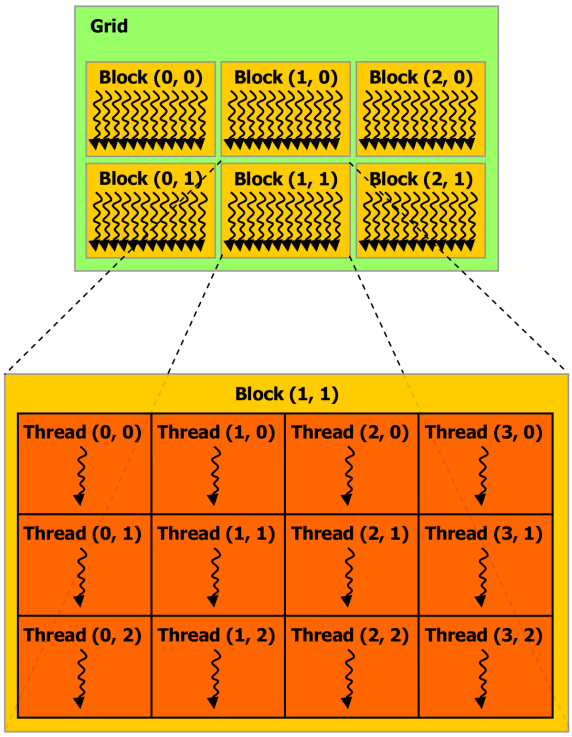

Abbildung 2.9: Aufbau eines Grids aus Blöcken und Threads [27, S. 9].

Aufgabe zwei Vektoren mit 4096 Komponenten zu addieren. Für eine parallele Addition könnten nun 4096 Threads gestartet werden, die jeweils ein Element der Vektoren aufsummieren. Mit der Annahme, dass die maximale Anzahl an Threads pro Block z.B. 256 Threads wäre, müssten 16 Blöcke à 256 Threads gestartet werden, um alle Elemente abzudecken. Dann wäre z.B. Thread 56 aus Block 0 für Element $0*256+56 = 56$ zuständig. Thread 3 aus Block 3 würde die Elemente $3*256+3 = 771$ addieren. Zusammengefasst ergibt sich aus der Hierarchie an Blöcken und Threads eine Möglichkeit, um parallel die gleichen Instruktionen auf unterschiedliche Daten anzuwenden. Eine Kommunikation unter Threads und Blöcken findet über die Speicher der GPU statt, die im nächsten Abschnitt näher erläutert werden.

CUDA-Speicherverwaltung

Grundsätzlich besitzen Host und Device getrennte Speicherbereiche [18, Abschn. 3.4]. Da Kernels nur mit dem Speicher der GPU operieren können, stehen dem Programmierer verschiedene Funktionen für die GPU-Speicherverwaltung zur Verfügung:

- Speicher-Allokierung und -Deallokierung (`cudaMalloc` u. `cudaFree`).

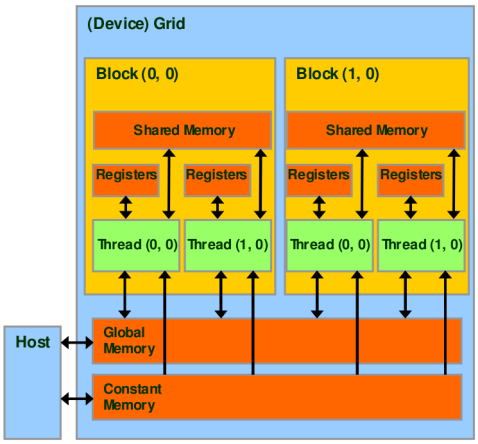

Abbildung 2.10: GPU Speicher- und Zugriffsmöglichkeiten aus der Sicht von Threads und Blöcken [18, S. 47].

- Kopieren von Speicher zwischen Host und Device (`cudaMemcpy`).

Darüber hinaus gibt es aufgrund des Hardware-Designs der GPUs verschiedene Arten von Speicher, die sich in Größe, Performance und möglichen Zugriffen von Threads und Blöcken aus unterscheiden. Abbildung 2.10 gibt einen Überblick über die unterschiedlichen Speicher, die Zugriffsmöglichkeiten ergeben sich wie folgt [18, S. 47]:

- Am Device
  - Lesen/Schreiben pro Thread *Register*
  - Lesen/Schreiben pro Thread *Local Memory*
  - Lesen/Schreiben pro Block *Shared Memory*
  - Lesen/Schreiben pro Grid *Global Memory*
  - Lesen pro Grid *Constant*, *Texture* und *Surface Memory*
- Am Host
  - Daten transferieren von/zu per Grid *Global*, *Constant*, *Texture* und *Surface Memory*

Das bedeutet Threads, Blöcke und Grids haben unterschiedliche Zugriffsrechte auf die Speicher der GPU und obendrein unterscheiden sich die einzelnen Speicher in ihrer Performance. Noch dazu sind jene Speicher, deren Performance wesentlich höher ist – wie z. B. die On-Chip Register – sehr begrenzt in ihrer Größe. Daher ist es für die Entwicklung einer CUDA-Applikation wichtig, dass die verschiedenen GPU-Speicher optimal ausgenützt werden. Doch nicht nur die Art des benutzten Speichers ist für die Performance wichtig, sondern auch, wie auf den Speicher zugegriffen wird (Stichwort „Coalesced Access“, mehr dazu in [26, Abschn. 3.2.1] oder auch

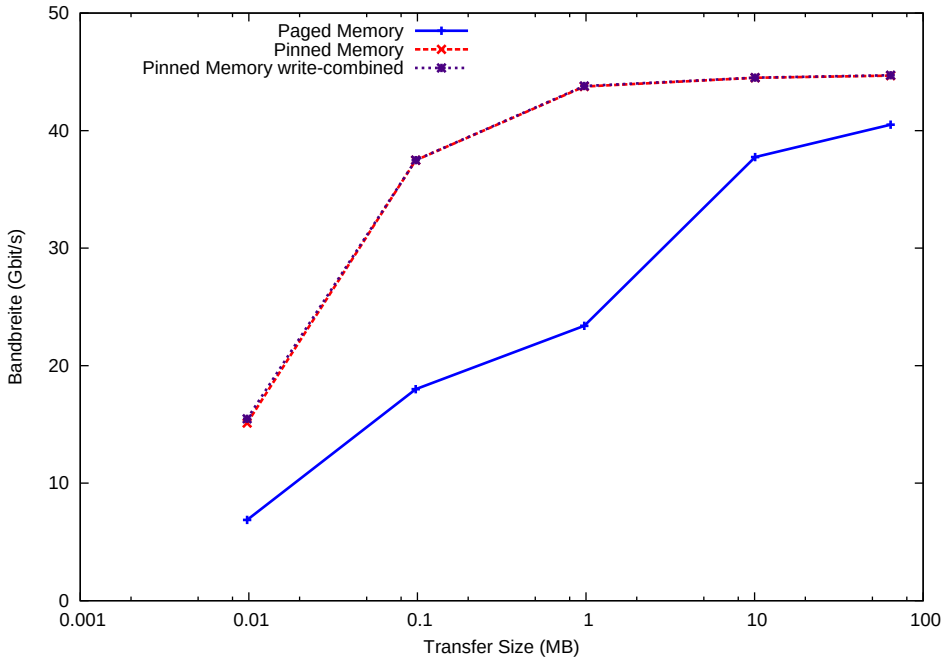

Abbildung 2.11: Performance-Unterschiede zwischen Paged, Pinned und Write-combined Memory beim Transfer von Host → Device. Die Tests wurden auf einer GTX480 (PCI Express 2.0×16) mit dem aus dem CUDA-SDK stammenden Programm „`bandwidthTest`“ durchgeführt.

[18, Abschn. 6.2]).

Auch die Zugriffe auf den globalen Speicher gehen nur sehr langsam vonstatten und sollten wenn möglich reduziert werden. Die Geschwindigkeit des globalen Speichers ist aber bei Weitem höher als die Übertragungsgeschwindigkeit vom Host zum Device. Das bedeutet, dass der Datentransfer zwischen Host und Device so minimal wie möglich zu gestalten ist [26, Kap. 3]. Dazu gehört auch, dass temporäre Datenstrukturen auf der GPU erzeugt, dort verarbeitet und nur die Ergebnisse zum Host zurück kopiert werden. Dadurch werden viele kleine Transfers zwischen Host und Device vermieden und es wird jeweils nur eine Speicher-Operation für das Hin- und eine für das Zurück-Kopieren benötigt. Mit diesem Schema wird der beim Kopieren entstehende Aufwand vermieden. Darüber hinaus gibt es Techniken und vom CUDA-Framework zur Verfügung gestellte Funktionen, mit denen sich der Datentransfer zwischen Host ↔ GPU beschleunigt:

- **Pinned Memory:** Speicher, der mittels `cudaHostAlloc` und nicht mit dem sonst üblichen `malloc` allokiert wird, wird als Pinned oder Page-locked Host Memory bezeichnet. Das Betriebssystem garantiert für Pinned Memory, dass dieser Speicherbereich nie auf die Festplat-

te ausgelagert wird und somit stetig im Hauptspeicher verweilt. Das heißt auch, dass das Betriebssystem Applikationen immer Zugriff auf diesen Speicher über die physikalische Adresse geben kann, da nie die Möglichkeit besteht, dass der Speicher ausgelagert wurde [31, Abschn. 10.2]. Aufgrund des Vorhandenseins der physikalischen Adresse kann die GPU mittels Direct Memory Access vom Host kopieren und die Geschwindigkeit wird nur mehr durch den Durchsatz des PCI-Busses beschränkt. Aus diesem Grund sind die Datentransfers zwischen Host und Device schneller als bei der Verwendung von normalem Speicher (vgl. Abb. 2.11).

Bei der Verwendung von Pinned Memory ist es außerdem möglich, dass Kernel-Ausführungen mit Speicher-Operationen überlagert werden (Concurrent Copy und Execution mit Streams, Abschn. 2.3.1). Wenn es die GPU-Baureihe unterstützt, kann Pinned Memory auch in den Device-Adressbereich „gemapped“ werden, sodass Kopiervorgänge vom CUDA-Framework übernommen und nicht mehr explizit angestoßen werden müssen [27, Abschn. 3.2.5].

Die Nachteile von zu exzessivem Gebrauch von Pinned Memory ergeben sich aus der Tatsache, dass der mit `cudaHostAlloc` allokierte Speicher nicht mehr ausgelagert werden kann. Es kann also die Situation entstehen, in der kein Speicher mehr am Host zur Verfügung steht, da von einer CUDA-Applikation zu viel physikalischer Speicher belegt wird [31, S. 187].

- **Write-combined Memory:** Pinned Memory kann überdies als write-combined gekennzeichnet werden. Für Write-combined Memory sollen Caches freigegeben werden und beim Transfer über den PCI-Bus das Snooping[6] deaktiviert werden. Der „Cuda Programming Guide“ [27, S. 37] empfiehlt Write-combined Memory nur für Speicher zu verwenden, in den der Host schreibt und von dem die GPU liest, da das Lesen vom Host Performance-Einbußen mit sich bringt. In Abb. 2.11 wird jedoch deutlich, dass die Verwendung von Write-combined Memory gegenüber Pinned Memory keine Vorteile mehr bringt.
- **Constant Memory:** Der Constant Memory ist für Threads read-only und besitzt eine beschränkte Größe von 64 KB und die Möglichkeit Caching zu verwenden. Für alle Threads eines Half-Warps[7] ist Lesen vom Constant-Cache genauso schnell wie Lesen von Registern, solange die Threads von derselben Adresse lesen [26, S. 36].
- **Texture Memory:** Der Texturen-Speicher ist ebenfalls nur read-only und besitzt auch einen Caching-Mechanismus. Dieser Cache ist optimiert für Zugriffe durch Threads, deren Anordnung nahe beisammen

[6] http://www.microsoft.com/whdc/archive/pcie_graphics.mspx.

[7] Ein Warp ist die Mindestmenge an Threads, die von einem GPU-Multiprozessor zugleich bearbeitet werden.

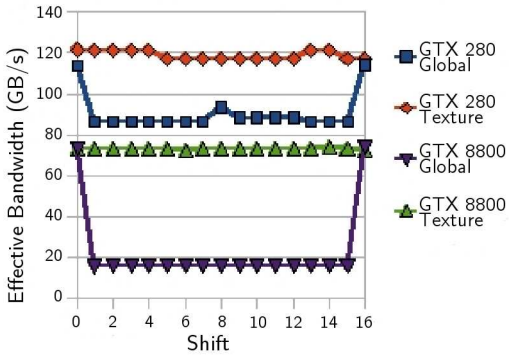

Abbildung 2.12: Performance-Unterschiede beim Lesen von Global und Texture Memory mit unterschiedlichen Offsets [26, S. 35].

liegt. Optimaler Weise erreichen die Threads, die demselben Warp angehören, die beste Performance [27, S. 93].

Darüber hinaus können durch sogenannte „Texture Fetches“ unoptimierte Zugriffe auf den globalen Speicher (Uncoalesced Loads) umgangen werden. Unter gewissen Umständen kann es dann sein, dass die Zugriffe auf den Texturen-Speicher schneller ausgeführt werden, wie Zugriffe auf den globalen Speicher (dargestellt in Abb. 2.12).

2.3.1 Streams

Normalerweise sind Speichertransfers mittels `cudaMemcpy` blockierend, d. h. der Host kann die Abarbeitung erst dann fortsetzen, wenn der Transfer abgeschlossen ist. Jedoch bietet die CUDA-API auch die Möglichkeit, asynchrone Funktionen zu verwenden. Asynchrone Funktionen blockieren nicht (non-blocking) und die Kontrolle wird direkt nach dem Aufruf an den Host zurück gegeben, auch wenn die aufgerufene Funktion noch abgearbeitet wird [26, Abschn. 3.1.2]. Bezüglich CUDA sind die folgenden Funktionen asynchron:

- Kernel-Starts
- Device ↔ Device Speicher-Kopieroperationen
- Host ↔ Device Kopiervorgänge bei Datenmengen kleiner 64 KB
- Kopiervorgänge unter Verwendung von Funktionen mit dem Suffix „`Async`“
- Memset Funktionen

Werden für Kopieroperationen die asynchronen Funktionen des Frameworks (z. B. `cudaMemcpyAsync`) eingesetzt, wird Pinned Memory und ein weiterer Parameter – eine Stream-ID – benötigt. Ein Stream ist eine Abfolge von

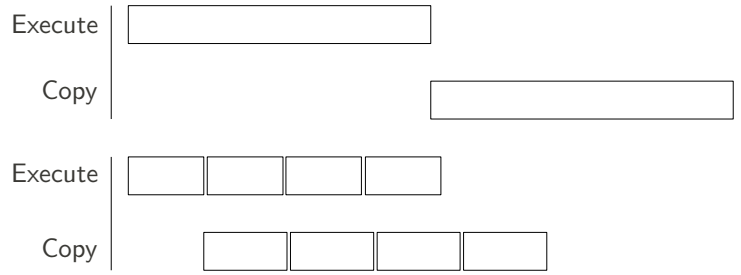

Abbildung 2.13: Überlagerung von Kernel- und Kopierzeiten [26, S. 18].

```
1 size=N*sizeof(float)/nStreams;
2 for (i=0; i<nStreams; i++){
3   offset = i*N/nStreams;
4   cudaMemcpyAsync(a_d+offset, a_h+offset, size, dir, stream[i]);
5 }
6 for (i=0; i<nStreams; i++){
7   offset = i*N/nStreams;
8   kernel<<<N/(nThreads*nStreams), nThreads,0, stream[i]>>>(a_d+offset);
9 }
```

Programm 2.1: Asynchrone Aufteilung auf Streams, die in mehreren Stages abgearbeitet werden [26, S. 17].

Operationen, die in geordneter Reihenfolge am Device ausgeführt werden [27, S. 39]. Operationen von verschiedenen Streams können verschachtelt und unter Umständen auch überlappend abgearbeitet werden – eine Möglichkeit Kopier- und Rechenvorgänge zu überlagern.

In Abbildung 2.13 wird dargestellt, wie die Aufteilung in mehrere asynchrone Execution- und Copy-Stages die Gesamtlaufzeit verringern kann. Listing 2.1 zeigt, wie der zugehörige Quellcode aussehen könnte. In diesem Beispiel können sich die Memcpy-Operationen und Kernels aus den verschiedenen Streams überlappen. Zu beachten ist, dass eine Erhöhung der Anzahl an Streams nicht stetig zu einer Verringerung der Laufzeit führt. Dies ist unter anderem darauf zurückzuführen, dass eine Erhöhung der Streams auch einen erhöhten Verwaltungs-Aufwand mit sich zieht. Außerdem können sich Streams nur dann ausreichend überlappen, wenn die GPU nicht voll ausgelastet ist und noch Ressourcen für die Überlappung von Operationen vorhanden sind. Des Weiteren müssen die Zugriffe auf den Speicher koordiniert werden, da nun der Kernel auf den Speicher zugreift, während kopiert wird. Deshalb ist es nötig die Auswirkung von der Verwendung von Streams für Applikationen und GPU-Baureihen separat zu analysieren.

2.3.2 AES-Implementierung mit CUDA

Für die Realisierung des AES-GCM wird – wie in Abschn. 2.2 erwähnt – ein AES im CTR-Modus benötigt. Der Vorteil des CTR-Modus ist, dass aufsteigende Counter verschlüsselt werden und dies parallel möglich ist. Bevor es an die eigentliche Implementierung geht, gilt es eine grundsätzliche Designfrage zu klären. Da die AES-Rundenoperationen auch mittels Nachschlagen in vorausberechneten Tabellen realisiert werden können (s. auch [5, Abschn. 5.2]), muss analysiert werden, ob dies auf der GPU Performance-Vorteile bringt.

Nach der Analyse von verschiedenen Applikationen (s. auch [2], [29], [17]) hat sich herausgestellt, dass eine für 32-Bit-Architekturen optimierte, mit Nachschlagen in Tabellen umgesetzte AES-Version, die größte Performance erreicht. So eine Implementierung beschreibt unter anderem Alexander Ottesen in seiner Master Thesis „*Efficient parallelisation techniques for applications running on GPUs using the CUDA framework*“ [29]. Er vergleicht die Unterschiede zwischen einer *standard-AES*-[8] und einer *AES-lookup*-Version[9]. Mit diesen zwei Implementierungen ist es ihm möglich, die Unterschiede zwischen rechen- und speicherintensiven Operationen auf GPUs zu analysieren. Die *standard-AES*-Implementierung hat ihren Fokus auf den Rechenkapazitäten der GPU, wohingegen *AES-lookup* auf die schnelle Abarbeitung von Speicherzugriffen angewiesen ist. Die in dieser Arbeit benötigte AES-CTR-Implementierung für den AES-GCM setzt auf die *AES-lookup*-Version von A. Ottesen auf. Die Änderungen an der Applikation beinhalten unter anderem den Wechsel von Datei-Verschlüsselung zu der Erstellung eines Speicherbereichs von verschlüsselten Countern. Der AES-Kernel soll dabei problemlos mit mehreren unterschiedlichen Schlüsseln, unterschiedlichen Startwerten und unterschiedlicher Anzahl an zu erstellenden Countern umgehen können. Die benötigten Vorarbeiten, sodass der AES-Kernel nur mehr auf die Startwerte zugreifen muss und Counter verschlüsseln kann, werden noch am Host erledigt. All diese Vorarbeiten werden von einer Funktion übernommen – diese wird von nun an auch als „GPU-Modul“ bezeichnet und in Abschn. 3.3 näher beschrieben. Die Umsetzung der AES-Runden selbst ist gleich geblieben, da diese bereits optimiert implementiert sind und eine ausreichende Performance aufweisen.

Ablauf des AES-Kernels

Vor dem Starten des Kernels muss vom GPU-Modul die Anzahl an Threads und Blöcken festgelegt werden. Außerdem müssen die Texturen für die Rundenschlüssel sowie die Initialisierungsvektoren (Nonces) angelegt werden. Die Rundenschlüssel und die Nonces werden am Host so vorbereitet, dass sie

[8] Basierend auf dem Projekt erhältlich unter http://www.hoozi.com/Articles/AESEncryption.htm.

[9] Unter Verwendung des Quellcodes von http://www.efgh.com/software/rijndael.htm.

später vom Device nur mehr gelesen werden müssen. Es ist daher auch kein Problem, dass diese beiden Parameter read-only im Texturen-Speicher abgelegt werden. Dies hat den Vorteil, dass die Parameter gecacht werden können und nicht auf Coalesced Access geachtet werden muss, wie es bei globalem Speicher der Fall wäre. Wichtig ist, dass wenn der Kernel läuft, immer ein Thread einen Counter erzeugt und diesen verschlüsselt. Das bedeutet auch, dass die Anzahl an gestarteten Threads die Datenmenge an verschlüsselten Countern bestimmt. So erzeugen z. B. 512 Threads (meist die verwendete Blockgröße) ebenso viele Counter, von denen jeder 16 Byte groß ist. Als Ergebnis liefern die 512 Threads $512 * 16 = 8192$ Byte an verschlüsselten Daten. Da 512 Threads meist nicht ausreichen, um die GPU auszulasten, wird eine fixe Anzahl an Blöcken für eine Key/Nonce-Kombination festgelegt. Dieser fixer Multiplikator wird mit 8 parametrisiert, sodass zumindest immer $8 \times$ `blockDim` Threads verschlüsselte Counter für einen Key erzeugen. Somit werden zumindest 65 KB – diese Datenmenge wird von nun an auch als „**Segment**“ bezeichnet – an verschlüsselter Datenmenge von der GPU berechnet. Die Anzahl an Segmenten kann pro Key/Nonce-Kombination individuell konfiguriert werden, je nachdem welche Datenmenge benötigt wird. Hierzu ein Beispiel:

- Für Key/Nonce A werden vier Segmente an verschlüsselten Daten benötigt. Es werden daher 32 CUDA-Blöcke erzeugt.
- Für Key/Nonce B werden acht Segmente an verschlüsselten Daten benötigt. Es werden daher 64 CUDA-Blöcke erzeugt.

Bei einem eindimensionalen Grid müssen also die ersten 32 Blöcke Key/Nonce A verwenden, und die nächsten 64 Blöcke Key/Nonce B. Da diesen Blöcken jedoch noch die Information fehlt, welchen Key bzw. Nonce sie zur Verschlüsselung verwenden müssen, werden dem Kernel Index-Felder mit übergeben. Diese Felder teilen den Blöcken mit, auf welchen Key/Nonce sie zur Laufzeit zugreifen müssen. Im obigen Beispiel würde dieses Feld wie folgt aussehen:

$$\begin{array}{rccccccc} & 0 & \cdots & 31 & 32 & \cdots & 95 \\ \text{Key/Nonce Index} = & \{0 & \cdots & 0 & 1 & \cdots & 1\} \end{array}$$

Darüber hinaus wird ein weiteres Feld benötigt, dass den Blöcken die zugehörigen Offsets mitteilt, die sie zum Nonce addieren müssen. Dieses Feld sähe so aus:

$$\begin{array}{rcccccccc} & 0 & & \cdots & 31 & 32 & & \cdots & 95 \\ \text{Nonce Offset} = & \{0 & 1 & \cdots & 31 & 0 & 1 & \cdots & 63\} \end{array}$$

Die Blöcke verwenden die Offsets, um die richtigen Werte, die zum Nonce addiert werden müssen, zu berechnen. Bei einer Block-Dimension von 512 Threads muss Block 0 $0*512$, Block 1 $1*512$, Block 2 $2*512$ usw. zum Nonce addieren. Die Threads aus den jeweiligen Blöcken verwenden das Resultat

als Startwert für ihre Berechnung der Counter. Dazu addieren sie zu dem soeben errechneten Wert ihre Thread-Indices:

$$\begin{aligned} \text{Nonce} &\Rightarrow 0\text{x}00224478 \\ 0\text{x}00224478 + \texttt{Thread-Index 0} &\Rightarrow 0\text{x}00224478 \\ 0\text{x}00224478 + \texttt{Thread-Index 1} &\Rightarrow 0\text{x}00224479 \\ &\vdots \\ 0\text{x}00224478 + \texttt{Thread-Index n} &\Rightarrow 0\text{x}00224478 + n \end{aligned}$$

Somit erhält jeder Thread einen Counter, den er verschlüsseln muss und da keine Abhängigkeiten unter den Threads bestehen, können alle Threads parallel ihre Counter verschlüsseln. Für die Verschlüsselung benötigen die Threads Lookup-Tabellen, auf die nur lesend zugegriffen werden muss, und die sich daher im Constant Memory befinden. Der interne State – die 16 Byte die verschlüsselt werden – beinhaltet den gerade berechneten Counter und wird bis zur letzten Runde in Registern abgelegt. Nach der letzten AES-Runde wir der State in einen Shared-Memory-Bereich geschrieben und wenn alle Threads die Verschlüsselung abgeschlossen haben, werden die Ergebnisse im Global Memory zusammengetragen und von dort zurück auf den Host übertragen.

Zusammengefasst ergibt sich für den Kernel folgender Ablauf:

1. Jeder Thread kopiert sich den Nonce aus der Textur in Register, der zu seinem Block gehört. Diese Information erhält er über den Zugriff auf das Index-Array: Index[`BlockIdx.x`].
2. Jeder Thread addiert seinen Thread-Indices zu seinem Nonce, um aufsteigende Counter-Werte zu generieren. Gehören mehrere Blocks zu einem Nonce, so muss außerdem ein Offset zum Nonce addiert werden, da ansonsten Blöcke die gleichen Werte berechnen. Der Offset ergibt sich aus: Nonce Offset[`BlockIdx.x`] $\times$ `BlockDim.x`.
3. Der soeben erstellte Counter-Wert wird von jedem Thread in Registern gespeichert und dient von nun an als State für die AES-Runden.
4. Jeder Thread operiert in den AES-Runden auf seinen Werten in den Registern unter Verwendung des für den Block zugehörigen Schlüssels über das Index-Feld.
5. Wurden die AES-Runden beendet, schreiben die Threads ihre Ergebnisse in den Shared Memory.
6. Sind alle Threads mit der Abarbeitung der AES-Runden fertig, so werden die Ergebnisse aus dem Shared Memory sequentiell in den Global Memory geschrieben.

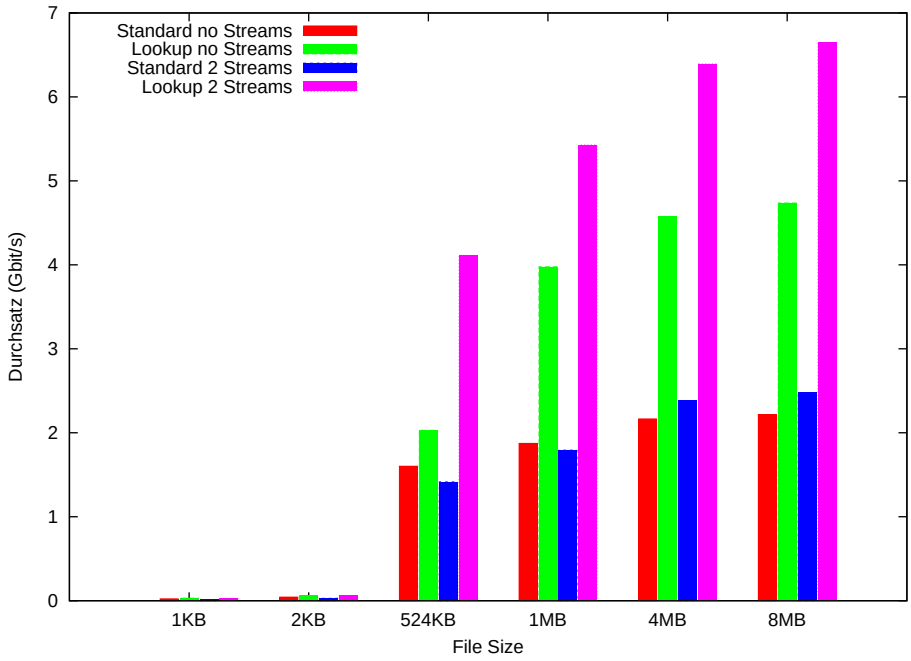

Abbildung 2.14: Performance-Unterschiede auf einer GTX285 (PCI Express 1.1×16) zwischen *standard-AES* mit und ohne Streams, sowie *AES-lookup* mit und ohne Streams.

Testergebnisse

Im folgenden Abschnitt werden die Performance-Ergebnisse, die aus dem AES-Kernel-Test resultieren, vorgestellt. Diese werden zwar schon ausführlich in der Thesis von A. Ottesen beschrieben (vgl. [29, Abschn. 4.4]), jedoch mit unterschiedlicher Hardware und Parametern. Ottesen verwendet für seine Tests stets eine Input-Datei mit 512 MB, um die GPU ausreichend zu beschäftigen. Solche Datenmengen sind aber im Netzwerk-Bereich nur dann realistisch, wenn für sehr viele Verbindungen gleichzeitig Daten vorausberechnet werden müssen. Im Regelfall müssen aber für einige Verbindungen sehr schnell Daten zur Verfügung stehen und daher muss die Performance auch bei kleinen Datenmengen sehr groß sein. Daher werden die Tests erneut durchgeführt und vor allem darauf geachtet, wie der Durchsatz bei kleineren Datenmengen aussieht. Abbildung 2.14 zeigt anschaulich, wie sich die Performance zwischen *standard-AES* und *AES-lookup* verhält. Es wird deutlich, dass die Lookup-Implementierung bei allen Datenmengen einen höheren Durchsatz erzielt. Außerdem wird dargestellt, dass durch die Verwendung von zwei Streams der Durchsatz noch einmal gesteigert werden kann.

Kapitel 3

Realisierung von AES/GCM für IPsec mit GPUs

Im folgenden Kapitel wird näher auf die Umsetzung des AES-GCM-Algorithmus für die Verwendung in IPsec eingegangen. Es wird begonnen mit einer Einführung, wie AES-GCM in IPsec verwendet wird und sich in das System einfügt. Dabei wird vor allem die Initialisierung und Zusammensetzung der einzelnen Parameter wichtig sein. Im zweiten Abschnitt werden die am Standard vorgenommen Änderungen sowie deren Rechtfertigung beschrieben. Da ein Teil der Umsetzung des AES-GCM-IPsec-Standards sich nicht für ein effizientes Vorausberechnen und zur Parallelisierung eignet, werden Änderungen vorgenommen und deren Relevanz für die spätere Implementierung erklärt. Im letzten Teil des Kapitels liegt der Fokus auf einer Möglichkeit der Implementierung des AES-GCM unter Verwendung eines GPU- und eines CPU-Moduls.

3.1 Verwendung von AES/GCM für IPsec

IPsec ist eine Sammlung von Protokollen für die Unterstützung von geschützter Kommunikation über IP-Netzwerke [11, Kap. 5]. Die Hauptprotokolle, im Sinne der Anwendung von kryptografischen Algorithmen auf Netzwerk-Pakete, werden durch Authentication Header (AH), Encapsulation Security Payload (ESP) und den Protokollen zum Schlüssel-Management repräsentiert[1]. Aus sicherheitstechnischer Sicht decken die Protokolle vor allem folgende Bereiche ab [21, Abschn. 7.1]:

- Integrität
- Data Origin Authentication

[1] Für detaillierte Informationen über ESP und AH wird auf [21, Abschn. 7.8] und [30] verwiesen.

- Vertraulichkeit der Pakete (Confidentiality)
- Traffice Flow Confidentiality
- Schutz vor Replay-Attacken und anderen Denial-of-Service-Angriffen

Diese Schutzfunktionen werden dabei z. B. durch HMACs, Verschlüsselungs-Algorithmen und Sequenz-Nummern umgesetzt (für weitere Informationen s. auch Abschn. 7.2 in [21]).

Die Verwaltung der Sicherheitsprotokolle – z. B. eingesetzte Algorithmen oder Gültigkeitsdauer – werden von sogenannten „Security Associations“ (SAs) übernommen, sie bilden das Herzstück von IPsec [21, Abschn. 7.4]. Eine SA ist immer unidirektional, weshalb es für jede Verbindung eine eingehende (Inbound-) und eine ausgehende (Outbound-) SA gibt.

Erzeugt werden SAs in der Regel vom Internet-Key-Exchange-Protokoll (IKE), da eine manuelle Erzeugung zwar unterstützt, aufgrund der erhöhten Sicherheitsrisiken aber nicht empfohlen wird. Die ausgehandelten SAs werden dann in einer Security Association Database (SAD) abgelegt. Auch der erforderliche Austausch von geheimen Schlüsseln wird von IKE durchgeführt. Wie die Schlüssel genau entstehen kann z. B. in [21, Abschn. 8.5.5] nachgelesen werden.

Die einzelnen Protokolle von IPsec sind in mehreren Request for Comments (RFCs) standardisiert. Für das Verständnis dieser Arbeit sind vor allem folgende wichtig:

- RFC 4301 – *Security Architecture for the Internet Protocol* [16]: Dieser Standard beschreibt die Anforderungen an ein System, das IPsec umsetzt. Er definiert fundamentale Elemente, deren Aufbau und Zusammenspiel in Zusammenhang mit IP.
- RFC 4303 – *IP Encapsulating Security Payload (ESP)* [15]: In diesem Dokument wird das ESP-Protokoll, sowie dessen Funktionen und Aufgaben auf Netzwerk-Ebene, erläutert.
- RFC 4106 – *The Use of Galois/Counter Mode (GCM) in IPsec Encapsulating Security Payload (ESP)* [32]: Dieser RFC erklärt die Benutzung von AES-GCM als IPsec-ESP-Algorithmus für die Sicherstellung von Vertraulichkeit und Data Origin Authentication.

Im folgenden Abschnitt wird vor allem der RFC 4106 im Vordergrund stehen. Es wird zusammengefasst, wie sich AES-GCM in IPsec einfügt und angewendet werden kann. Der Fokus wird dabei vor allem auf die vorhandenen Input-Parameter, die Verschlüsselung von einzelnen Paketen sowie die Änderung der Parameter über mehrere Pakete hinweg gelegt.

Wie bereits erwähnt spezifiziert der RFC 4106 wie AES-GCM als ESP-Combined-Mode-Algorithmus eingesetzt wird. Der traditionelle CTR-Modus hat sich als Algorithmus für High-Speed-Implementierungen aufgrund seiner Pipelining-Möglichkeiten etabliert [32, S. 2]. Der Name „Combined Mode“ für AES-GCM ergibt sich aus der Kombination von Vertraulichkeit, Authen-

tizität und Integrität. Die Verwendung von AES-GCM für ESP in IPsec wird von nun an als auch AES-GCM-ESP bezeichnet.

Die Parameter, die für die Verwendung von AES-GCM benötigt werden, finden sich in Abschn. 2.1 wieder. Diese Parameter werden im ESP-Protokoll spezifisch initialisiert und verwendet, um eine standardisierte Ver- und Entschlüsselung zu garantieren. Auch wie die Nutzdaten eines Netzwerk-Pakets (Payload) und dessen zugehörige Initialisierungs-Daten angeordnet und versendet werden, ist geregelt. Vorab jedoch eine Erklärung, wie ein ESP-Paket aufgebaut ist (Abb. 3.1):

1. SPI: Der Security Parameter Index wird vom Empfänger eines Pakets für die Zuordnung zu einer SA genutzt [15, S. 10].
2. Sequence Number: Die Sequence Number ist ein 32 Bit langer Counter, der pro gesendetem Paket erhöht wird. Sie kann somit auch als per-SA Paket Sequence Number verstanden werden [15, S. 12]. Bei der Aushandlung einer SA wird dieser Wert auf 0 gesetzt und von da an bei einem ausgehenden Paket um eins erhöht. Als Erweiterung dieser Sequence Numbers ist es auch möglich, 64 Bit lange Werte zu verwenden. Die Umsetzung dieser 64-Bit-Nummern ist aber nicht verpflichtend [15, S. 12].
3. Payload: Der Payload ist von variabler Länge und besteht aus den Daten des zu verschlüsselnden IP-Pakets.
4. Padding: Padding wird benötigt, um die Länge des Payloads auf ein Vielfaches der Blocklänge von AES zu bringen. Padding wird aber auch verwendet, um sicher zu stellen, dass der resultierende Ciphertext an einer 4-Byte-Grenze endet. Wie in Abb. 3.1 ersichtlich, müssen vor allem die Felder „Pad Length“ und „Next Header“ an einer rechts ausgerichteten 4-Byte-Grenze enden, um korrektes Alignment des ICV zu garantieren. Die maximale Länge des Paddings beträgt 255 Byte [15, S. 14].
5. Pad Length: Dieses Feld beinhaltet die Länge der zum Payload hinzugefügten Pad-Bits.
6. Next Header: Der Next Header beschreibt die Art der Daten des Payloads, z. B. ein IPv4- oder IPv6-Paket. Dazu werden Protokoll-Nummern aus einem vordefiniertem Set gewählt und in den Next Header eingetragen.
7. Traffic Flow Confidentiality (TFC) Padding: Diese Art von Padding ist optional und soll die Durchführung von Traffic-Flow-Attacken verhindern. TFC-Padding wird separat durchgeführt, da das normale Padding mit einer maximalen Länge von 255 Byte für die Verhinderung der Attacken nicht ausreicht. TFC-Padding kann auch nur dann erfolgen, wenn im Payload eine genau Angabe zur Länge des enthaltenen IP-Datagrams angegeben ist. Im Tunnel Mode ist dies immer der Fall,

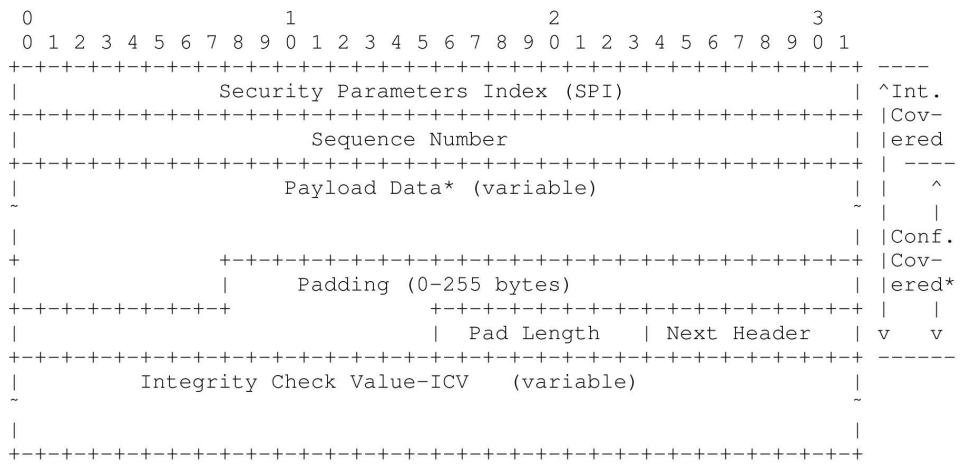

Abbildung 3.1: Aufbau eines ESP-Pakets [15, S. 5].

da der originale IP-Header erhalten bleibt. Im Transport Mode kommt es auf das Protokoll des nächst höher gelegenen Layers an [15, S. 17].

8. Integriy Check Value (ICV): Der ICV wird unter Einbeziehen des ESP-Headers, des Payloads und des ESP-Trailers errechnet. Da der ICV optional ist, ist er nur vorhanden, wenn der Integritätsschutz im Zuge der SA ausgehandelt wurde. Im Falle von AES-GCM-ESP ist die Verwendung des ICV also bindend.

Die einzelnen Paket-Bereiche dienen nun als Input-Parameter für die Erstellung eines verschlüsselten AES-GCM-ESP-Pakets:

1. Payload: Der Payload wird im Falle von AES-GCM-ESP noch weiter strukturiert. Da für die Ver- bzw. Entschlüsselung ein Initialisierungsvektor (IV) benötigt wird, ist der IV immer Teil des Payloads und wird unverschlüsselt im Paket mit übertragen (Abb. 3.2). Somit besteht der Payload aus dem 8 Byte langen IV und dem Ciphertext. Diese Unterstruktur mit dem IV als Zusatz zum Payload läuft transparent ab und wird nicht separat angegeben, da sie sich aus dem verwendeten Cipher ergibt. Der IV des AES-GCM-ESP-Pakets ist nicht zu verwechseln mit dem Initialisierungsvektor des AES-GCM, der zum besseren Verständnis nun nicht mehr als IV, sondern als „Nonce“ bezeichnet wird (s. auch Abb. 3.5). Für diese beiden Teile des Payloads gelten folgende Anforderungen:

    - IV: Der IV darf sich für einen vereinbarten Schlüssel nicht wiederholen, da die Sicherheit des Algorithmus vor allem davon abhängt, dass eine Kombination von IV und Schlüssel nur ein einziges Mal verwendet wird. Daher müssen diese 8 Bytes, sowie deren Änderung über mehrere Pakete hinweg, so erzeugt werden, dass

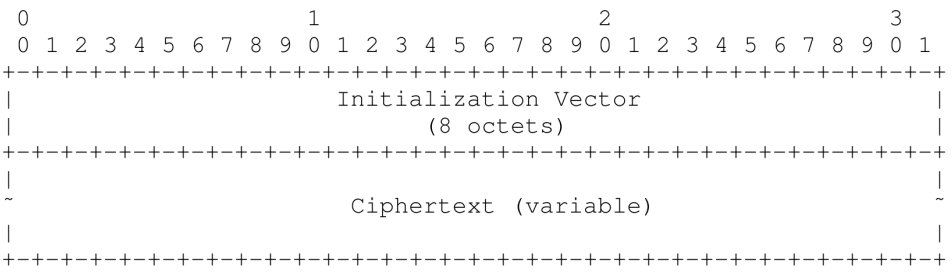

Abbildung 3.2: Strukturierter Payload eines mit AES-GCM verschlüsselten Pakets [32, S. 3].

sie sich für den aktuellen Schlüssel nicht wiederholen. Ein nahe liegender Weg dies umzusetzen ist den IV pro Paket um eins zu erhöhen – sozusagen ein Counter der Pakete. Jedoch ist jegliche Methode, die Einzigartigkeit garantieren kann, einsetzbar, auch z.B. ein Linear Feedback Shift Register (LFSR) [32, S. 4]. Da der IV unverschlüsselt bleibt und im Paket mit übertragen wird, ist keine Koordination mit dem Empfänger des Pakets nötig.

- Ciphertext: Die Daten, die mittels AES-GCM verschlüsselt werden, setzen sich aus dem IP-Datagram gefolgt vom Padding (inklusive etwaigem TFC-Padding) und der Pad-Länge, sowie dem Next Header zusammen. Diese Daten werden verschlüsselt und resultieren im Ciphertext, dessen Länge identisch zu der Summe der Längen der Inputs ist.

2. Nonce: Der Nonce, der für die AES-GCM-Ver- und -Entschlüsselung benötigt wird, setzt sich aus einem von IKE generierten Salt und dem AES-GCM-ESP-IV zusammen (Abb. 3.3). Der Salt ist 4 Byte lang und wird zu Beginn einer SA durch IKE erzeugt und bleibt anschließend über die Gültigkeitsdauer dieser SA konstant. Der IV (8 Byte) konkateniert mit dem Salt (4 Byte) ergeben zusammen einen 12 Byte langen Nonce für die Verschlüsselung eines Pakets mit AES-GCM. Diese 12 Byte sind insofern ideal, da der AES-GCM für 12-Byte-Nonces eine verkürzte Erstellung der Initialisierungsvektoren spezifiziert [10, S. 15]. Der Standard sieht vor, dass bei 12-Byte-Nonces nur mehr die Bits $0^{31} \parallel 1$ angehängt werden müssen (sieh dazu auch Abschn. 2.2.3).
3. AAD: SPI und Sequence Numbers werden zusammen als AAD in AES-GCM eingesetzt (s. auch Abb. 3.4). Dies gilt ebenso bei erweiterten Sequence Numbers von 64 Bit.
4. ICV: Der ICV besteht aus dem von AES-GCM berechnetem Authentication Tag, wobei 16 Byte lange Tags unterstützt werden müssen, 8 oder 12 Byte lange Tags unterstützt werden können. Wie bereits erwähnt muss bei AES-GCM-ESP ein ICV verwendet werden, da der

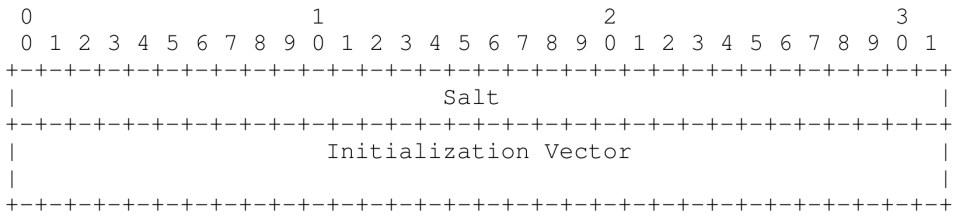

Abbildung 3.3: Zusammensetzung der Nonce aus dem IKE-Salt und dem AES-GCM-ESP-IV [32, S. 4].

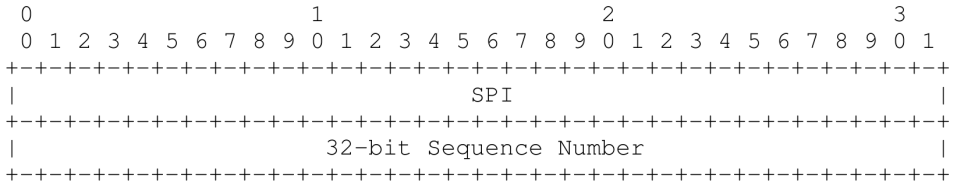

Abbildung 3.4: Zusammensetzung der AAD aus dem SPI und der Sequence Number [32, S. 5].

Authentication Tag für die Integritätsprüfung zuständig ist.

Abb. 3.5 zeigt abschließend einen Überblick, wie sich die einzelnen In- und Outputs des AES-GCM-ESP zusammensetzen.

3.2 Aufbau der Architektur

Bevor nun näher auf die einzelnen Teile der Applikation sowie deren Implementierungen eingegangen wird, werden der grundsätzliche Aufbau und der schematische Ablauf erklärt. Wie bereits in Abschn. 2.2 erwähnt ist der Vorteil des CTR-Modus, dass die Verschlüsselung von Countern im Voraus stattfinden kann. Diesen Teil übernimmt die GPU, wobei gesagt werden muss, dass sich jede Art von Coprozessor für diese Aufgabe eignen würde. Für das Vorausberechnen werden dem GPU-Modul die durch die SA spezifizierten Parameter und die gewünschte Anzahl an Segmenten übergeben (vgl. Abschn. 3.3). Daraufhin verschlüsselt die GPU die Counter und transferiert die Daten in einen Speicherbereich am Host zurück. Die aufwendige AES-Verschlüsselung wurde somit durch die GPU bereits erledigt.

Am Host werden von einem Controller-Modul die von der GPU verschlüsselten Counter in einem Double-Buffer-System verwaltet. D. h. jede SA besitzt zwei Puffer, die mit verschlüsselten Countern befüllt sind. Trifft am Host nun ein Netzwerk-Paket ein, das verschlüsselt werden soll, so greift die CPU auf einen gefüllten Puffer zu und führt ein XOR mit den verschlüsselten Countern durch. Mit jedem zu verschlüsselnden Paket wird demnach ein

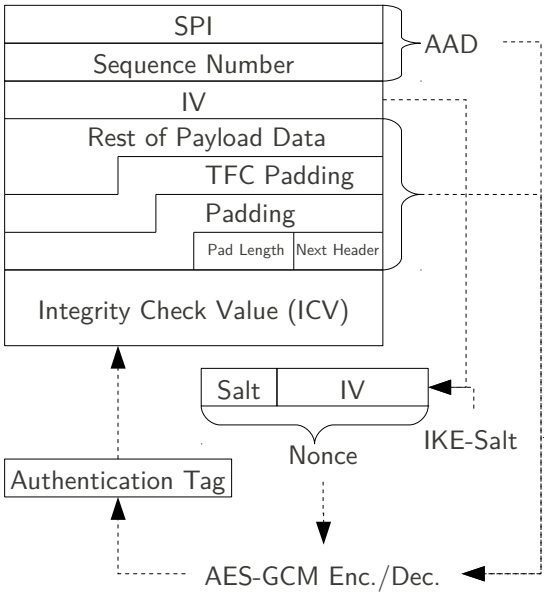

Abbildung 3.5: Zusammensetzung der einzelnen Parameter aus einem ESP-Paket für die Ver- bzw. Entschlüsselung mit AES-GCM.

Counter aus dem Puffer verbraucht und wenn die Anzahl an Countern eines Puffers aufgebraucht ist, wird ein Wechsel zum zweiten Puffer durchgeführt. Der zweite Puffer kann somit sofort wieder zum Verschlüsseln hergenommen werden. Der soeben leer gewordene Puffer muss jedoch von der GPU wieder aufgefüllt werden. Solange also nicht schneller verschlüsselt wird, als ein Puffer von der GPU befüllt werden kann, wird es nicht dazu kommen, dass ein Paket nicht verschlüsselt werden kann. Die maximale Geschwindigkeit, mit der die Puffer befüllt werden können, ist daher auch gleich bedeutend mit der maximalen Verschlüsselungs-Geschwindigkeit, wenn die Zeit für Verwaltungsaufgaben außer Acht gelassen wird.

Für die Berechnung des Authentication Tags wird am Host auf die GHASH-Funktion zurückgegriffen. Wurde erfolgreich der Ciphertext erstellt und der Authentication Tag berechnet, so werden die einzelnen Teile zu einem Paket zusammengefügt. Abb. 3.6 zeigt den soeben erklärten Ablauf, jedoch ohne das Befüllen eines leeren Puffers.

3.2.1 Modifizierungen der Standards

Einige Design-Eigenschaften im RFC 4106 [32] und in der GCM-Spezifikation [23] verhindern ein effizientes Vorausberechnen von verschlüsselten Countern für AES-GCM-ESP. Diese Hindernisse stehen nicht direkt in Bezug auf das Vorausberechnen auf der GPU, sondern sind auch für ein Vorausberechnen auf der CPU oder anderen Coprozessoren relevant. Im folgenden

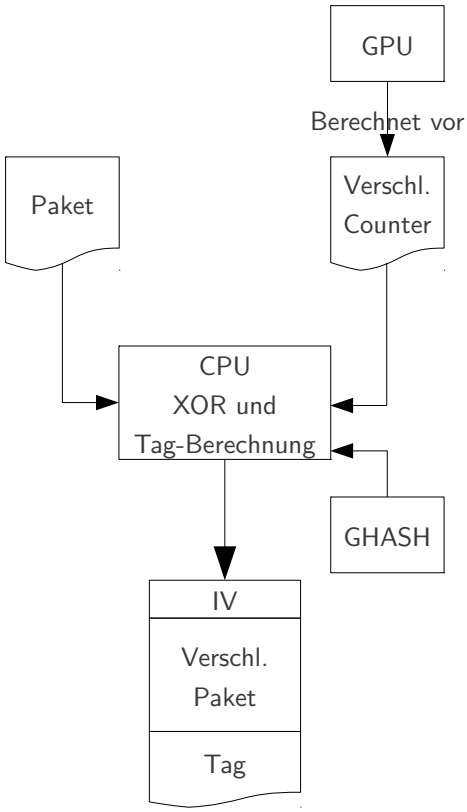

Abbildung 3.6: Schematischer Ablauf der Verschlüsselung eines Netzwerk-Pakets: 1. Die Counter wurden bereits durch die GPU im Voraus verschlüsselt. 2. Die CPU muss für die Verschlüsselung des Pakets nur mehr ein XOR mit den verschlüsselten Countern durchführen. 3. Für die Berechnung des Authentication Tags – Multiplikation mit H im Galois-Feld – wird die CPU benutzt. 4. Abschließend fügt die CPU die einzelnen Teile zusammen.

Abschnitt werden die grundlegenden Probleme aufgezeigt und anschließend Lösungen sowie deren Auswirkungen näher betrachtet.

IV- und Nonce-Zusammensetzung

Laut RFC 4106 setzen sich die ersten 12 Byte der Nonce aus dem von IKE generierten Salt und dem in ESP definierten IV zusammen (Abb. 3.3). Anschließend werden zu Beginn von AES-GCM zur Vervollständigung auf 16 Byte die Bits $0^{31} \parallel 1$ angehängt (siehe dazu [23, S. 5]). Diese einfache Art der Nonce-Erstellung wird jedoch nur dann vollzogen, wenn der fordere Teil 12 Byte lang ist. Ansonsten wird der Nonce der Funktion GHASH (Abschn. 2.2.2) übergeben[2]. Der RFC 4106 ist also so auf den AES-GCM ab-

[2]Auf die Details des Algorithmus wird auch schon in Abschn. 2.2.3 näher eingegangen.

gestimmt, dass für den Nonce GHASH nicht aufgerufen werden muss. Dazu ein kleines praktisches Beispiel:

- Paket 1
  - Größe: 64 Byte
  - Salt: 0x12, 0x14, 0x65, 0x42
  - IV: 0x15, 0x48, 0x98, 0x32, 0x89, 0x63, 0x48, 0x82
  - Nonce: Salt ‖ IV ‖ 0x00, 0x00, 0x00, 0x01
- Paket 2
  - Größe: 32 Byte
  - Salt: 0x12, 0x14, 0x65, 0x42
  - IV: 0x15, 0x48, 0x98, 0x32, 0x89, 0x63, 0x48, 0x83
  - Nonce: Salt ‖ IV ‖ 0x00, 0x00, 0x00, 0x01

Für die Verschlüsselung des ersten Pakets müsste also nun

$$\text{Nonce} = \text{Salt} \parallel \ldots 0\text{x}82 \parallel 0\text{x}00, 0\text{x}00, 0\text{x}00, 0\text{x}01$$

solange inkrementiert und verschlüsselt werden, bis 64 Byte an verschlüsselten Countern für ein XOR mit dem Plaintext vorhanden sind. Für das nächste Paket wäre

$$\text{Nonce} = \text{Salt} \parallel \ldots 0\text{x}83 \parallel 0\text{x}00, 0\text{x}00, 0\text{x}00, 0\text{x}01$$

der Wert, der erhöht werden müsste. Für dieses Paket wären aber nur 32 Byte für ein XOR nötig. Nach der Verschlüsselung muss dem Empfänger nur – wie Eingangs im ESP-Paketformat beschrieben – der IV übermittelt werden, da der Salt beiden bekannt ist und die letzten 4 Byte laut Spezifikation stets gleich bleiben. Für die Verschlüsselung des nächsten Pakets wird der IV um eins hochgezählt, der Salt vorne hinzugefügt und $0^{31} \parallel 1$ hinten angehängt. Berechnet nun die GPU für die einzelnen Pakete verschlüsselte Counter voraus, müsste die durchschnittliche Größe eines Pakets abgeschätzt werden. Dann könnten auf der GPU verschlüsselte Counter in dieser Größe erzeugt und anschließend verwendet werden. Bei diesem Ansatz ergeben sich folgende Nachteile:

Da die Größe eines Netzwerk-Pakets kaum abzuschätzen ist, wird es dazu kommen, dass in einigen Fällen zu viel und in anderen Fällen zu wenig vorausberechnet wird. In den Fällen, in denen von der GPU zu viel berechnet wird, ist unnötig Rechenleistung verschwendet worden, da die nicht benötigten verschlüsselten Counter verworfen werden müssen. In den anderen Fällen kann das Paket nicht vollständig verschlüsselt werden, da die von der GPU erstellten Counter nicht ausreichen. Es müsste z. B. auf die CPU ausgewichen werden.

Das bedeutet, dass durch die Verwendung von zwei Countern im AES-GCM-ESP nicht effizient vorausberechnet werden kann. Der erste Counter

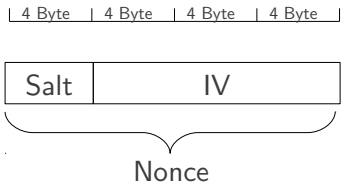

Abbildung 3.7: Modifikation 1 – Verlängerung des IV auf 12 Byte, um effizient Vorausberechnen zu können.

ist der IV, dieser erhöht sich pro verschlüsseltem Paket um eins. Der zweite Counter wird im AES-GCM selbst verwendet und fängt für jedes Paket wieder von vorne bei eins an. Besser ist es, wenn der IV auf 12 Byte verlängert wird und somit nur mehr ein Counter verwendet wird. Dieser IV wird, wie im Standard vorgesehen, zusammen mit dem Salt als Nonce verwendet (s. auch Abb. 3.7). Durch diese Verlängerung gibt es über alle Pakete hinweg und in AES-GCM selbst nur mehr einen Counter. Das hat den Vorteil, dass ein Strom von verschlüsselten Countern vorausberechnet wird und später bei der Ver- bzw. Entschlüsselung auf diesen Strom zugegriffen wird. Da in diesem Strom nur ein fortlaufender Counter verschlüsselt ist, kann für ein Paket gerade soviel aus dem Strom genommen werden, wie für das jeweilige Paket benötigt wird. Die Daten können sequentiell verwendet werden und die Größe eines Pakets spielt keine Rolle mehr.

Die Nachteile, die sich daraus ergeben, sind unter anderem die Übertragung von vier zusätzlichen Bytes pro Paket. Zusätzlich bedeutet die Modifikation für eine Hardware-Implementierung mehr Aufwand, da ein größerer Zähler umgesetzt werden muss [22, S. 5]. Auch die Verwendung eines LFSR für die Erzeugung der Counter würde mehr Aufwand bedeuten. Diese beiden Nachteile können aber im Falle einer Software-Implementierung vernachlässigt werden.

Verwendung von GHASH

Durch die Verlängerung des IV von 8 auf 12 Byte müsste – laut Spezifikation – auf jeden IV die GHASH-Funktion angewendet werden [10, S. 15]. GHASH würde dann aus den 16 Byte (Salt $\|$ IV) neue, unter Verwendung des Galois-Feldes gehashte, 16 Byte erzeugen. Da mit dieser Anwendung von GHASH erneut das Problem auftritt, dass die Counter nicht vorausgesehen werden können und die Größe der Pakete wieder abgeschätzt werden müsste, wird als zweite Modifikation GHASH für 16-Byte-Nonces weggelassen. Die 16 Byte des Nonce werden direkt als initialer Counter für AES-GCM verwendet (Abb. 3.8).

Zusammengefasst ergeben sich durch die beiden Modifikationen folgende Vorteile:

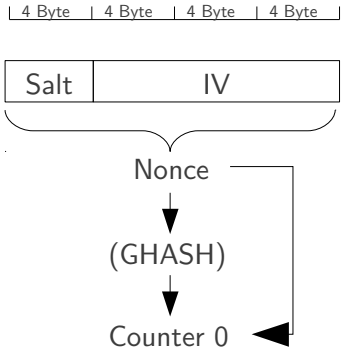

Abbildung 3.8: Modifikation 2 – Die 16 Byte von Salt und IV dienen direkt als initialer Counter (Counter 0) für AES-GCM. Solange kein Counter unter demselben Key ein zweites Mal verwendet wird, ist die Sicherheit nicht beeinträchtigt.

- Es werden pro Paket keine Counter zu viel verschlüsselt, die verworfen werden müssten. Aus dem Strom von verschlüsselten Countern, der vorausberechnet wird, kann für jedes Paket die benötigte Datenmenge entnommen werden. Das heißt, für die Verschlüsselung ist nur mehr ein XOR und die Berechnung des Authentication Tags nötig.
- Durch die direkte Benutzung von 16-Byte-Nonces ohne Anwendung von GHASH können die zukünftigen Counter vorhergesagt werden.

3.3 GPU-Modul

Das GPU-Modul nutzt den in Abschn. 2.3.2 beschriebenen AES-Kernel, um für mehrere SAs verschlüsselte Counter zu erzeugen. Es hat dabei im wesentlichen die Aufgabe, die Parameter für den Kernel-Aufruf vorzubereiten:

1. Anpassung der Parameter für die Verwendung mit Streams.
2. Berechnung der Start-Parameter für den AES-Kernel.
3. Erstellung der Texturen für Rundenschlüssel und Nonces.
4. Erstellung der Index- und Offset-Felder für den Kernel.
5. Allokierung des Speichers für das Ergebnis – die von der GPU verschlüsselten Counter.
6. Transfer der Parameter zum Device und Transfer des Ergebnisses vom Device zum Host.
7. Starten des Kernels unter Verwendung von Streams.

An die Funktion (vgl. Programm 3.1) werden dabei folgende Parameter übergeben:

- `uint32_t *rkeys`: Ein Array mit den zu verwendenden Rundenschlüssel. Die Generierung der Rundenschlüssel (Key Expansion) für AES

```
1 extern "C" float d_encrypt_nonces(uint32_t *rks, uint keys_count,
      uint32_t *nonces, uint nonces_count, uint nrounds, uint *segments,
      uint32_t **out, size_t *out_size);
```

Programm 3.1: Interface zum GPU-Modul.

muss daher schon auf der CPU erfolgt sein, bevor das GPU-Modul benutzt werden kann.

- `uint keys_count`: Die Anzahl der übergebenen Rundenschlüssel. Entspricht auch der Anzahl der übergeben Nonces (`nonces_count`) und kann somit auch als Anzahl der SAs verstanden werden, für die Counter verschlüsselt werden.
- `uint32_t *nonces`: Ein Array der Nonces, die verschlüsselt werden sollen.
- `uint nrounds`: Die Anzahl der durchzuführenden Runden für den AES-Algorithmus. Dies ergibt sich auch aus der verwendeten Schlüssellänge.
- `uint *segments`: Ein Array der gewünschten Anzahl an Segmenten pro SA (für eine Erklärung der Segmente s. auch Abschn. 2.3.2 auf Seite 21). Das erste Element des Arrays beschreibt die Anzahl der Segmente für SA 1, Element n gibt an, wie viele Segmente für SA n erzeugt werden sollen.
- `uint32_t **out`: Ein Zeiger auf das Array der Ergebnisse – die verschlüsselten Counter für alle SAs. Das GPU-Modul weist diesem Zeiger-Zeiger die richtige Adresse zu.
- `size_t *out_size`: Ein Zeiger auf die Länge des erstellten Speichers (die Größe des `out`-Arrays in Byte). Das GPU-Modul initialisiert diesen Zeiger mit der richtigen Adresse.

Beispielhafte Konfiguration der SAs

Um eine beispielhafte SA-Konfiguration vorzunehmen, können diese in einer Konfigurations-Datei parametrisiert werden. Diese Datei wird anschließend ausgelesen und die einzelnen SAs werden in einer einfach verketteten Liste am Host verwaltet. Anhang A.1 auf S. 73 zeigt ein Beispiel einer Konfiguration mit zwei SAs. Diese Konfiguration führt beim Auslesen dazu, dass am Host eine verkettete Liste mit zwei Elementen des Typs „`sa_t`“ erstellt wird. `sa_t` ist ein C-Struct und jener Datentyp, mit dem eine SA verwaltet wird. Er setzt sich aus folgenden Teilen zusammen (Programm 3.2):

- `unsigned char* key`: Der SA zugewiesene AES-Schlüssel.
- `uint32_t *rks`: Die dem AES-Schlüssel zugehörigen Rundenschlüssel.

```
 1 struct sa {
 2   unsigned char* key;
 3   uint32_t *rks;
 4   unsigned int keybits;
 5   unsigned int nrounds;
 6   uint32_t *nonce;
 7   unsigned int segments;
 8   unsigned char *H;
 9   double_buffer_t double_buffer;
10   struct sa *next;
11 };
12 typedef struct sa sa_t;
```

Programm 3.2: C-Struct für die Verwaltung der SAs.

- `unsigned int keybits`: Die Bitlänge des AES-Schlüssels.
- `unsigned int nrounds`: Die durchzuführenden AES-Runden.
- `uint32_t *nonce`: Der Startwert für AES-GCM-ESP.
- `unsigned int segments`: Die Anzahl der Segmente, die für die SA erzeugt werden.
- `unsigned char *H`: Der Hash-Subkey, verwendet von GHASH (vgl. Abschn. 2.2.2 auf S. 7).
- `double_buffer_t double_buffer`: Der SA zugewiesene Double-Buffer (näheres dazu in Abschn. 3.4.1).
- `struct sa *next`: Ein Zeiger auf das nächste SA-Element.

Mit Hilfe der Funktion „`read_config`“ wird die Konfigurations-Datei ausgelesen und die verkettete Liste erzeugt. Es wird dabei auf die Bibliothek „`libconfig`“[3] zurückgegriffen. `libconfig` ermöglicht ein einfaches parsen von strukturierten Konfigurations-Dateien und unterstützt auch bei der richtigen Zuordnung zu Datentypen beim Einlesen der Datei. An die Funktion `read_config` müssen folgende Parameter übergeben werden (Progr. 3.3):

- `char *file`: Ein String, der den Ort der Konfigurations-Datei spezifiziert. Diese Datei muss nach den Regeln von `libconfig` aufgebaut sein (vgl. Anhang A.1 auf S. 73).
- `sa_t **sa_list`: Ein Zeiger auf die Adresse des ersten SA-Elements der verketteten Liste.
- `int *sa_list_items`: Die Anzahl der in der Datei konfigurierten SAs.
- `int *sa_conf_count`: Die Anzahl der SAs, deren Segmente nicht mit 0 konfiguriert wurden.

Bis zu diesem Zeitpunkt ist es also möglich, SAs in einer Datei zu konfigurieren und diese auch auszulesen. Die daraus resultierende verkettete

[3] Zu finden unter http://www.hyperrealm.com/libconfig/.

```
1 /*
2  * Reads the config parameters in the config file to a linked list.
3  *  Each config SA section will be an elment in the linked list.
4  */
5 int read_config(char *file,sa_t **sa_list, int *sa_list_items, int *
      sa_conf_count);
```

Programm 3.3: Funktion zum Auslesen der Konfigurations-Datei. Diese Funktion erstellt auch die verkettete Liste der SA-Structs.

```
1 /*
2  * Steps through the list and collects all keys, nonces and segments in arrays.
3  * These arrays can then be used to call the gpu modul.
4  */
5 int merge_sa_data(sa_t *sa_list, unsigned char* keys, uint32_t *nonces,
      uint *segments, int *count);
```

Programm 3.4: Funktion zum Zusammentragen der konfigurierten SAs als Vorbereitung für den Aufruf des GPU-Moduls.

Liste wird dem GPU-Modul aber nicht direkt übergeben, es ist zuvor noch ein Zwischenschritt nötig. Der Funktion „`d_encrypt_nonces`“ müssen die zu verwendenden Rundenschlüssel, Nonces und Segmente aller SAs konsolidiert übergeben werden. D. h. ein Array in dem alle Rundenschlüssel der konfigurierten SAs nacheinander enthalten sind, sowie ein Array für die Nonces und eines für die Segmente. Die Konsolidierung der Schlüssel, Nonces und Segmente kann automatisiert aus der verketteten Liste mit einer Funktion namens „`merge_sa_data`“ erstellt werden. Vor dem Aufruf des GPU-Moduls braucht anschließend nur mehr die Key Expansion durchgeführt werden. Die Funktion `merge_sa_data` nimmt folgende Parameter entgegen (Programm 3.4):

- `sa_t *sa_list`: Ein Zeiger auf den Beginn der verketteten Liste.
- `unsigned char *keys`: Ein Array, groß genug, um alle Schlüssel aller SAs aufzunehmen. Die Variable „`sa_conf_count`“ wurde durch `read_config` entsprechend der Anzahl der konfigurierten SAs, deren Anzahl an Segmenten nicht 0 ist, initialisiert. Diese Variable kann nun dazu verwendet werden, um die Arrays `keys`, `nonces` und `segments` anzulegen.
- `uint32_t *nonces`: Ein Array, um alle Nonces aller SAs aufzunehmen.
- `uint *segments`: Ein Array, um die Anzahl der Segmente aller SAs aufzunehmen.

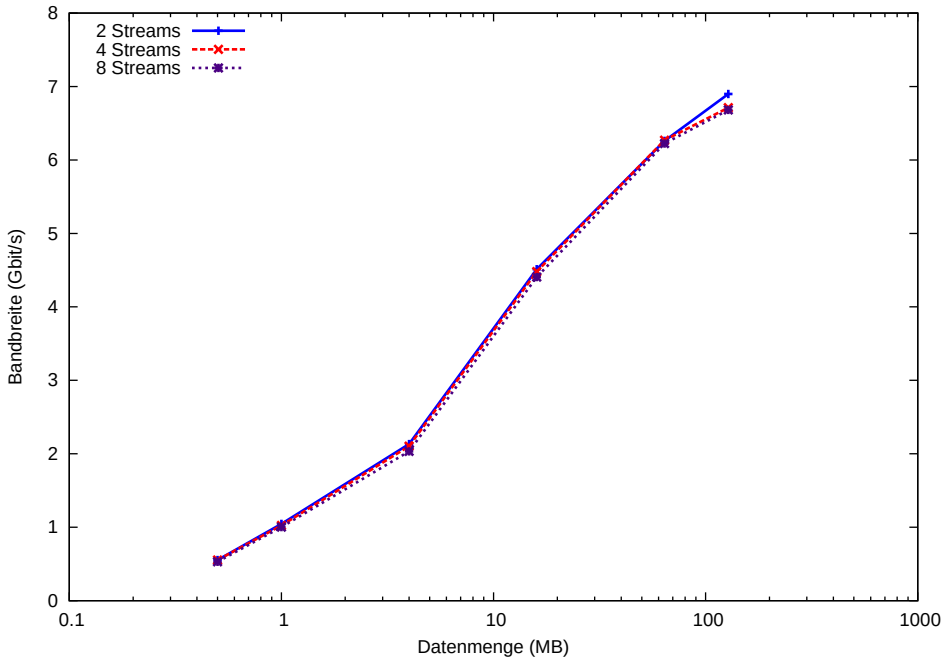

Abbildung 3.9: Unterschiede der Laufzeiten bei 2, 4 und 8 Streams (GTX480).

Abschließend findet sich im Anhang A.2 ein umfassendes Beispiel, dass die oben beschriebenen Funktionen und deren Zusammenspiel verdeutlicht.

3.3.1 Verwendung von Streams

Im Abschn. 2.3.1 wurden die Vorteile der Verwendung von Streams bereits erläutert. Es wird nun näher, im Kontext des GPU-Moduls, darauf eingegangen, wie die einzelnen Arbeitsschritte auf Streams aufgeteilt werden. Wie bereits erwähnt, können durch die Verwendung von asynchronen Funktionen unter Umständen Kernel- und Kopiervorgänge unterschiedlicher Streams überlagert werden. Im Falle des GPU-Moduls sind die Unterschiede zwischen 2, 4 und 8 Streams nicht sehr markant (vgl. Abb. 3.9), aufgrund der Aufteilung der CUDA-Blöcke und Segmente auf die Streams werden 4 Streams eingesetzt[4]. Eine weitere Erhöhung der Streams kann dagegen die Laufzeit nicht mehr erheblich erhöhen . Dies ist wiederum auf den erhöhten Verwaltungsaufwand, die Koordination der Speicherzugriffe und die vorhandene hohe Auslastung der GPU zurückzuführen. Aus der Tatsache, dass die Operationen auf Streams aufgeteilt werden, resultiert auch, dass sich

[4]Zu Beginn wurden außerdem die Tests auf einer GTX285 durchgeführt, auf der 4 Streams die größte Performance erzielten.

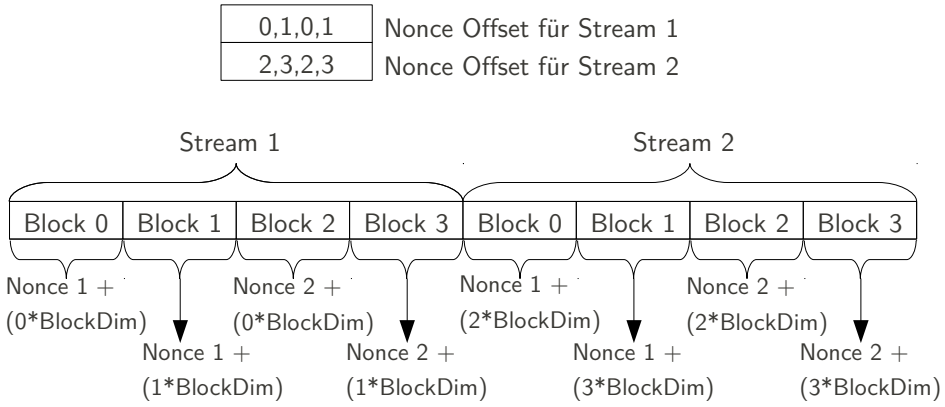

Abbildung 3.10: Das Array „Nonce Offset“ muss für die Verwendung mit Streams angepasst werden. Es wird so erweitert, dass jeder Block eines Streams den richtigen relativen Offset zum übergebenen Nonce addiert.

die Start-Parameter und die berechneten Ergebnisse aufteilen. In diesem Zusammenhang muss auch das verwendete Array „Nonce Offset“ erweitert werden, da ansonsten jeder Stream dieselben Counter verschlüsselt. Nonce Offset wird dabei so erweitert, dass jeder Block eines Streams den richtigen Offset zum übergebenen Nonce addiert. Somit entsteht für jeden Block eine einmalige relative Position zum Nonce und kein Block arbeitet mit demselben Nonce (Abb. 3.10).

Bezüglich der Berechnung der Endergebnisse ergeben sich ebenfalls Änderungen gegenüber einer Variante ohne Streams. Die vom Benutzer angeforderten Segmente werden gleichmäßig auf die Streams aufgeteilt (vgl. Abb. 3.11). Für das Output-Array hat dies zur Folge, dass sich die verschlüsselten Counter, die zu einem Nonce gehören, nicht mehr sequentiell im Speicher befinden. Da jeder Stream abwechselnd seine berechneten Ergebnisse in das Output-Array schreibt, ergibt sich eine Fragmentierung. Abb. 3.12 stellt grafisch dar, wie die verschlüsselten Counter im Zusammenhang mit den SAs im Speicher liegen. Aus den Färbungen in der Grafik lässt sich erkennen, dass sich immer $\frac{1}{4}$ der angeforderten Segmente einer SA, gefolgt vom ersten Viertel der nächsten SA usw., im Speicher befindet. Dieses Muster ist insgesamt vier Mal vorhanden, da vier Streams eingesetzt werden.

Für die Anzahl an Segmenten, die angefordert werden können, bedeuten die Streams, dass sie immer ein Vielfaches von 4 sein müssen. Wie in Abschn. 2.3.2 erläutert, ist ein Segment mit 65 KB an verschlüsselten Countern gleichzusetzen. Die minimale Größe von vier Segmenten bedeutet daher, dass von den vier Streams $4 * 65$ KB erzeugt werden und zumindest immer 65 KB sequentiell im Speicher liegen.

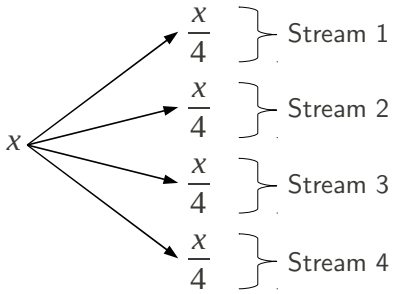

Abbildung 3.11: Die angeforderten x Segmente werden auf Streams aufgeteilt. Bei vier Streams erstellt jeder Stream $\frac{1}{4}$ der Gesamtanzahl an Segmenten.

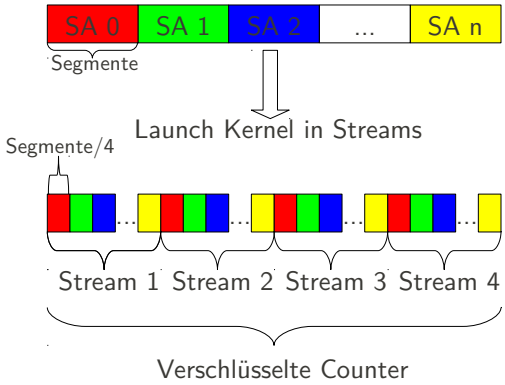

Abbildung 3.12: Die einzelnen Daten der Streams, die zu einem Nonce gehören, werden fragmentiert im Output-Array abgelegt.

3.4 CPU-Modul

Mit dem GPU-Modul steht nun eine Lösung zur Verfügung, mit der für mehrere SAs verschlüsselte Counter berechnet werden können. Wie im letzten Abschnitt geschildert, entsteht dabei ein fragmentierter Speicherbereich, der abwechselnd die einzelnen Teile der SAs enthält. Es stellt sich nun die Frage, wie dieser Speicherbereich weiter verwendet werden kann und wie das Management des GPU-Moduls gehandhabt wird. Es werden nun zwei mögliche Lösungen vorgestellt und analysiert. Die Beschreibung der konzeptionellen Implementierung einer dieser Lösungen wird ebenfalls Teil eines Abschnitts sein. Diese Lösungen werden insofern benötigt, da sich der fragmentierte Speicherbereich nicht für eine performante Verwendung zur Verschlüsselung eignet. Optimal ist ein zusammenhängender Speicher, der sequentiell abgearbeitet werden kann. Dadurch entfällt ein umständliches Umrechnen von Offsets und der Speicher kann, nachdem er aufgebraucht wurde, problemlos freigegeben werden.

Abbildung 3.13: Durch die Aufteilung auf Zeiger, erhält jede SA n Zeiger, die jeweils auf 1 Segment zeigen. n kann pro SA, je nach Anforderung, unterschiedlich sein. Die Zeiger entstehen durch addieren von Offsets zum GPU-Modul gelieferten Zeiger.

3.4.1 Verwaltung des GPU-Moduls

Lösung 1 - Zeiger auf Segmente

Eine Möglichkeit, die Fragmentierung des Speicherbereichs zu entfernen, ist die Aufteilung auf einzelne Zeiger. Folgende Schritte werden dabei durchgeführt:

1. Die GPU liefert einen Zeiger auf alle Segmente aller SAs.
2. Dieser Zeiger wird nun in Zeiger auf Segmente, den richtigen SAs zugehörig, aufgeteilt. Da bekannt ist, wie viele Segmente einer SA angefordert worden sind, sind auch die Offsets der Startpositionen der Speicherbereiche einer SA berechenbar. Eine Aufteilungs-Funktion kann somit, direkt nach dem Aufruf des GPU-Moduls, den fragmentierten Speicher auf Zeiger auf Segmente aufteilen.
3. Im Detail werden dabei für eine SA, die n Segmente angefordert hat, n Zeiger erstellt. Jeder dieser Zeiger zeigt dabei auf ein Segment (vgl. Abb. 3.13).

Nach der Aufteilung auf Zeiger wird eine Komponente benötigt, die verbrauchte und neu generierte Segment-Zeiger verwaltet. Die Zeiger auf Segmente könnten z. B. in einem Ring-Puffer verwaltet werden. Dies würde bedeuten, dass die neu generierten Segment-Zeiger die bereits benutzten im Ring-Puffer ersetzen. Werden verschlüsselte Counter für ein XOR mit einem Plaintext benötigt, so kann ein Segment aus dem Ring-Puffer genommen werden. Damit können 65 KB verschlüsselt werden. Reicht diese Datenmenge nicht aus, so kann auf das nächste Segment im Ring-Puffer zugegriffen werden. Währenddessen ersetzt die GPU die bereits verwendeten Segment-Zeiger wieder durch neue Zeiger (s. auch Abb. 3.14). Innerhalb eines Segments gilt es noch den richtigen Counter für das XOR zu finden, da kein Counter ein zweites Mal verwendet werden darf. Es muss daher sicher gestellt sein, dass, wenn alle Counter eines Segments aufgebraucht wurden, zum nächsten Segment im Ring-Puffer gewechselt wird.

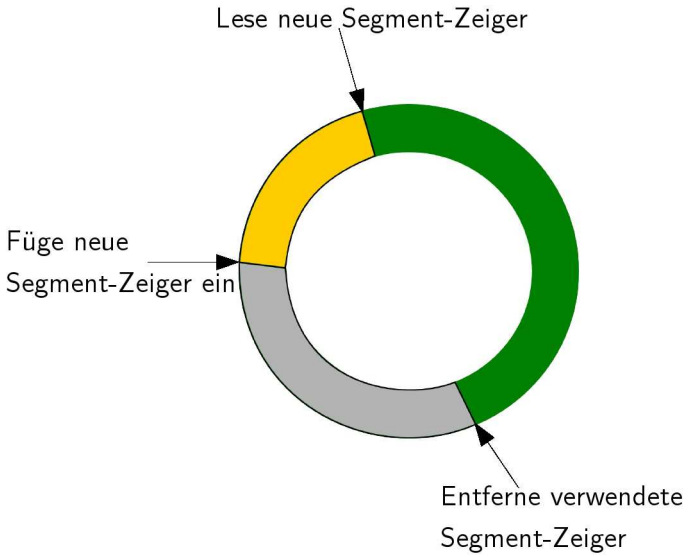

Abbildung 3.14: Ein Ring-Puffer könnte für die Verwaltung der Segment-Zeiger verwendet werden. Während die Applikation für die Verschlüsselung Segmente verbraucht, füllt die GPU den Puffer wieder mit neuen Zeigern.

Nachteile der Segment-Zeiger: Die Nachteile der Segment-Zeiger ergeben sich aus der exzessiven Verwendung von Zeiger-Arithmetik und bei der Freigabe des verwendeten Speicherbereichs. Da die einzelnen Segment-Zeiger aus dem einen Zeiger des GPU-Moduls durch addieren von Offsets erzeugt werden, können diese Zeiger nicht mehr freigegeben werden. D. h. der Speicherbereich kann nur durch ein `free` auf den Zeiger des GPU-Moduls durchgeführt werden. Für die Freigabe eines Zeigers ist im C-Standard Folgendes definiert [1, S. 313]:

> The free function causes the space pointed to by ptr to be deallocated, that is, made available for further allocation. If ptr is a null pointer, no action occurs. Otherwise, if the argument does not match a pointer earlier returned by the calloc, malloc, or realloc function, or if the space has been deallocated by a call to free or realloc, the behavior is undefined.

Somit können die einzelnen Zeiger der SAs nicht freigegeben werden, da sie nicht durch `malloc`, sondern aus der Addition von Offsets zu einem Zeiger entstanden sind. Gleichzeitig muss solange mit dem Freigeben des Speicherbereichs gewartet werden, bis alle SAs alle Segment-Zeiger aufgebraucht haben! Erst dann kann durch Anwenden eines `free` auf den einen Zeiger des GPU-Moduls, der gesamte Speicher freigegeben werden.

Lösung 2 - Double-Buffer

Aufgrund der Nachteile bei der Verwendung von Segment-Zeigern wird als zweite Lösung die Verwendung eines doppelten Puffer-Systems vorgestellt. Hierbei soll der fragmentierte Speicherbereich des GPU-Moduls aber nicht in Zeiger unterteilt, sondern in vorhandene Puffer der SAs kopiert werden. Der wesentliche Unterschied besteht darin, dass viele Speicherbereiche kopiert werden müssen und nicht mit Zeiger-Arithmetik gearbeitet wird. Um Redundanzen in der Beschreibung der Double-Buffer zu vermeiden wird hier nicht mehr näher darauf eingegangen. Die Details folgen im Zuge der Beschreibung der Implementierung im nächsten Abschnitt.

3.4.2 Implementierung der Double-Buffer

Die Struktur für die Verwaltung der SAs wurde bereits in Abschn. 3.2 beschrieben. Der Datentyp „`double_buffer_t`“, der Teil dieser Struktur ist, wird nun näher erklärt (s. auch Progr. 3.5):

- `uint32_t *buffer_0`: Ein Zeiger auf den ersten Puffer, der die verschlüsselten Counter enthält.
- `unsigned int buffer_0_size`: Die Größe des ersten Puffers, angegeben in der Anzahl an Elementen, die `buffer_0` beinhaltet.
- `uint32_t *buffer_1`: Ein Zeiger auf den zweiten Puffer.
- `unsigned int buffer_1_size`: Die Größe des zweiten Puffers, wieder als Anzahl der Elemente.
- `int buffer_active`: Der Index des aktiven Puffers, der entweder 0 oder 1 sein kann.
- `unsigned int buffer_size`: Die Größe des aktiven Puffers, entweder die Größe von `buffer_0` oder `buffer_1`.
- `unsigned int buffer_used`: Die Anzahl an verschlüsselten Countern des aktiven Puffers, die schon für eine Verschlüsselung verwendet wurden. Erreicht `buffer_used` `buffer_size`, so müssen die Puffer gewechselt werden.
- `unsigned char switch_nonce[16]`: Dieser Nonce wird für den Wechsel der Puffer benötigt. `switch_nonce` wird mit jenem unverschlüsselten Counter initialisiert, der verwendet wurde als `buffer_used` die Größe von `buffer_size` erreicht hat. `switch_nonce` wird dann für das Befüllen des soeben leer gewordenen Puffers benutzt.
- `unsigned char nonce_0[16]`: Der unverschlüsselte Counter, der den Start von `buffer_0` markiert. Mit `nonce_0` und `buffer_used` können alle unverschlüsselten Counter für `buffer_0` erzeugt werden. Diese werden dann als IV im ESP-Paket verwendet.
- `unsigned char nonce_1[16]`: Der unverschlüsselte Counter, der den Start von `buffer_1` markiert. Die Erzeugung der unverschlüsselten

```
1 struct double_buffer{
2   uint32_t *buffer_0;//The first buffer for the encrypted counters
3   unsigned int buffer_0_size;//The size of the first buffer
4   uint32_t *buffer_1;//The second buffer for the encrypted counters
5   unsigned int buffer_1_size;//The size of the second buffer
6   int buffer_active;//The number of the actual buffer, can be either 0 or 1
7   unsigned int buffer_size;//The size of the actual valid buffer
8   unsigned int buffer_used;//The used counters of the actual buffer
9   unsigned char switch_nonce[16];//the nonce at switching the buffers
10  unsigned char nonce_0[16];//plaintext counter for the start of the first buffer
11  unsigned char nonce_1[16];//plaintext counter for the start of the second buffer
12 };
13 typedef struct double_buffer double_buffer_t;
```

Programm 3.5: Struktur für die Verwaltung der Double-Buffer einer SA. Der Datentyp „`double_buffer_t`“ ist ein Teil des Typs „`sa_t`“.

```
1 int init_buffer(sa_t *sa_list);
```

Programm 3.6: Initialisierungsfunktion der Double-Buffer. Alle Variablen von `double_buffer_t` werden mit 0 initialisiert, ausgenommen `buffer_active`, das den Wert -1 erhält, da noch kein Puffer aktiv ist.

Counter für `buffer_1` erfolgt mit `buffer_used` analog zu `buffer_0`.

Mit den Variablen der Struktur „`double_buffer_t`“ sind nun alle Informationen vorhanden, mit denen die Double-Buffer verwaltet werden können. Es wird nun Schritt für Schritt auf die einzelnen Funktionen eingegangen, bis hin zur Ver- und Entschlüsselung. Als erste Funktion wird eine Initialisierungsfunktion für die Double-Buffer vorgestellt (Progr. 3.6). Der Funktion „`init_buffer`“ wird dabei nur ein Parameter übergeben:

- `sa_t *sa_list`: Ein Zeiger auf den Beginn der verketteten Liste der SAs.

`init_buffer` iteriert anschließend über alle SAs und initialisiert die Puffer, sodass im nächsten Schritt mit dem erstmaligen Befüllen der Puffer begonnen werden kann. Beim ersten Füllen der Puffer muss bedacht werden, dass noch beide Puffer leer sind. Es müssen daher genügend Segmente vom GPU-Modul angefordert werden, um beide Puffer ausreichend ausstatten zu können. Die Funktion „`fill_buffers`“ nimmt folgende Parameter entgegen (Progr. 3.7):

- `sa_t *sa_list`: Ein Zeiger auf den Beginn der verketteten Liste der SAs.
- `uint32_t *out`: Ein Zeiger auf ein Array, das die vom GPU-Modul erstellten, verschlüsselten Counter enthält.

```
1 int fill_buffers(sa_t *sa_list, uint32_t *out, size_t out_size);
```

Programm 3.7: Eine Funktion zum erstmaligen Befüllen beider Puffer. Das `out`-Array wurde zuvor vom GPU-Modul erstellt und ist auf `sa_list` abgestimmt.

- `size_t out_size`: Die Größe von `out` in Bytes.

`fill_buffers` nützt dabei die Informationen aus `sa_list` über die zu erstellenden Segmente und kopiert die Fragmente aus `out` in die Puffer der zugehörigen SAs. Es muss dabei zumindest immer vier Mal pro SA die Funktion „`memcpy`“ aufgerufen werden, da vier Streams vom GPU-Modul verwendet wurden. Die erste Hälfte der Segmente füllt `buffer_0` und die zweite `buffer_1`. Da nun die Puffer befüllt sind, müssen noch die Parameter von `double_buffer` aktualisiert werden. Konkret werden dabei folgende Tätigkeiten von `fill_buffers` durchgeführt:

- `buffer_0_size` ist gleich der halben Größe der angeforderten Segmente, jedoch nicht in Bytes sondern in der Anzahl an Elementen, die im Array enthalten sein werden[5].
- `buffer_1_size` ist ebenfalls gleich der halben Größe der angeforderten Segmente, da sie gleichmäßig aufgeteilt wurden.
- `nonce_0` ist gleich dem Nonce, der für den Aufruf des GPU-Moduls verwendet wurde. Dieser Nonce markiert den unverschlüsselten Startwert für `buffer_0`.
- Da `nonce_1` den Startwert für `buffer_1` beinhalten soll, muss hier zu `nonce_0` ein Offset addiert werden. Dieser Offset ist gleich der Größe von `buffer_0` gemessen in der Anzahl an enthaltenen Countern.
- `buffer_active` wird mit 0 initialisiert, da nun `buffer_0` der aktive Puffer ist.
- `buffer_size` wird daher auch gleich `buffer_0_size`.
- `buffer_used` bleibt gleich 0, da noch kein Counter aus `buffer_0` für eine Verschlüsselung verwendet wurde.

Es kann nun der vom GPU-Modul allokierte Speicher in Form des `out`-Arrays freigegeben werden, da bereits alle benötigten Teile in die einzelnen Puffer kopiert wurden. Diese Möglichkeit der Freigabe des Speichers ist ein wesentlicher Vorteil gegenüber der Ring-Puffer-Variante, die in Abschn. 3.4.1 vorgestellt wird.

Abschließend wird eine Funktion vorgestellt, die die bis jetzt beschriebenen Funktionen vereint. D. h. es wird dir Konfigurations-Datei ausgelesen, die Paramenter mittels `merge_sa_data` in Arrays zusammengetragen, die

[5]Somit muss hier durch die Byte-Größe eines Elementes dividiert werden, um die Anzahl zu erhalten.

```
1 int sas_init(char *file, sa_t **sa_list);
```

Programm 3.8: `sas_init` fasst die bist jetzt beschriebenen Funktionen zusammen. Die Funktion kann daher als einfache Schnittstelle für die Initialisierung von SAs, die in der Konfigurations-Datei spezifiziert wurden, gesehen werden.

Rundenschlüssel werden berechnet, das GPU-Modul wird aufgerufen um die Segmente zu erzeugen und schließlich werden mit `fill_buffers` die Puffer befüllt und der allokierte Speicher freigegeben. Im Anhang A.3 findet sich der zugehörige Quellcode zu der Funktion `sas_init` (Progr. 3.8):

- `char *file`: Pfad zur Konfigurations-Datei.
- `sa_t **sa_list`: Ein Zeiger-Zeiger, dem das erste SA-Element der verketteten Liste zugewiesen wird.

3.4.3 Patch der LibTomCrypt-Bibliothek

LibTomCrypt ist eine in C geschriebene Bibliothek für die Entwicklung von kryptografischen Anwendungen [7, S. 1]. Die Bibliothek wird als Public-Domain-Software unter http://libtom.org/ veröffentlicht, von wo auch der Quellcode bezogen werden kann. LibTomCrypt stellt unter anderem Implementierungen von symmetrischen Block-Chiffren, Hash-Funktionen, Pseudo-Zufallszahlen-Generatoren oder auch Public-Key-Kryptographie zur Verfügung. Der Autor der LibtTomCrypt – Tom St. Denis – beschreibt ausführlich in seinem Buch „*Cryptography for Developers*“ [8] die Umsetzung und Implementierung des GCM.

Diese GCM-Implementierung wird in dieser Arbeit so abgeändert, dass für die Verschlüsselung auf die Double-Buffer einer SA zugegriffen wird. Hierzu ein Beispiel, wie der Ablauf einer Verschlüsselung mit GCM in der LibTomCrypt aussieht:

```
1  //Rregister AES
2  register_cipher(&aes_desc);
3  //Init the GCM state
4  if ((err = gcm_init(&gcm, find_cipher("aes"), key, 16)) != CRYPT_OK) {
5    return err;
6  }
7  //Reset the state
8  if ((err = gcm_reset(gcm)) != CRYPT_OK) {
9    return err;
10 }
11 //Add the IV
12 if ((err = gcm_add_iv(gcm, iv, IVLEN)) != CRYPT_OK) {
13   return err;
14 }
15 //Add the AAD
```

```
16  if ((err = gcm_add_aad(gcm, NULL, 0)) != CRYPT_OK) {
17    return err;
18  }
19  //Process the Plaintext
20  if ((err = gcm_process(gcm, pt, len, ct, GCM_ENCRYPT)) != CRYPT_OK) {
21    return err;
22  }
23  //Finish gcm state and get tag
24  if ((err = gcm_done(gcm, tag, &tag_len)) != CRYPT_OK) {
25    return err;
26  }
```

Die einzelnen Funktionen übernehmen hierbei folgende Aufgaben [7, Abschn. 3.5.4][8, S. 304ff]:

- `register_cipher`: Registrierung des verwendeten Chiffre, hier AES.
- `gcm_init`: Initialisierung von `gcm_state gcm` für AES mit dem gegebenen Key und der verwendeten Key-Länge. Die LibTomCrypt bietet außerdem die Möglichkeit GCM mit vorausberechenbaren Lookup-Tabellen zu verwenden. Diese Tabellen werden dann von `gcm_init` für den verwendeten Key berechnet und müssen erst neu berechnet werden, wenn sich der Key ändert [7, Abschn. 13.7.12].
- `gcm_reset`: Setzt `gcm_state gcm` auf den Ausgangszustand, der dem Zustand nach dem Aufruf von `gcm_init` gleicht, zurück.
- `gcm_add_iv`: Nach dem Reset oder der Initialisierung kann der zu verwendende IV hinzugefügt werden. In dieser Arbeit wird dieser IV zur Abgrenzung zum ESP-IV als Nonce bezeichnet.
- `gcm_add_aad`: Fügt die Additional Authentication Data hinzu.
- `gcm_process`: Im Falle von `GCM_ENCRYPT` wird `pt` gelesen, verschlüsselt und in `ct` abgelegt. Bei einer Verwendung von `GCM_DECRYPT` wird das Umgekehrte durchgeführt.
- `gcm_done`: Terminiert `gcm_state gcm` und speichert den Authentication Tag der Länge `tag_len` in `tag`.

Zur einfacheren Verwendung und um sicher zu stellen, dass die Funktionen in der richtigen Reihenfolge aufgerufen werden, werden die oben aufgezeigten Funktionen in eigenen Methoden – `gcm_encrypt` und `gcm_decrypt` – zusammengefasst (Progr. 3.10 und 3.11). In `gcm_decrypt` wird zusätzlich der übergebene und der errechnete Authentication Tag verglichen. Dazu wird außerdem ein neuer Datentyp definiert, der den `gcm_state` der LibTomCrypt und die SA (vom Typ `sa_t`, s. auch Progr. 3.2 auf S. 37) vereint (vgl. Progr. 3.9).

Somit stehen alle Funktionen und Parameter zur Verfügung, die benötigt werden, um den GCM komplett umzusetzen. Das Ziel des Patches ist es, die Funktion „`gcm_process`“ so abzuändern, dass die Double-Buffer einer SA verwendet werden. Der erste Schritt besteht daher darin, die übergebenen Parameter um eine Variable des Typs „`sa_t`“ zu erweitern (Progr. 3.12).

```
1 struct sa_context{
2   sa_t *sa;
3   gcm_state gcm;
4 };
5 typedef struct sa_context sa_context_t;
```

Programm 3.9: Der GCM-Status und die SA werden in einem eigenen Datentyp zusammengefasst. Dieser wird später bei der Ver- und Entschlüsselung zur Anwendung kommen.

```
1 int gcm_encrypt(sa_context_t *sa_context, unsigned char *pt, unsigned
      char *ct, unsigned int len,
2     unsigned char *aad, unsigned int alen, unsigned char *tag, unsigned
      long taglen)
```

Programm 3.10: Die einzelnen Funktionen der LibTomCrypt werden in einer Methode zusammengefasst. Dieser müssen anschließend alle Parameter – inkl. SA – übergeben werden, die zur Verschlüsselung benötigt werden.

```
1 int gcm_decrypt(sa_context_t *sa_context, unsigned char *pt, unsigned
      char *ct, unsigned int len,
2     unsigned char *aad, unsigned int alen, unsigned char *tag, unsigned
      long taglen)
```

Programm 3.11: Die Funktion `gcm_decrypt` übernimmt neben der Entschlüsselung von Daten auch die Überprüfung des Authentication Tags.

Es kann nun innerhalb von `gcm_process` auf die Puffer von `sa_list_item` zugegriffen werden. Im nächsten Schritt müssen jene Code-Abschnitte in `gcm_process.c` gefunden und geändert werden, die die Verschlüsselung der Counter durchführen. Diese Abschnitte können einfach aufgefunden werden, da nur nach einem Aufruf der Verschlüsselungs-Funktion des Ciphers gesucht werden muss. In der originalen Version der LibTomCrypt sieht die

```
1 int gcm_process(gcm_state *gcm,
2                     unsigned char *pt,     unsigned long ptlen,
3                     unsigned char *ct,   sa_t *sa_list_item,
4                     int direction);
```

Programm 3.12: Die bestehende Funktion `gcm_process` wird um einen Parameter – `sa_t *sa_list_item` – erweitert. Diese Änderung wird in der Datei `tomcrypt_mac.h` durchgeführt.

Verschlüsselung eines Counters wie folgt aus:

```
1 /* encrypt the counter */
2 if ((err = cipher_descriptor[gcm->cipher].ecb_encrypt(gcm->Y, gcm->buf,
      &gcm->K)) != CRYPT_OK) {
3   return err;
4 }
```

Es wird also der Nonce, der in `gcm->Y` enthalten ist, mit `gcm->K` verschlüsselt und in `gcm->buf` abgelegt. Diese Zeile wird nun durch das Code-Stück aus Progr. 3.13 ersetzt. Dieser Code führt Folgendes durch:

1. Zuerst wird überprüft, ob die benutzten Counter bereits die Größe des aktiven Puffers erreicht haben (Zeile 2). Ist `buffer_used` bereits größer als die Größe des aktuellen Puffers, so wird `switch_buffer` aufgerufen. Die Switch-Funktion aktualisiert die Variablen auf die des neuen Puffers (`buffer_active`, `buffer_size`, `buffer_used`) und kopiert den Nonce zum Zeitpunkt des Wechsels nach `switch_nonce`, der später für das Wieder-Befüllen benötigt wird (vgl. S. 57). Der Return-Wert der Switch-Funktion ist ein Zeiger auf den Puffer, zu dem gewechselt wurde.
2. Sind beide Puffer nicht leer, wird der aktive Puffer mit der Funktion „`get_buffer`“ der Variable `ct_buf` zugewiesen (Zeile 8). Wird von `get_buffer` `NULL` zurückgegeben, dann sind beide Puffer leer und es muss mit der CPU verschlüsselt werden (Zeile 12). Dieser Fall sollte jedoch nie eintreten, da sobald ein Puffer leer geworden der andere verwendet werden kann.
3. Schließlich wird mit `buffer_used` der nächste unbenutzte Counter im aktiven Puffer aufgefunden und kopiert (Zeile 18).
4. Der soeben kopierte Counter muss noch in den richtigen Datentyp umgewandelt und in `gcm->buf` geschrieben werden (ab Zeile 20).

Für eine Verschlüsselung eines Counters wird somit nur mehr ein `memcpy` benötigt, solange die Double-Buffer der verwendeten SA nicht leer sind. Die Funktionen der ursprünglichen LibTomCrypt können weiter verwendet werden, vor allem die Berechnung des Authentication Tags wird von der Bibliothek übernommen. Wie bereits erwähnt braucht der Funktion `gcm_process` nur mehr eine `sa_t` Variable übergeben werden, deren Puffer zuvor befüllt wurden.

Im nächsten Kapitel wird nun darauf eingegangen, wie eine effiziente Abstimmung und ein Zusammenspiel von CPU und GPU erfolgen kann.

```
1     //Check if there are unused counters
2       if((sa_list_item->double_buffer.buffer_used) > ((sa_list_item->
      double_buffer.buffer_size) / 4)){
3         ct_buf = switch_buffer(sa_list_item);
4         //as we skipped one time during the switch function we must increase now
5         (sa_list_item->double_buffer.buffer_used)++;
6       }
7       //we try to get an active buffer
8       ct_buf = get_buffer(sa_list_item);
9
10      //if we don't have a valid buffer from the gpu we use the CPU
11      if(ct_buf == NULL){
12            if ((err = cipher_descriptor[gcm->cipher].ecb_encrypt(gcm->Y
      , gcm->buf, &gcm->K)) != CRYPT_OK) {
13          return err;
14        }
15      }
16      else{
17        //we copy the right encrypted counter from the buffer to helper
18        memcpy(helper, ct_buf + ((sa_list_item->double_buffer.
      buffer_used) * 4), 4 * sizeof(uint32_t));
19        //we convert our helper to fit to buf
20        gcm->buf[0] = (helper[0] & 0xff000000) >> 24;
21        gcm->buf[1] = (helper[0] & 0x00ff0000) >> 16;
22        gcm->buf[2] = (helper[0] & 0x0000ff00) >> 8;
23        gcm->buf[3] = (helper[0] & 0x000000ff);
24        gcm->buf[4] = (helper[1] & 0xff000000) >> 24;
25        gcm->buf[5] = (helper[1] & 0x00ff0000) >> 16;
26        gcm->buf[6] = (helper[1] & 0x0000ff00) >> 8;
27        gcm->buf[7] = (helper[1] & 0x000000ff);
28        gcm->buf[8] = (helper[2] & 0xff000000) >> 24;
29        gcm->buf[9] = (helper[2] & 0x00ff0000) >> 16;
30        gcm->buf[10] = (helper[2] & 0x0000ff00) >> 8;
31        gcm->buf[11] = (helper[2] & 0x000000ff);
32        gcm->buf[12] = (helper[3] & 0xff000000) >> 24;
33        gcm->buf[13] = (helper[3] & 0x00ff0000) >> 16;
34        gcm->buf[14] = (helper[3] & 0x0000ff00) >> 8;
35        gcm->buf[15] = (helper[3] & 0x000000ff);
36      }
```

Programm 3.13: Die Verschlüsselungen der Counter in der `process`-Funktion werden durch die vorausberechneten Counter aus den Double-Buffern ersetzt.

Kapitel 4

Testsystem und Evaluierung

Kapitel 4 behandelt die Herausforderungen und Schwierigkeiten bei der Umsetzung von AES-GCM-ESP mit GPUs als Coprozessoren. Im ersten Teil steht im Vordergrund, wie eine parallele Implementierung von Paket-Verarbeitung und Puffer-Befüllen umgesetzt werden kann. Zusätzlich dazu wird auf die Problemfälle eingegangen, die bei der Kommunikation zwischen zwei Systemen auftreten können, wenn diese Counter, mit der GPU oder anderen Coprozessoren, vorausberechnen.

Im zweiten Teil werden die Geschwindigkeits-Änderungen des LibTomCrypt-Patches analysiert. Dabei wird vor allem wichtig sein, ob der Patch Vor- oder Nachteile im Vergleich zur reinen CPU-Lösung mit sich bringt.

4.1 Funktionalität

Der Abschnitt „Funktionalität“ beschreibt die Implementierung einer bidirektionalen, verschlüsselten Verbindung zwischen zwei Systemen über virtuelle Netzwerkadapter. Diese Applikation wird eine IPsec-Verbindung simulieren und zeigen, wie CPU- und GPU-Modul (aus dem vorhergehenden Kapitel) zusammen agieren können. Die in den nächsten Abschnitten beschriebene Applikation wird von nun an auch als „`cuvpnapp`“ bezeichnet. Um zu Beginn einen Überblick zu bekommen, wird in Abb. 4.1 der schematische Aufbau der `cuvpnapp`-Architektur dargestellt. Darin enthalten sind die folgenden zentralen Teile:

1. **TUN-/TAP-Devices**: TUN-/TAP-Devices bieten eine einfache Möglichkeit, um Netzwerk-Pakete von Applikationen zu verarbeiten [19]. Ein TUN-/TAP-Device fungiert als einfaches Punkt-zu-Punkt- oder Ethernet-Device. Der Unterschied zu physischen Netzwerk-Adaptern besteht darin, dass die virtuellen Devices Pakete nicht von physischen Medien, sondern von User-Space-Applikationen empfangen. Im umgekehrten Weg werden die Daten von den virtuellen Devices nicht

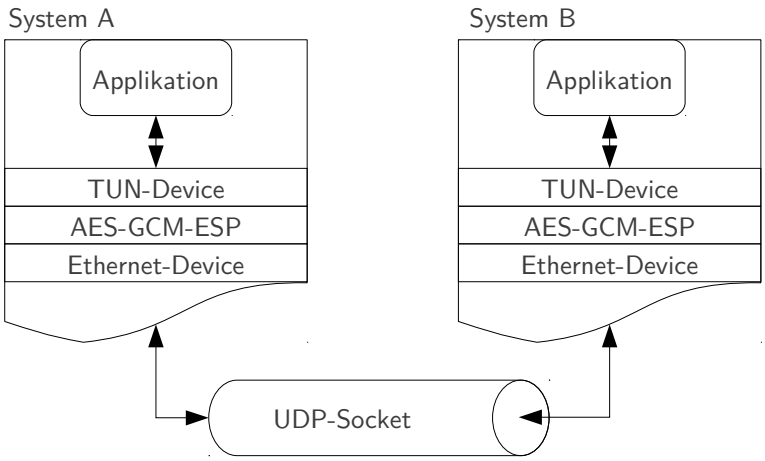

Abbildung 4.1: Aufbau der verschlüsselten Kommunikation zwischen zwei Systemen. Die Applikationen verwenden ein virtuelles Device, welches die Daten mit AES-GCM-ESP verarbeitet und über eine Socket-Verbindung an den Empfänger weiterleitet.

über den physischen Adapter gesendet, sondern an eine User-Space-Applikation weitergegeben. Abhängig vom Typ des virtuellen Devices werden von der User-Space-Applikation entweder IP-Pakete (TUN-Device) oder Ethernet-Frames (TAP-Device) gelesen bzw. geschrieben. Im Falle der `cuvpnapp` kommt ein TUN-Device zum Einsatz, das IP-Pakete zur Ver- bzw. Entschlüsselung an die `cuvpnapp` weitergibt. Konkret werden folgende Operationen auf ein- bzw. ausgehende Pakete angewendet:

(a) Liegen am TUN-Device Pakete zur Abholung bereit, so liest die `cuvpnapp` die Daten vom TUN-Device und verarbeitet diese mit AES-GCM-ESP. Dabei kommt das im vorhergehenden Kapitel vorgestellte GPU-Modul zum Einsatz. Wurde das Paket erfolgreich verschlüsselt und der Authentication Tag berechnet, so wird es über eine Socket-Verbindung an den Kommunikations-Partner gesendet.

(b) Kommt am Socket ein verschlüsseltes Paket an, so muss dieses zuerst entschlüsselt werden. Dazu liest die `cuvpnapp` die Daten vom Socket, entschlüsselt sie und überprüft den Authentication Tag. Erst wenn erfolgreich entschlüsselt und der erstellte Tag mit jenem im Paket übereinstimmt, kann das Paket wieder in das TUN-Device geschrieben werden. So erhält das Ziel-Programm wieder ein unverschlüsseltes Paket.

2. **UDP-Sockets:** Über eine Socket-Verbindung werden die verschlüsselten Pakete zwischen den beiden Systemen ausgetauscht. Auf Protokoll-

Ebene werden verschlüsselte IP-Pakete in User Datagram Protocol (UDP)-Pakete gekapselt und dann verschickt. Die Verwendung von UDP bietet dabei einige Vor-, aber auch gewisse Nachteile. Zu den Vorteilen gehört unter anderem eine höhere Geschwindigkeit, da im Gegensatz zu verbindungsorientierten Protokollen – wie z. B. dem Transmission Control Protocol (TCP) – zum Senden eines Pakets keine explizite Verbindung aufgebaut werden muss [12, S. 134]. Der Aufwand, der für den Aufbau einer Verbindung benötigt wird, fällt somit weg. In puncto Flexibilität kann hervorgehoben werden, dass mit jeder Sende-Operation ein anderer Adressat angesprochen werden kann und auch die Möglichkeit von Broadcasts besteht.

Neben den Vorteilen existieren aufgrund der Verbindungslosigkeit von UDP auch mehrere Nachteile, die das Protokoll nicht für jedes Einsatz-Szenario geeignet erscheinen lassen. Der gravierendste Nachteil ergibt sich aus der Tatsache, dass keine Garantie besteht, dass ein UDP-Paket auch beim Empfänger ankommt. Das Paket kann auf dem Weg zum Empfänger verloren gehen – z. B. von einem Router verworfen werden – ohne Verständigung des Senders. Ein weiteres Problem beim Versenden von UDP-Nachrichten ist die Vertauschung von Paketen. Werden zwei UDP-Pakete nacheinander versendet, so wird von UDP nicht garantiert, dass diese beiden Nachrichten auch in der gesendeten Reihenfolge beim Empfänger ankommen [12, S. 135].

4.1.1 Abläufe der `cuvpnapp`

Die `cuvpnapp` vereint CPU- und GPU-Modul in einer Applikation und nutzt TUN-Device und UDP-Socket für den Austausch der Daten mit dem Kommunikationspartner. Dabei können Daten ver- bzw. entschlüsselt und verschickt werden, während parallel die Puffer der SAs von der GPU wieder befüllt werden. Für eine gesicherte Kommunikation zwischen zwei Systemen (z. B. einem Client und einem Server) wird die `cuvpnapp` in Form eines Kommandozeilen-Programms aufgerufen. Zu Beginn wird dabei noch zwischen Client und Server unterschieden, der spätere Code-Ablauf für die Ver- und Entschlüsselung ist jedoch auf beiden Seiten gleich.

Für den Aufruf der Applikation im Server-Modus werden der `cuvpnapp` folgende Parameter mitgegeben:

```
sudo ./cuVPNapp -s2022 -iInitString
```

Der Parameter „`-s`“ gibt dabei den Port an, auf dem der Server auf eingehende Verbindungen warten soll – im Falle des Beispiels Port 2022. Der Parameter nach „`-i`“ dient für die Initialisierung der Verbindung zwischen Client und Server. Die Zeichen, die nach `-i` folgen werden unverschlüsselt zwischen Client und Server ausgetauscht, um einen ersten Erreichbarkeitstest durchzuführen. Auf der Client-Seite wird ebenfalls die `cuvpnapp` gestartet, jedoch

mit unterschiedlichen Parametern. Am Client wird die IP-Adresse und der Port des Servers sowie der vereinbarte Init-String angegeben:

```
sudo ./cuVPNapp -c10.0.0.2:2022 -iInitString
```

Nach dem Start der `cuvpnapp` wird das TUN-Device erstellt und auf seitens des Servers der UDP-Socket für die Kommunikation angelegt. Bevor das TUN-Device von der `cuvpnapp` verwendet werden kann, muss ihm zuvor noch eine IP-Adresse zugewiesen werden. Das virtuelle Device lässt sich hierbei wie ein übliches Netzwerk-Device konfigurieren[1]:

```
sudo ifconfig tun0 192.168.0.1 netmask 255.255.255.0
```

Alle Netzwerk-Pakete von Applikation, die an Adressen im 192.168.0.0/24-Bereich adressiert sind, werden jetzt vom Linux-Kernel an das TUN-Device übergeben und anschließend von der `cuvpnapp` verarbeitet. Auf den nächsten Seiten folgt detailliert, wie diese Verarbeitung aussieht. Da dies auf Server- und Client-Seite analog durchgeführt wird, wird auf eine weitere Unterscheidung der beiden Seiten verzichtet.

Zu Beginn wartet der Server solange, bis er vom Client den Init-String empfängt. Wurde der Init-String erfolgreich ausgetauscht, so kann auf beiden Seiten mit dem ersten Befüllen der Puffer begonnen werden. Dazu wird die Funktion `sas_init` (vgl. S. 47) mit der entsprechenden Konfigurations-Datei aufgerufen. Auch die Initialisierung von AES-GCM kann durchgeführt werden, dazu wird die Funktion `gcm_init` aufgerufen [7, S. 35]. Zusammengefasst wurden bis zum jetzigen Zeitpunkt folgende Schritte durchgeführt:

- Anlegen des virtuellen TUN-Devices,
- Erstellung der Sockets,
- Austausch des Init-Strings,
- erstmaliges Befüllen der Puffer und
- Initialisierung von AES-GCM.

Um ein effizientes Zusammenspiel zwischen Lesen vom TUN-Device, Schreiben auf den Socket und Befüllen der Puffer zu gewährleisten, muss mit nicht blockierenden Funktionen bzw. Multiplexing gearbeitet werden. Bezüglich Multiplexing muss es eine Möglichkeit geben, zu überprüfen, ob Daten vom TUN-Device zur Verschlüsselung oder vom Socket zur Entschlüsselung gelesen werden müssen. Da sich Sockets und TUN-Devices über File-Descriptors verwalten lassen ([12, S. 10]) wird für diese Entscheidungsfindung die `select`-Funktion verwendet. Mit `select` können mehrere Descriptors überwacht und auf Aktivitäten geprüft werden [6, S. 102ff].

Im Falle der `cuvpnapp` werden zuerst das TUN-Device (`fd`) und der Socket (`s`) zu einem Descriptor-Set hinzugefügt:

```
1 FD_ZERO(&fdset);//set the fd set bits to 0
```

[1] Die IP-Adresse muss jeweils am Server und am Client konfiguriert werden und unterschiedlich sein.

```
2 FD_SET(fd, &fdset);//add the fd for the tun device to the set
3 FD_SET(s, &fdset);//add the fd for the socket to the set
```

Im nächsten Schritt wird mittels `select` jener Descriptor geflaggt, der Daten bereit hält:

```
1 //we want to find out which fd is ready to read or write data
2 if(select(fd+s+1, &fdset,NULL,NULL,NULL) < 0){
3   perror("select");
4   close(fd);
5   close(s);
6   exit(-1);
7 }
```

Mit `FD_ISSET` kann dann der Descriptor, der geflaggt wurde, ausgewählt werden. Beim TUN-Devices müssen die gelesenen Daten verschlüsselt und über den Socket verschickt werden[2]:

```
1 //read from TUN-Device
2 if(FD_ISSET(fd, &fdset)){
3   l = read(fd, package, sizeof(package));
4
5   //encrypt packet
6   gcm_encrypt(sa_context,package,enc_package,l,0,0,tag,TAG_LEN);
7   //copy tag after packet and IV
8   memcpy(enc_package+l+16,tag,TAG_LEN);
9
10  //sent data over socket
11  l_sent = send(s, enc_package, l+16+TAG_LEN, 0);
12 }
```

Im Falle des Sockets werden die Daten gelesen, entschlüsselt und in das TUN-Device geschrieben:

```
1 //if there is nothing to read from TUN-Device, read from socket
2 if(FD_ISSET(s, &fdset)) {
3   l = recv(s, enc_package, sizeof(enc_package), 0);
4
5   //decrypt packet
6   gcm_decrypt(sa_context_in,package,enc_package,l,0,0,tag,TAG_LEN);
7   //Compare tag!
8   if((memcmp(enc_package+(l-TAG_LEN),tag,TAG_LEN))!=0){
9     printf("\nTag of package did not match with tag in encrypted package
      !!!\n");
10    exit(-1);
11  }
12
13  //write the received buffer to the TUN-Device
14  l_recv = write(fd, package, l-16-TAG_LEN);
```

Da in jedem Durchlauf von `gcm_encrypt` bzw. `gcm_decrypt` die Double-Buffer schrittweise geleert werden, müssen parallel zur Ver- und Entschlüsselung eigene Threads gestartet werden, die die Puffer befüllen. Die `cuvpnapp`

[2] Aus Gründen der Übersichtlichkeit wird in den folgenden Code-Beispielen die Fehlerbehandlung vernachlässigt.

besitzt daher einen Haupt-Thread und zwei Füller-Threads – einen für die eingehende und einen für die ausgehende SA. Diese beiden Threads werden vom Haupt-Thread benachrichtigt, wenn ein Puffer einer SA leer ist. Für die Implementierung der Threads wird auf POSIX Threads [20, S. 53] zurückgegriffen.

Im ersten Schritt werden die globalen Variablen definiert, mit denen die Threads miteinander kommunizieren und sich synchronisieren:

- `pthread_mutex_t buffer_mutex`: `buffer_mutex` ist ein Mutual Exclusion Lock (Mutex) [20, S. 106] das verhindert, dass Haupt- und Füller-Thread gleichzeitig auf `global_buffer` und `global_size` zugreifen. `buffer_mutex` wird außerdem dazu verwendet, um die Variable `buffer_empty`, geschützt durch das Lock, zu ändern.
- `pthread_cond_t buffer_cond`: `buffer_cond` ist eine sogenannte „Condition Variable“. Diese Variablen werden immer in Verbindung mit einem Mutex verwendet, unter dessen Schutz die Variable geprüft wird. Ist die Bedingung wahr, so kann der Thread den folgenden Code abarbeiten. Ist die Bedingung nicht wahr, wechselt der Thread in den Sleep-Zustand [20, S. 117]. Aus einem anderen Thread heraus kann dann, wenn sich die Bedingung wieder geändert hat, mit `pthread_cond_signal` der schlafende Thread „geweckt“ werden. Im Falle der `cuvpnapp` weckt der Haupt-Thread den Füller-Thread, wenn einer der Double-Buffer einer SA leer geworden ist.
- `unsigned char global_nonce[16]`: Dieser Nonce wird vom Füller-Thread als Startwert zum Befüllen des leer gewordenen Puffers verwendet. Der `global_nonce` ergibt sich aus dem Nonce zum Zeitpunkt des Wechsels der Puffer (`switch_nonce`) und der Größe des aktuellen Puffers.
- `uint32_t *global_buffer`: Der Füller-Thread befüllt `global_buffer` (ein Array mit Countern), welches im Anschluss vom Haupt-Thread entweder nach `buffer_0` oder `buffer_1` kopiert wird, je nachdem welcher Puffer leer ist. Alle Füll-Operationen des globalen Puffers sowie die Änderung der Variable `global_size` werden unter dem Schutz von `buffer_mutex` vollzogen.
- `size_t global_size`: Die aktuelle Größe von `global_buffer`.

Wie das Zusammenspiel der oben genannten Variablen aus der Sicht des Haupt-Threads aussieht, wird in Abb. 4.2 dargestellt. In dieser Abbildung taucht auch die Funktion „`pthread_mutex_trylock`“ auf. Diese Funktion ist die non-blocking Variante von `pthread_mutex_lock`, d. h. der aufrufende Prozess wird nicht blockiert bis er das Lock erhält, sondern er kann den Return-Wert von `pthread_mutex_trylock` auswerten [20, S. 132]. Die Funktion gibt 0 zurück falls erfolgreich gelockt werden konnte, `EBUSY` wenn ein Thread bereits das Mutex belegt hat, `EINVAL` wenn das Mutex noch nicht

```
1 //Initialize variables...
2 while (!buffer_empty) {
3   err = pthread_cond_wait(&buffer_cond, &buffer_mutex);
4   if (err){
5     printf("\nFiller: condwait failed, err=%d\n",err);
6     pthread_mutex_unlock(&buffer_mutex);
7     exit(1);
8   }
9 }
10 //Call GPU and copy result to global buffer, set buffer_empty=0
```

Programm 4.1: Der Füller-Thread verweilt solange im Sleep-Zustand, bis er vom Haupt-Thread das Signal zum Befüllen erhält.

initialisiert wurde und `EFAULT` falls der übergebene Zeiger nicht valide ist[3].

Die Verwendung dieser Funktion ist für den Haupt-Thread insofern wichtig, da dieser nicht solange warten soll bis der Füller-Thread das Mutex freigibt. Stattdessen führt der Haupt-Thread die Abarbeitung der Daten aus dem TUN-Device und dem Socket fort und prüft erst im nächsten Schleifendurchlauf wieder, ob der Füller-Thread seine Tätigkeiten beendet hat. Dadurch wird sicher gestellt, dass der Haupt-Thread nicht durch den Füller-Thread blockiert wird.

Aus Sicht des Füller-Threads wird solange im Sleep-Zustand gewartet, bis der Haupt-Thread `pthread_cond_signal` ausführt (s. auch Progr. 4.1). Somit wird dem Füller-Thread signalisiert, dass ein Puffer leer geworden ist und die globalen Variablen für das Befüllen verwendet werden können. Der Füller-Thread ruft dann mit den Parametern aus den globalen Variablen das GPU-Modul auf. Wurde das Ergebnis berechnet, werden die verschlüsselten Counter in den globalen Puffer kopiert und `buffer_empty` auf 0 gesetzt. Danach wird im nächsten Schleifendurchlauf von `pthread_cond_wait` das Mutex automatisch wieder freigegeben.

Durch die Verwendung von separaten Füller-Threads ergibt sich eine einfache Möglichkeit, mit der ein leerer Puffer effizient befüllt werden kann. Die Anforderung des Nicht-Blockierens des Haupt-Threads durch den Füller-Thread, wird mit der regelmäßigen Überprüfung durch die nicht-blockierende Funktion `pthread_mutex_trylock` umgesetzt. In puncto Performance treten nur dann Probleme auf, wenn schneller Pakete ver- bzw. entschlüsselt werden, als ein Puffer wieder befüllt werden kann. Im nächsten Abschnitt wird des Weiteren beschrieben, wie die Double-Buffer bei Ver- und Entschlüsselung eingesetzt werden.

[3] S.a http://linux.die.net/man/3/pthread_mutex_trylock.

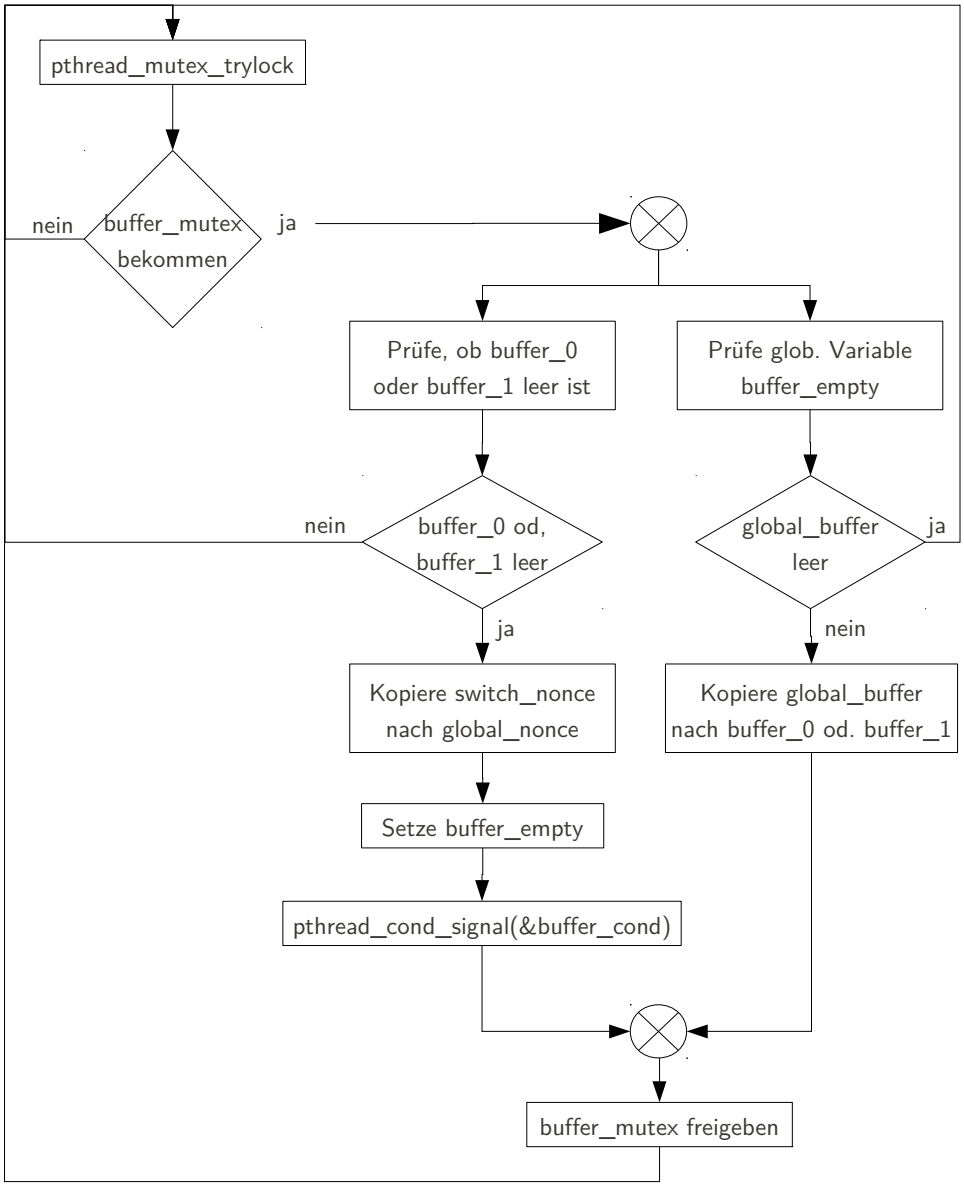

Abbildung 4.2: Abläufe im Haupt-Thread zur Überprüfung, ob einer der Double-Buffer gefüllt werden muss oder ob der globale Puffer vom Füller-Thread bereits befüllt wurde.

4.1.2 Fehlerbehandlung mit Double-Buffer

Da es bei der Übertragung der Netzwerk-Pakete über den UDP-Tunnel zum Verlust oder der Vertauschung von Paketen kommen kann, muss es eine Möglichkeit geben diese Fehler mit den Double-Buffern abzufangen. Der Verlust von Paketen beim Transport zum Empfänger erfordert kein Eingreifen

der `cuvpnapp`. Die verlorenen Pakete müssen ein weiteres Mal übertragen und somit auch noch einmal verschlüsselt werden. Bei dieser Verschlüsselung kommen wieder neue Counter zum Einsatz. Das Paket welches verloren ging hat keinen Einfluss auf die erneute Verschlüsselung.

Anders sieht dies bei der Vertauschung von Paketen aus. Aufgrund der Verwendung von UDP kann es dazu kommen, dass bei der Ankunft eines Pakets erst der richtige Counter im Double-Buffer gesucht werden muss. Da in der Variable `sa→nonce` stets der aktuelle Counter mitgeführt wird, kann der benötigte Offset im Double-Buffer einfach gefunden werden. Kommt ein Paket über den UDP-Socket an, ist in diesem immer auch der Start-Nonce, der zur Verschlüsselung verwendet wurde, enthalten (vgl. Abb. 3.2 auf S. 29). Zum besseren Verständnis wird dieser Nonce von nun an als auch „`packet_nonce`“ bezeichnet. Dieser `packet_nonce` wird jedes Mal, wenn ein Paket am Socket eintrifft, mit dem aktuellen Nonce der SA verglichen. Dazu werden die beiden Nonces zuerst der Funktion „`compare_nonces`“ übergeben. `compare_nonces` vergleicht die beiden Nonces und gibt zurück, ob der erste oder der zweite Nonce größer ist oder ob beide gleich sind. Sind die Nonces aus dem Paket und der SA gleich dann kann ohne weitere Tätigkeiten das Paket entschlüsselt werden. Sind die beiden Nonces aber nicht gleich, muss deren Unterschied in Anzahl an Countern berechnet werden. Diese Unterschieds-Berechnung wird von der Funktion „`diff_nonces`“ übernommen.

Mit der Differenz zwischen `sa→nonce` und `packet_nonce` kann schließlich die Variable `buffer_used` des Double-Buffers so angepasst werden, dass der richtige Counter für die Entschlüsselung verwendet wird. Die Vertauschung von Paketen bei der Übertragung ist daher in den meisten Fällen kein Problem mehr, da über die Variable `buffer_used` einfach der richtige Counter im Puffer gefunden werden kann. Problematisch kann es dann werden, wenn genau nach dem Wechsel der Puffer Pakete vertauscht werden und somit eigentlich Counter aus dem gerade leer gewordenen Puffer benötigt werden. Da dieser Puffer sofort wieder von der GPU befüllt wird, kann nicht mehr zurück gewechselt werden. Um diesen Sonderfall zu behandeln wird eine Überlappung der beiden Puffer eingeführt. D. h. wenn ein leerer Puffer wieder befüllt wird, werden eine gewisse Anzahl an Countern des zuvor gültigen Puffers noch einmal berechnet. Dadurch werden zwar Berechnungen doppelt durchgeführt, dafür kann für einen gewissen Bereich auf ältere Counter zurückgegriffen werden. Mit dieser Architektur können nur in ganz seltenen Fällen eingehende Pakete nicht entschlüsselt werden und sollte es doch dazu kommen, kann immer noch auf die CPU zurückgegriffen werden.

4.2 Performance

Der zweite Teil von Kapitel 4 analysiert die Auswirkungen des GPU-Patches (Abschn. 3.4.3) auf die Performance des LibTomCrypt AES-GCM-Algorithmus. Die Tests wurden, wenn nicht explizit anders angegeben, ausschließlich auf folgender Hard- und Software-Plattform durchgeführt:

- Hardware
  - Extreme Series DX58SO[4]: Hervorzuheben bei dem Motherboard ist der PCI Express 2.0×16 Erweiterungs-Slot für die GPU, da dieser höhere Transferraten, als ein PCI Express 1.1×16 Slot, garantiert.
  - CPU: Intel Core i7 940[5]
  - RAM: 3×4GB 1333MHz DDR3 Non-ECC CL9[6]
  - Hard-Disk: Barracuda 7200.12
  - GPU: Nvidia GeForce GTX480[7]. Für die Ausgabe des mit dem CUDA-SDK mitgelieferten Programms „`deviceQuery`“ (vgl. Progr. 4.2).
- Software
  - Ubuntu 10.4 LTS, Kernel 2.6.32-28
  - Nvidia Treiber 260.24
  - gcc-4.4.3
  - CUDA 3.2
  - LibTomCrypt in der Version 1.17

Die im Abschnitt „Funktionalität“ (4.1) vorgestellte Architektur eignet sich nur bedingt zur Evaluierung der Performance der `cuvpnapp`. Das hat vor allem folgende Gründe:

1. Die benötigte Hardware für Netzwerk-Verbindungen von mehreren Gbit/s kann nicht mehr als „Standard-Hardware“ bezeichnet werden. Es hätte die Möglichkeit zur Übertragung mit Infiniband gegeben, die Implementierung einer Applikation, die Infiniband als physisches Medium zur Übertragung verwendet, hätte intensive und zeitlich aufwendige Recherchen erfordert, die nicht Kern-Bestandteil dieser Arbeit sind.
2. Die verwendeten asynchronen Sockets erfordern für Geschwindigkeiten jenseits der Gbit/s-Marke eine spezielle Umsetzung. Die Implementierung von Netzwerkverbindungen über Sockets für Gbit-Verbindungen würde den Rahmen dieser Arbeit sprengen.

[4] http://www.intel.com/products/desktop/motherboards/dx58so/dx58so-overview.htm
[5] http://ark.intel.com/Product.aspx?id=37148
[6] Hersteller: Kingston, Art. Nr.: KVR1333D3N9K2
[7] http://www.nvidia.com/object/product_geforce_gtx_480_us.html

```
Device 0: "GeForce GTX 480"
  CUDA Driver Version:                           3.20
  CUDA Runtime Version:                          3.20
  CUDA Capability Major/Minor version number:    2.0
  Total amount of global memory:                 1609760768 bytes
  Multiprocessors x Cores/MP = Cores:            15 (MP) x 32 (Cores/MP)
     = 480 (Cores)
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 32768
  Warp size:                                     32
  Maximum number of threads per block:           1024
  Maximum sizes of each dimension of a block:    1024 x 1024 x 64
  Maximum sizes of each dimension of a grid:     65535 x 65535 x 1
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Clock rate:                                    1.40 GHz
  Concurrent copy and execution:                 Yes
  Run time limit on kernels:                     Yes
  Integrated:                                    No
  Support host page-locked memory mapping:       Yes
  Compute mode:                                  Default (multiple host
    threads can use this device simultaneously)
  Concurrent kernel execution:                   Yes
  Device has ECC support enabled:                No
  Device is using TCC driver mode:               No

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 3.20, CUDA
    Runtime Version = 3.20, NumDevs = 1, Device = GeForce GTX 480
```

Programm 4.2: Ausgabe von `deviceQuery` für die GTX480.

3. Die maximalen Geschwindigkeiten unter der Verwendung des TUN-Devices lagen bei den durchgeführten Tests weit unter der nativen Performance des physischen Mediums. Auch nach intensiver Recherche konnte nicht herausgefunden werden, bei welcher Grenze die maximale Performance eines TUN-Devices liegt.

Aufgrund der Nicht-Vorhersagbarkeit der Performance des TUN-Devices ist eine Test-Applikation entwickelt worden, die eine Gigabit-Netzwerkverbindung simuliert. Dazu werden fiktive Netzwerk-Pakete aus einem Puffer im Hauptspeicher gelesen, verschlüsselt und in einen zweiten Puffer kopiert. Die Geschwindigkeit, mit der die Pakete dabei ankommen, werden daher nicht durch ein Netzwerk-Device, sondern durch die Performance der `memcpy`-Operationen aus dem Hauptspeicher bestimmt. Im ersten Schritt werden daher die Performance-Daten der `memcpy`-Operationen ermittelt und analysiert. Wie in Abb. 4.3 ersichtlich, reicht die Performance weit über die gewünschten 10 Gbit/s hinaus. Die grünen Balken in der Abbildung zeigen

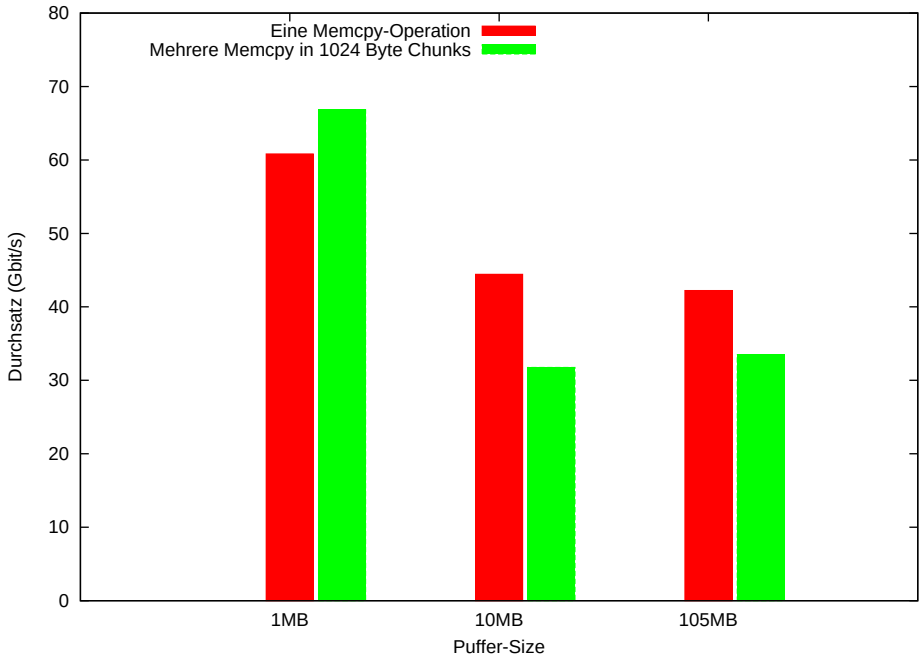

Abbildung 4.3: Performance-Unterschiede zwischen kopieren eines Puffers im RAM mit einem `memcpy` oder der Teilung in 1024 Byte lange Stücke („Chunks“), die sequentiell kopiert werden.

außerdem, wie sich die Aufteilung der Puffer-Größe in 1024-Byte-Chunks auf die Geschwindigkeit auswirkt. Bei einer Daten-Größe von 1 MB beeinflusst die Teilung in Chunks noch nicht die Performance. Bei 10 und 105 MB wird deutlich, dass sich die vielen `memcpy`-Operationen und der Overhead, der durch die zusätzliche `for`-Schleife entsteht, negativ auf das Ergebnis auswirken. Die Daten kommen aber in beiden Fällen mit ausreichender Geschwindigkeit für die Verarbeitung mit AES-GCM an.

Im nächsten Schritt wird die Verschlüsselung der fingierten Pakete vorgenommen. Um eine vergleichsweise realistische Simulation umzusetzen und die Testfälle rekonstruieren zu können, wird folgendes Schema durchgeführt:

1. Auf der Festplatte wird eine Datei der gewünschten Größe, mit Zufalls-Daten aus `/dev/urandom`, erzeugt.
2. Die erzeugte Datei wird in einen Puffer im RAM geladen.
3. Für die Verschlüsselung werden immer 1024-Byte-Chunks des Puffers kopiert und der `gcm_encrypt`-Funktion (vgl. S. 49) übergeben. Der Ciphertext und der Authentication Tag werden anschließend in einem Paket zusammengefügt und in einen zweiten Puffer kopiert.
4. Der verschlüsselte Puffer wird zusammen mit den Authentication Tags zur späteren Überprüfung in eine eigene Datei geschrieben.

```
1 gettimeofday(&start, 0);
2 for(i=0;i<packet_rounds;i++){
3
4   //copy from file buffer to encrypt it
5   memcpy(package,file+(i*PACKET_SIZE),PACKET_SIZE);
6
7   //copy start nonce after encrypted package
8   PUTU32(enc_package+PACKET_SIZE, (sa_context->sa->nonce[0]));
9   PUTU32((enc_package+PACKET_SIZE+4), (sa_context->sa->nonce[1]));
10  PUTU32((enc_package+PACKET_SIZE+8), (sa_context->sa->nonce[2]));
11  PUTU32((enc_package+PACKET_SIZE+12),(sa_context->sa->nonce[3]));
12
13  //encrypt packet
14  gcm_encrypt(sa_context,package,enc_package,
15    PACKET_SIZE,0,0,tag,TAG_LEN);
16
17  //copy tag after nonce
18  memcpy(enc_package+PACKET_SIZE+16,tag,TAG_LEN);
19
20  //copy encrypted pacekt to buffer
21  memcpy(file_out+(i*(PACKET_SIZE+16+TAG_LEN)),enc_package,
22    (PACKET_SIZE+16+TAG_LEN));
23 }
24 gettimeofday(&end, 0);
25 time=((end.tv_sec - start.tv_sec)*1000.0) + ((end.tv_usec - start.
       tv_usec)/1000.0);
```

Programm 4.3: Programmcode für die Evaluierung der Performance des LibTomCrypt-Patches.

Für die Zeitmessungen wird jeweils vor und nach der Schleife der Verschlüsselungs-Operation mit `gettimeofday` die aktuelle Systemzeit abgefragt und abschließend die Differenz der beiden Zeiten berechnet. Progr. 4.3 zeigt, wie die einzelnen Schritte in der Implementierung umgesetzt sind.

Für den Vergleich mit der ursprünglichen LibTomCrypt wird exakt derselbe Code-Abschnitt verwendet, nur dass der GPU-Patch der Bibliothek fehlt. Dadurch wird ein fairer Vergleich zwischen den beiden Versionen möglich, der die Geschwindigkeits-Unterschiede, zwischen der AES-Verschlüsselung eines Counters und dem Kopieren eines Counters aus dem Puffer, zeigt. Die Geschwindigkeiten des GPU-Moduls sind für diesen Test jedoch nur bedingt relevant. Da die `gmc_encrypt`-Funktion eine SA benötigt, deren Puffer bereits befüllt sind, beeinflusst die Performance des GPU-Moduls diesen Test nicht. Die Geschwindigkeit des GPU-Moduls rückt erst dann wieder in den Vordergrund, wenn Pakete schneller abgehandelt werden können, als Puffer befüllt werden können. Es müsste dann mit der Paket-Ver-/Entschlüsselung gewartet werden, wenn beide Puffer einer SA leer sind. Solange die GPU mit dem Befüllen schneller ist, als die CPU mit der Ver-/Entschlüsselung

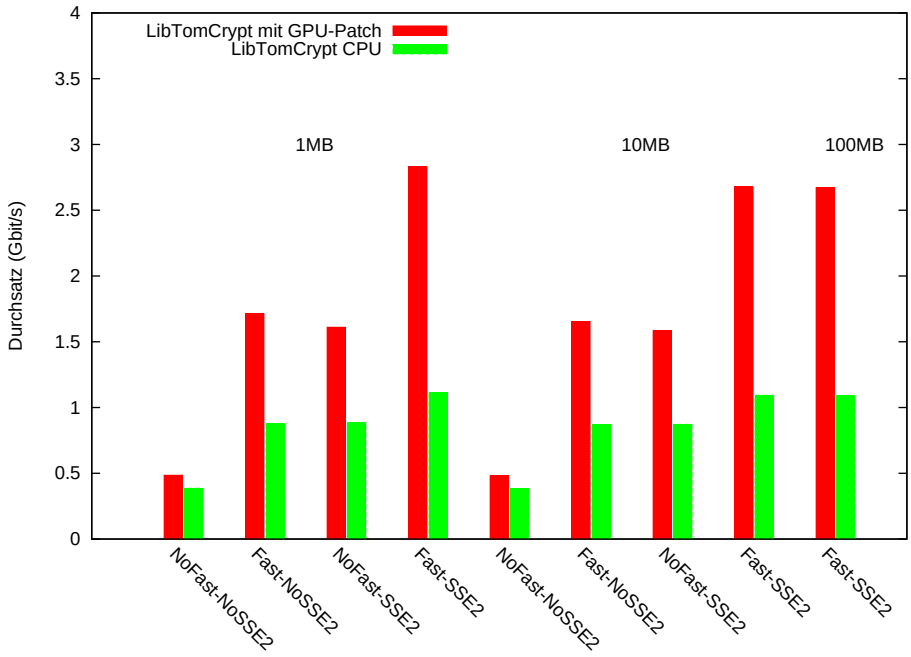

Abbildung 4.4: Performance-Vergleich von: (i) Der unveränderten LibTomCrypt und (ii) der LibTomCrypt mit GPU-Patch (jeweils mit und ohne LTC_FAST uns SSE2-Instruktionen kompiliert).

der Pakete, entstehen keine Wartezeiten. Für den Performance-Test wird daher eine SA verwendet, deren Puffer bereits gefüllt sind. Auf den ersten Blick mag die Tatsache, dass das Befüllen der Puffer nicht mit in die Zeitmessung einfließt, ungerecht erscheinen. Wird jedoch bedacht, dass im Regelfall das Befüllen parallel durchgeführt werden kann, so ist das Herausnehmen der anfänglichen Befüllung durchaus gerechtfertigt.

Die Abbildung 4.4 stellt die Ergebnisse der durchgeführten Tests für die Puffer-Größen 1, 10 und 100 MB grafisch dar. Für die Puffer-Größen 1 und 10 MB werden vier unterschiedlich kompilierte Versionen der Bibliothek verwendet und die Laufzeiten gemessen. Die Versionen `Fast` und `NoFast` stehen jeweils für die Kompilierung mit und ohne aktiviertem `LTC_FAST`-Flag. Dazu wird in der Datei „`tomcrypt_custom.h`“ (Zeile 123) das Define „`LTC_NO_FAST`“ entweder aus- oder kommentiert. Mit aktiviertem `LTC_FAST` werden XOR-Operationen nicht mehr auf Byte-Level (mit dem Datentyp `unsigned char`), sondern unter Verwendung von `unsigned long` durchgeführt [8, S. 307]. Wie dies zur Beschleunigung von GCM eingesetzt wird, kann detailliert in [8, S. 306ff] nachgelesen werden. Die zweite Optimierungs-Möglichkeit betrifft den Einsatz der „Intel Streaming SIMD Extensions 2“

(SSE2)[8]. Diese können über `LTC_GCM_TABLES_SSE2` – wiederum in der Datei `tomcrypt_custom.h` (Zeile 242) – de- oder aktiviert werden und wirken sich z. B. positiv auf die Laufzeit der Multiplikation im Galois-Feld aus [8, S. 325].

Dass die De- und Aktivierung sowie die Kombination von `LTC_FAST` und `SSE2` sich tatsächlich positiv auf den Durchsatz von AES-GCM auswirken, lässt sich deutlich in Abb. 4.4 erkennen. Der Durchsatz steigt von ca. $0,5$ Gbit/s ohne Optimierung bis auf ca. 3 Gbit/s mit beiden Optionen aktiviert. Bei voller Optimierung erreicht der Patch fast das Dreifache an Durchsatz gegenüber der ursprünglichen LibTomCrypt-Version. Interessanterweise profitiert der Patch mehr von `LTC_FAST` und `SSE2`, da sich hier die Performance auf ca. das Sechsfache erhöht. Ohne Patch wächst der Durchsatz nur auf ca. das Dreifache an. Dass die Durchsatz-Raten dann auch konstant bleiben, belegen die evaluierten Zahlen für die Verschlüsselung von 100 MB. Hierbei wurde der Test nur mehr mit aktiviertem `LTC_FAST` und `SSE2` durchgeführt.

Die ermittelten Daten ergeben sich aus dem Mittelwert der 1000-fachen Verschlüsselung von 1, 10 und 100 MB in 1024-Byte-Paketen. Auf die Darstellung der Streuung der Laufzeit der 1000 Runden wird aufgrund zu geringer Abweichungen (unter 1 Millisekunde) verzichtet.

Im nächsten Schritt stellt sich die Frage, wo bei der Verschlüsselung die meiste Zeit benötigt wird. Die Haupt-Schleife zur Evaluierung der Performance (Code 4.3) besteht aus dem Kopieren des Puffers mit den Zufallsdaten, dem Aufruf von `gcm_encrypt` (vgl. S. 49) für die Verschlüsselung und dem Zusammenbau des verschlüsselten Pakets (Nonce und Authentication Tag mit dem Ciphertext in ein Paket packen). Abbildung 4.5 stellt die Anteile an der Laufzeit der Haupt-Schleife dar. Ungefähr 5,5% werden für den Zusammenbau eines Pakets aufgewendet und mehr als 90% der Zeit wird an der Verschlüsselung gearbeitet. Die übrigen Prozentanteile (ca. 4%) werden für das Kopieren aus dem Zufalls- und in den Ergebnis-Puffer, Funktionsaufrufen und Parameter-Übergabe sowie dem Prüfen der Schleifen-Bedingung benötigt. Da in der Funktion `gcm_encrypt` am längsten verweilt wird, ist diese Funktion für eine weitere Analyse interessant. Die Funktion `gcm_encrypt` besteht aus mehreren Teilen (S. 49), von denen annähernd die gesamte Laufzeit `gcm_process` (s. auch S. 47) zuzuordnen ist. In Bezug auf die Gesamtlaufzeit beträgt der prozentuale Anteil von `gcm_process` ca. 90%, was die Rechen-Intensität dieser Funktion verdeutlicht. Zur Erinnerung kurz die Aufgaben von `gcm_process`:

- Kopieren der verschlüsselten Counter aus dem Double-Buffer.
- XOR der Counter mit dem Plain- bzw. Ciphertext.
- Berechnung des Authentication Tags.

In Abbildung 4.6 wird ersichtlich, dass mehr als die Hälfte der Zeit in `gcm_process` (ca. 60%) für die Multiplikation mit dem Hash-Subkey

[8] S.a. http://www.intel.com/support/processors/sb/CS-030123.htm?wapkw=(SSE).

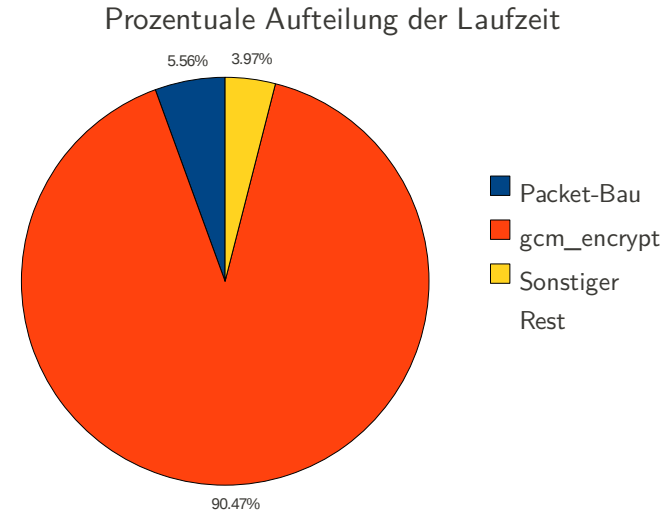

Abbildung 4.5: Aufschlüsselung der einzelnen Anteile an der Schleife zur Verschlüsselung der Daten (vgl. Progr. 4.3).

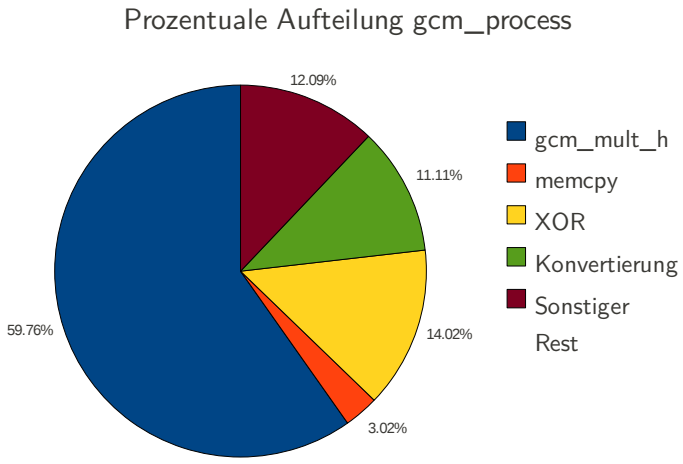

Abbildung 4.6: Aufschlüsselung der einzelnen Anteile an `gcm_process`.

(`gcm_mult_h`) benötigt wird. Umso wichtiger ist es daher, dass diese Multiplikation effizient durchgeführt wird. Die nächst größeren Teile sind die Durchführung des XORs (die Counter mit dem Plaintext, das Ergebnis von `gcm_mult_h` mit dem Ciphertext) mit ca. 14% und die Konvertierung der Counter aus dem Double-Buffer in das Format der LibTomCrypt (ca. 11%). Bezüglich der Performance der `memcpy`-Operationen bestätigen sich die Daten aus Abb. 4.3 auf S. 63, da für das Kopieren aus den Double-Buffern lediglich ca. 3% aufgewendet werden. Ungefähr 12% verbleiben unter anderem bei den Tätigkeiten für Funktionsaufrufe, Schleifen-Umsetzung oder

auch Variablen-Initialisierung.

4.2.1 Unterschiede zwischen GPU- und CUDA-Versionen

Im Zuge dieser Arbeit wurde mit verschiedenen GPU-Baureihen und CUDA-Versionen gearbeitet. Begonnen wurde die Entwicklung der Applikationen auf einer GTX285 (vgl. Progr. 4.4), auf der auch die ursprünglichen Tests zu den Streams und Texturen durchgeführt wurden. Auf der GTX285 zeigte die Verwendung von Streams und Texturen einen deutlichen Performance-Schub gegenüber einer unoptimierten Variante. Auch der Einsatz von 4 anstatt 2 Streams brachte eine deutliche Verkürzung der Laufzeit hervor. Bei den späteren Tests mit der GTX480 zeigten sich die Vorteile der Streams nicht noch ein weiteres Mal. Im Gegenteil, hier hatte die Variante ohne Streams sogar leichte Vorteile.

Im Übrigen ist auch die Verwendung des Texturen-Speichers seit der Fermi-Architektur nicht immer vorteilhaft. Im Tuning Guide zu Fermi findet sich folgende Aussage wieder [28, S. 3]:

> ... since global memory loads are cached in L1 and the L1 cache has higher bandwidth than the texture cache.

Das kann für Texture Fetches bedeuten, dass sie unter Umständen langsamer sind als Uncoalesced Loads vom gecachten Global Memory. Das Vorhandensein des L1- und L2-Caches mit der Einführung der Compute Capability 2.x beeinflusst daher die Geschwindigkeit einer CUDA-Applikation. Ob die Performance auf Baureihen mit Compute Capability 1.x oder 2.x besser ist, muss für jede Applikation eigens evaluiert werden. Dazu kann zum Zeitpunkt der Entwicklung der Caching-Mechanismus des Global Memory deaktiviert werden. Die Größe des L1-Cache des Local Memory lässt sich ebenfalls kontrollieren. Mit diesen beiden Kontroll-Einstellungen können die neuen Caches getestet werden.

Für das GPU-Modul bedeuten diese beiden Neuerungen, dass unter Umständen eine Variante ohne Streams und ohne Texturen-Speicher auf einem Device mit Compute Capability 2.x schneller wäre. Es war im Zuge dieser Arbeit nicht mehr genug Zeit, das GPU-Modul für die GTX480 zu optimieren. Es zeigte sich aber deutlich, dass die Verwendung einer neuen GPU-Baureihe nicht mit einer höheren Performance verbunden sein muss.

```
1 Device 0: "GeForce GTX 285"
2   CUDA Driver Version / Runtime Version 4.0 / 4.0
3   CUDA Capability Major/Minor version number: 1.3
4   Total amount of global memory: 1024 MBytes (1073545216 bytes)
5   (30) Multiprocessors x ( 8) CUDA Cores/MP: 240 CUDA Cores
6   GPU Clock Speed: 1.48 GHz
7   Memory Clock rate: 1242.00 Mhz
8   Memory Bus Width: 512-bit
9   Max Texture Dimension Size (x,y,z) 1D=(8192), 2D=(65536,32768), 3D
      =(2048,2048,2048)
10  Max Layered Texture Size (dim) x layers 1D=(8192) x 512, 2D
      =(8192,8192) x 512
11  Total amount of constant memory: 65536 bytes
12  Total amount of shared memory per block: 16384 bytes
13  Total number of registers available per block: 16384
14  Warp size: 32
15  Maximum number of threads per block: 512
16  Maximum sizes of each dimension of a block: 512 x 512 x 64
17  Maximum sizes of each dimension of a grid: 65535 x 65535 x 1
18  Maximum memory pitch: 2147483647 bytes
19  Texture alignment: 256 bytes
20  Concurrent copy and execution: Yes with 1 copy engine(s)
21  Run time limit on kernels: No
22  Integrated GPU sharing Host Memory: No
23  Support host page-locked memory mapping: Yes
24  Concurrent kernel execution: No
25  Alignment requirement for Surfaces: Yes
26  Device has ECC support enabled: No
27  Device is using TCC driver mode: No
28  Device supports Unified Addressing (UVA): No
29  Device PCI Bus ID / PCI location ID: 8 / 0
30
```

Programm 4.4: Ausgabe von `deviceQuery` für die GTX285.

Kapitel 5

Resümee

In den letzten Jahren entwickelte sich ein starkes Interesse an der Umsetzung von Algorithmen für GPUs. Dies wurde vor allem durch die rasche Weiterentwicklung und Verbesserung der CUDA-Architektur durch Nvidia vorangetrieben[1]. Diese Arbeit zeigt, wie kryptografische Algorithmen auf GPUs umgesetzt werden und im Netzwerk-Bereich zum Einsatz kommen können. Als Algorithmus wurde der standardisierte „AES im Galois/Counter Mode“ ausgewählt, da er als Combined Cipher Vertraulichkeit, Integrität und Authentizität bietet. Darüber hinaus eignet sich AES-GCM für eine kombinierte Implementierung auf CPU und GPU, da die AES-Verschlüsselung unabhängig vom Authentifizierungs-Teil durchgeführt werden kann. Diese Unabhängigkeit wird in dieser Arbeit durch das Zusammenspiel von GPU- und CPU-Modul demonstriert. Das GPU-Modul dient zum Vorausberechnen von AES verschlüsselten Countern, das CPU-Modul wird für die Berechnung der Authentication Tags und die Verwaltung der vorausberechneten Counter verwendet.

Diese Verwaltung wurde hierbei in Form von Double-Buffer realisiert, die abwechselnd für die Ver- und Entschlüsselung eingesetzt werden. Im Falle eines leer gewordenen Puffers wird die GPU dazu verwendet, um den Puffer neu zu befüllen. Dabei ist vor allem wichtig, dass das GPU-Modul parallel zum CPU-Modul arbeitet und dieses nicht blockiert bzw. die CPU benachrichtigt wird, wenn der Puffer wieder befüllt wurde. Dadurch ist die maximale Geschwindigkeit einer Ver- oder Entschlüsselung nicht in erster Linie von der AES-Verschlüsselung abhängig. Die Double-Buffer zeigen daher eine elegante Möglichkeit, wie vorausberechnete Ergebnisse – ob von der GPU oder anderen Coprozessoren – verwaltet und eingesetzt werden können.

Während der gesamten Implementierungszeit wurde auf eine Verwendung der einzelnen Teile für IPsec geachtet. Dies ist in erster Linie von Be-

[1] Zum Zeitpunkt dieser Arbeit wurde die Version 4.0 des CUDA-Toolkits veröffentlicht (http://bit.ly/cuda4features).

deutung, da die Parameter für AES-GCM in IPsec korrekt initialisiert und verwendet werden müssen. In dieser Arbeit wurden hierbei auch Änderungen an den Standards vorgestellt, die nötig sind, um Counter vorausberechnen zu können. Diese Änderungen betreffen die Zusammenstellung der Nonce aus ESP-IV und IKE-Salt und den Einsatz von GHASH zu Beginn von AES-GCM.

In weiterer Hinsicht bedeutet der Einsatz der einzelnen Teile für IPsec, dass viele kleine Datenpakete, aber nie sehr große Datenmengen verschlüsselt werden. Diese Tatsache kam vor allem bei der Durchführung der Performance-Tests zum Tragen, da auf eine realistische Simulation einer IPsec-Verbindung geachtet werden musste. Aufgrund fehlender Netzwerk-Hardware für mehrere Gbit/s wurde die maximal zu erreichende Geschwindigkeit bei der Verschlüsselung von Datenpaketen aus dem Arbeitsspeicher eruiert. Dabei wurde eine AES-GCM-Implementierung der LibTomCrypt gegen eine gepachte Version (Integration des GPU-Moduls) getestet und die Geschwindigkeiten verglichen. Im nächsten Schritt wurde analysiert und gezeigt, in welchen Teilen der Applikation die meisten Ressourcen verbraucht wurden. Hier zeigte sich deutlich der Vorteil des GPU-Patches und der Aufwand der Operation `gcm_mult_h` bei der Berechnung des Authentication Tags.

Des Weiteren wurde auf die möglichen Fehlerszenarien bei der Übertragung von Netzwerkpaketen eingegangen und erläutert, wie diese mit Hilfe der Double-Buffer behandelt werden können. In der Praxis wurden anschließend die ausgearbeiteten Ideen unter Verwendung von virtuellen TUN-Devices getestet und gezeigt, dass die in der Theorie entwickelten Techniken funktionieren.

5.1 Weiterführende Arbeiten

Im Zuge dieser Arbeit traten einige Fragen auf, die nicht oder nur mehr zum Teil beantwortet werden konnten. Vor allem für die zeitaufwendige, aber zentrale Operation `gcm_mult_h`, gäbe es neue Möglichkeiten für eine effiziente Implementierung. Intel veröffentlichte mit der neuen Westmere-Architektur auch eine Instruktion namens „`PCLMULQDQ`“, die für die Multiplikation im Galois-Feld (verwendet in GHASH) eingesetzt werden kann [13]. Eine komplett neu geschriebene Implementierung ohne LibTomCrypt unter Verwendung dieser Operationen wäre interessant für eine weitergehende Analyse.

Der wohl aufwändigste, aber sicher auch interessanteste Punkt, wäre die Integration des GPU-Moduls in eine bestehende IPsec-Implementierung im Linux-Kernel. Intel hat dazu in der Arbeit „*Using Intel AES New Instructions and PCLMULQDQ to Significantly Improve IPsec Performance on Linux*“ [14] erste Ergebnisse vorgestellt und gezeigt, dass sich AES-GCM gut für Gbit-VPN-Verbindungen eignet und sich mit den neuen Intel-

Operationen beschleunigen lässt.

Generell muss die Thematik der Verwendung von GPUs zur Beschleunigung von Netzwerkapplikationen noch tiefer erforscht und vor allem ausführlich getestet werden. Da z. B. bei einem VPN-Gateway viele parallele Verbindungen existieren können, sind eine oder mehrere GPUs für die Ver- und Entschlüsselung aller Verbindungen verantwortlich. Es wird daher nötig sein eine geeignete Architektur zu entwickeln, die es den Verbindungen erlaubt, die GPUs gemeinsam fair zu benutzen. Interessante Ergebnisse zu diesem Problem können z. B. in [3] gefunden werden.

Abschließend kann zur Programmierung mit CUDA gesagt werden, dass sich das Framework schnell weiterentwickelt und immer wieder neue Funktionen hinzukommen. Es wird für High-Performance-Anwendungen vor allem wichtig sein, einen Weg zu finden, um die CUDA-Applikationen analysieren zu können. Mit dem „Visual Profiler“[2] und dem Visual Studio Plugin „Parallel Nsight“ sind passende Tools bereits vorhanden. Es wird sich zeigen, wie gut und in welchen Fällen sich diese Werkzeuge für eine Analyse eignen, um Flaschenhälse von Applikationen aufzudecken und die Unterschiede zwischen einzelnen GPUs zu vergleichen.

[2] http://developer.download.nvidia.com/compute/cuda/4_0_rc2/toolkit/docs/VisualProfiler/Compute_Visual_Profiler_User_Guide.pdf.

Anhang A

Programmbeispiele

A.1 SA-Konfiguration

```
testapp =
{
  SAs = (
    #SA 0
    { keybits = 128;
    key = [0xfe,0xff,0xe9,0x92,0x86,0x65,0x73,0x1c,0x6d,0x6a,0x8f,0x94,0
     x67,0x30,0x83,0x08];
    nonce = [0xcafebabe, 0xfacedbad, 0xdecaf888, 0x00000001];
    segments = 4; },
    #SA 1
    { keybits = 128;
    key = [0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08,0x09,0x0a,0x0b,0
     x0c,0x0d,0x0e,0x0f];
    nonce = [0x00112233,0x44556677,0x8899aabb,0xccddeeff];
    segments = 4; },
    #SA 2
    { keybits = 128;
    key = [0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08,0x09,0x0a,0x0b,0
     x0c,0x0d,0x0e,0x0f];
    nonce = [0x0,0x0,0x0,0x00000002];
    segments = 0; });
};
```

A.2 Zusammenspiel der GPU-Modul-Funktionen

```
char config[]="./SA.cfg"; //Path to config file
sa_t *sa_list = NULL;  //The head of the SA linked list.
int sa_list_items = 0; //The number of SAs in the config file
int sa_conf_count = 0; //The number of SAs for which the segments are not 0
uint nrounds;          //Variable for the AES-rounds
size_t out_size;       //Size of the resulting enc. counters in bytes
uint32_t *out;         //Pointer to the resulting enc. counters
uint32_t *nonces;       //Array for all nonces of the SAs
unsigned int *segments; //Array for all segments of the SAs
```

```
10 unsigned char *keys;     //Array for all keys of the SAs
11 size_t rks_size;         //Size of a round key
12 uint32_t *rks;           //Array for the round keys of all SAs
13
14 /*Read out the config file to get our parameters and create a linked list of all
       SA items*/
15 read_config(config,&sa_list,&sa_list_items,&sa_count);
16 if((*sa_list_item) == NULL){
17   fprintf(stderr, "Could not init and config SA successfully!\n");
18   exit(-1);
19 }
20
21 /*Allocate memory for all keys, nonces and segments*/
22 if( (keys = malloc(sa_conf_count * sizeof(unsigned char) * (KEYBITS/8)))
       == NULL){
23   fprintf(stderr, "Could not allocate memory for keys\n");
24   return EXIT_FAILURE;
25 }
26 if( (nonces = malloc(sa_conf_count * sizeof(uint32_t) * 4)) == NULL){
27   fprintf(stderr, "Could not allocate memory for nonces\n");
28   free(keys);
29   return EXIT_FAILURE;
30 }
31 if( (segments = malloc(sa_conf_count * sizeof(uint))) == NULL){
32   fprintf(stderr, "Could not allocate memory for segments\n");
33   free(keys);
34   free(nonces);
35   return EXIT_FAILURE;
36 }
37
38 /*Merge the items for that we want to calculate segments as all SAs were read
       from the config file. Items with 0 segments are skipped. All keys, nonces
       and segments are collected in arrays*/
39 merge_sa_data(sa_list,keys,nonces,segments,&sa_count);
40
41 /*Allocate memory for all round keys*/
42 rks_size = sa_count * RKLENGTH(KEYBITS) * sizeof(uint32_t);
43 if( (rks = malloc(rks_size)) == NULL){
44   fprintf(stderr, "Could not allocate memory for round keys\n");
45   free(keys);
46   free(nonces);
47   free(segments);
48   return EXIT_FAILURE;
49 }
50
51 /*Expand the keys read from the config file */
52 nrounds = expand_keys(keys,sa_count,rks);
53
54 /*Request the segments from the GPU for all SAs.
55  *Note that the counters are mixed up due to the use of streams*/
56 gputime = d_encrypt_nonces(rks, sa_count, nonces, sa_count, nrounds,
       segments, &out, &out_size);
```

A.3 Initialisieren der Double-Buffer

```
1  int sas_init(char *file, sa_t **sa_list){
2
3    int i;
4    float gputime;
5    int sa_list_items = 0;//number of SAs configured in the config file
6    int sa_conf_count = 0;//number of SAs for which segments are != 0
7    int sa_count = 0;
8    size_t out_size;
9    uint32_t *out;//counters generated by the gpu
10   unsigned int *segments;//merged segments of all SAs
11   unsigned char *keys;//merged keys of all SAs
12   uint32_t *nonces;//merged nonces of all SAs
13   size_t rks_size;//merged round keys of all SAs
14   uint32_t *rks;//the round keys
15
16   uint nrounds;
17   sa_t *helper = NULL;
18
19   read_config(file, sa_list,&sa_list_items,&sa_conf_count);
20   if(sa_conf_count != 2){
21       fprintf(stderr, "\nMore than two SAs configured, just one in and
       one out must be configured!\n");
22       return EXIT_FAILURE;
23   }
24
25   if((*sa_list) == NULL){
26     fprintf(stderr, "\nCould not init and config SAs successfully!\n");
27     return EXIT_FAILURE;
28   }
29   helper = (*sa_list);//helper points to the start of our list
30   while(helper != NULL){
31
32     if((helper->segments) == 0){
33       helper = helper->next;
34       continue;
35     }
36     //we set all the buffer attributes to 0 and NULL
37
38     init_buffer(helper);
39     helper = helper->next;
40   }
41   //set helper back to the start of the list!
42   helper = (*sa_list);
43
44   if( (keys = malloc(sa_conf_count * sizeof(unsigned char) * (KEYBITS/8)
       )) == NULL){
45       fprintf(stderr, "\nCould not allocate memory for keys\n");
46       return EXIT_FAILURE;
47   }
48   if( (nonces = malloc(sa_conf_count * sizeof(uint32_t) * 4)) == NULL){
49     fprintf(stderr, "\nCould not allocate memory for nonces\n");
```

```
50      free(keys);
51      return EXIT_FAILURE;
52    }
53    if( (segments = malloc(sa_conf_count * sizeof(uint))) == NULL){
54      fprintf(stderr, "\nCould not allocate memory for segments\n");
55      free(keys);
56      free(nonces);
57      return EXIT_FAILURE;
58    }
59    //merge that items for that we want to calculate segments
60    //items with 0 segments are skipped
61    merge_sa_data((*sa_list),keys,nonces,segments,&sa_count);
62    if(sa_conf_count != sa_count){
63      fprintf(stderr, "\nConfigured and merged SAs don't match!!!\n");
64      return EXIT_FAILURE;
65    }
66    //calculate round keys for all SAs
67    rks_size = sa_count * RKLENGTH(KEYBITS) * sizeof(uint32_t);
68    if( (rks = malloc(rks_size)) == NULL){
69      fprintf(stderr, "Could not allocate memory for round keys\n");
70      free(keys);
71      free(nonces);
72      free(segments);
73      return EXIT_FAILURE;
74    }
75    nrounds = expand_keys(keys,sa_count,rks);
76    if(nrounds == 0){
77      fprintf(stderr, "\nRound Key function returned 0 for nrounds!!!\n");
78      free(rks);
79      free(keys);
80      free(nonces);
81      free(segments);
82      return EXIT_FAILURE;
83    }
84
85    //as we want to fill both buffers we must double the generated segments
86    for(i=0;i<sa_count;i++)
87      segments[i] = segments[i] * 2;
88
89    //call GPU to produce counters for all SAs!
90    gputime = d_encrypt_nonces(rks, sa_count, nonces, sa_count, nrounds,
        segments, &out, &out_size);
91
92    //we copy the counters from one array to the two buffers of each SA
93    if((fill_buffers((*sa_list),out,out_size)) != EXIT_SUCCESS){
94      fprintf(stderr, "\nCould not fill both buffers of all SAs!!!\n");
95      return EXIT_FAILURE;
96    }
97
98    return EXIT_SUCCESS;
99  }
```

Literaturverzeichnis

[1] *ISO/IEC 9899:1999 - Programming languages – C.*

[2] *Cuda compatible GPU as an efficient hardware accelerator for AES cryptography*, IEEE International Conference on Signal Processing and Communication, 2007. http://www.manavski.com/downloads/PID505889.pdf, abgerufen am 24.05.2011.

[3] *SSLShader: Cheap SSL Acceleration with Commodity Processors*, Proceedings of the USENIX Symposium on Networked Systems Design and Implementation, March 2011. www.ndsl.kaist.edu/papers/sslshader.pdf, abgerufen am 24.05.2011.

[4] Daemen, J. und V. Rijmen: *The Design of Rijndael: AES. The Advanced Encryption Standard*. Springer, 2002.

[5] Daemen, J. und V. Rijmen: *AES Proposal: Rijndael*, Sep. 2009. http://csrc.nist.gov/archive/aes/rijndael/Rijndael-ammended.pdf, abgerufen am 24.05.2011.

[6] Davis, K., J. W. Turner und N. Yocom: *The Definitive Guide to Linux Network Programming*. Apress, 2004.

[7] Denis, T. S.: *LibTomCrypt 1.17 Developer Manual*. Ontario Canada. http://libtom.org/?page=features&whatfile=crypt, abgerufen am 24.05.2011.

[8] Denis, T. S. und S. Johnson: *Cryptography for Developers*. Syngress Publishing, Inc., 2007.

[9] Dworkin, M.: *Recommendation for Block Cipher Modes of Operation: Methods and Techniques*. NIST Special Publication 800-38A, 2001. http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf, abgerufen am 24.05.2011.

[10] Dworkin, M.: *Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC*. NIST Special Publication 800-38D, 2007. http://csrc.nist.gov/publications/nistpubs/800-38D/SP-800-38D.pdf, abgerufen am 24.05.2011.

[11] Frankel, S., K. Kent, R. Lewkowski, A. D. Orebaugh und R. W. R. S. R. Sharma: *Guide to IPsec VPNs*. NIST Special Publication 800-77, 2005. http://csrc.nist.gov/publications/nistpubs/800-77/sp800-77.pdf, abgerufen am 24.05.2011.

[12] Gay, W. W.: *Linux Socket Programming by Example*. QUE, 2000.

[13] Gueron, S. und M. E. Kounavis: *Intel Carry-Less Multiplication Instruction and its Usage for Computing the GCM Mode*. Techn. Ber., Intel Corporation, 2010. http://software.intel.com/en-us/articles/intel-carry-less-multiplication-instruction-and-its-usage-for-computing-the-gcm-mode/, abgerufen am 24.05.2011.

[14] Hoban, A.: *Using Intel AES New Instructions and PCLMULQDQ to Significantly Improve IPSec Performance on Linux*. Techn. Ber., Intel Corporation, 2010. http://download.intel.com/design/intarch/papers/324238.pdf, abgerufen am 24.05.2011.

[15] Kent, S.: *IP Encapsulating Security Payload (ESP)*, Dez. 2005. http://tools.ietf.org/html/rfc4303, abgerufen am 24.05.2011.

[16] Kent, S. und K. Seo: *Security Architecture for the Internet Protocol*, Dez. 2005. http://tools.ietf.org/html/rfc4301#ref-Eas05, abgerufen am 24.05.2011.

[17] Kipper, M., J. Slavkin und D. Denisenko: *Implementing AES on GPU*. Techn. Ber., University of Toronto, 2009. http://www.eecg.toronto.edu/~moshovos/CUDA08/arx/AES_ON_GPU_report.pdf, abgerufen am 24.05.2011.

[18] Kirk, D. und W. M. W. Hwu: *Programming Massively Parallel Processors: A Hands-On Approach*. Morgan Kaufman Publ. Inc., 2010.

[19] Krasnyansky, M.: *Universal TUN/TAP device driver - Linux Kernel Documentation*. http://www.mjmwired.net/kernel/Documentation/networking/tuntap.txt, abgerufen am 24.05.2011.

[20] Lewis, B. und D. J. Berg: *PThreads Primer – A Guide to Multithreaded Programming*. SunSoft Press, Prentice Hall, 1996.

[21] Lipp, M.: *VPN - Virtuelle Private Netzwerke: Aufbau und Sicherheit*. Addison-Wesley, 2001.

[22] McGrew, D. A. und J. Viega: *Flexible and Efficient Message Authentication in Hardware and Software*, 2003. http://www.cryptobarn.com/gcm/gcm-paper.pdf, abgerufen am 24.05.2011.

[23] McGrew, D. A. und J. Viega: *The Galois/Counter Mode of Operation (GCM) - revised*. Techn. Ber., 2005. http://www.csrc.nist.gov/groups/ST/toolkit/BCM/documents/proposedmodes/gcm/gcm-revised-spec.pdf, abgerufen am 24.05.2011.

[24] Menezes, A. J., P. C. van Oorschot und S. A. Vanstone: *Handbook of Applied Cryptography*. CRC Press, 2001. http://www.cacr.math.uwaterloo.ca/hac/, abgerufen am 24.05.2011.

[25] NIST: *Advanced Encryption Standard (AES) (FIPS PUB 197)*, 2001. http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf, abgerufen am 24.05.2011.

[26] NVIDIA: *CUDA C Best Practices Guide*. Online, Aug. 2010. http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_C_Best_Practices_Guide.pdf, abgerufen am 24.05.2011.

[27] NVIDIA: *NVIDIA CUDA C Programming Guide Version 3.2*. Online, Sep. 2010. http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_C_Programming_Guide.pdf, abgerufen am 24.05.2011.

[28] NVIDIA: *Tuning CUDA Applications for Fermi*. Techn. Ber., 2010. http://developer.download.nvidia.com/compute/cuda/4_0_rc2/toolkit/docs/Fermi_Tuning_Guide.pdf, abgerufen am 24.05.2011.

[29] Ottesen, A.: *Efficient parallelisation techniques for applications running on GPUs using the CUDA framework*. Diplomarbeit, Universität Oslo, 2009. http://www.duo.uio.no/sok/work.html?WORKID=91432, abgerufen am 24.05.2011.

[30] Paterson, K. G.: *A Cryptographic Tour of the IPsec Standards*. Inf. Secur. Tech. Rep., 11:72–81, 2006. http://eprint.iacr.org/2006/097.pdf, abgerufen am 24.05.2011.

[31] Sanders, J. und E. Kandrot: *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 2010.

[32] Viega, J. und D. McGrew: *The Use of Galois/Counter Mode (GCM) in IPsec Encapsulating Security Payload (ESP)*, Juni 2005. http://www.rfc-editor.org/rfc/rfc4106.txt, abgerufen am 24.05.2011.