Parallelisierungsmechanismen für eine kryptographische Sicherungsschicht

Bachelorarbeit

von

Moritz Gericke

aus

Celle

vorgelegt am

Lehrstuhl für Rechnernetze und Kommunikationssysteme
Prof. Dr. Martin Mauve
Heinrich-Heine-Universität Düsseldorf

Januar 2012

Betreuer:
Yves Igor Jerschow

Danksagung

An erster Stelle möchte ich mich bei meinem Betreuer Yves Igor Jerschow für die zuverlässige und freundliche Unterstützung über die drei vergangenen Monate bedanken. In zahlreichen Gesprächen wurde mir die Möglichkeit geboten, unterschiedliche Ansätze zu diskutieren ohne den Blick für das Wesentliche zu verlieren. Ich habe mich durchweg in meinen Ideen bestärkt gefunden, was mir eine große Motivation über die Dauer der Ausarbeitung war. Meinen großen Respekt möchte ich ihm als Autor der dieser Arbeit zugrunde liegenden Software mitteilen.

Einen besonderen Dank möchte ich meinen Mitbewohnerinnen und Mitbewohnern aussprechen, die mich verständnisvoll unterstützt und entlastet haben. Insbesondere Ruth Neeßen und Jahn Bertsch hatten darüber hinaus stets ein offenes Ohr für technische Probleme und halfen in hohem Maße bei der Korrektur des vorliegenden Textes.

Meinen Eltern danke ich herzlichst für ihre Bemühungen, die fortwährende Unterstützung sowie das Korrekturlesen der schriftlichen Ausarbeitung.

Einen abschließenden Dank möchte ich an Prof. Dr. Martin Mauve richten, der mir das Verfassen dieser Arbeit ermöglichte.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1.

Einleitung

1.1. Motivation

Die dieser Arbeit zugrunde liegende kryptographische Sicherungsschicht *(Cryptographic Link Layer, kurz CLL)* stellt eine Erweiterung des Protokollstapels der Internetprotokollfamilie dar [8]. Konzeptionell angesiedelt zwischen Sicherungsschicht und Netzwerkschicht bietet CLL transparent für existierende Protokolle Authentifizierungs- und Vertraulichkeitsmechanismen im lokalen Netzwerk. Die zu diesem Zweck angewandten kryptographischen Operationen sowie optionale Komprimierungsverfahren sind im Vergleich zum Normalbetrieb rechenintensiv und können je nach Ausstattung der kommunizierenden Hosts bzw. der Netzwerkinfrastruktur einen Flaschenhals bezüglich des Datendurchsatzes darstellen. Zwar lassen sich im *Fast-Ethernet* zwischen Rechnern mit Einkernprozessor ab 2 GHz Resultate nahe der theoretisch erreichbaren 100 Mbit/s messen, im *Gigabit-Ethernet* hingegen steht aufgrund hoher CPU-Auslastung nur ein Bruchteil der erreichbaren Bandbreite zur Verfügung. Mit Blick auf die steigenden Verkaufszahlen von gigabitfähiger Netzwerkhardware [3] wird eine adäquate Performance in diesem Bereich vom erwünschten Feature zur Anforderung. Auch da seit 2003 der vormalig exponentielle Anstieg der Taktfrequenz neu entwickelter Einkernprozessoren stagniert und statt dessen eine zunehmende Tendenz zur Produktion von Mehrkernprozessoren wahrnehmbar ist [22], macht es Sinn, sich verstärkt mit nebenläufiger Berechnung auseinanderzusetzen um rechenaufwändige Probleme effizienter bearbeiten zu können.

1.2. Problemstellung und Beitrag

Im Rahmen dieser Bachelorarbeit soll die Möglichkeit der Durchsatzsteigerung von CLL durch Anwendung von Parallelisierungsmechanismen untersucht sowie außerdem eine Grundlage für den Vergleich verschiedener angewandter Techniken geschaffen werden. Es wird dazu die bereits vorhandene CLL-Software auf mögliche Datenparallelitäten hin überprüft und abwärtskompatibel um Nebenläufigkeitsfunktionalität erweitert. Weiterhin soll die Effizienz der implementierten Lösung im Verhältnis zu alternativen Verfahren abgeschätzt werden. Zusätzlich soll CLL unter Windows die Möglichkeit erhalten, als Systemdienst gestartet zu werden, wie es bereits unter Linux in Form eines Daemon implementiert ist.

1.3. Struktur der Arbeit

In Kapitel 2 wird zuerst allgemein die existierende CLL-Software vorgestellt, um dann im Speziellen auf Konzepte und Komponenten einzugehen, welche für das Verständnis von besonderer Bedeutung sind oder eine Parallelisierung entweder begünstigen oder erschweren können. Kapitel 3 gibt einen Überblick über verschiedene Modelle bzw. Mechanismen der Parallelisierung und deren Anwendbarkeit auf die konkrete Problemstellung. Die Entscheidung zu einer Umsetzung des Threadpool-Musters wird in Kapitel 4 begründet sowie dessen Implementierung an ausgewählten Beispielen erläutert. Sowohl Testszenarien als auch Messergebnisse werden in Kapitel 5 präsentiert. In den Messungen wird die modifizierte Software mit der ursprünglichen verglichen. Des weiteren wird die manuelle Threadpool-Implementierung in Relation zu einem inkrementellen Ansatz unter Verwendung von *OpenMP* gesetzt. Dieses Kapitel schließt mit einer Reflexion der Ergebnisse und einem Ausblick auf mögliche Ziele einer weitergehenden Projektarbeit. Kapitel 6 befasst sich mit der neu implementierten Funktionalität, CLL als Windows-Systemdienst ausführen zu können. Mit einem Resümee über das Erreichte sowie einem kurzen Ausblick schließt Kapitel 7.

Kapitel 2.

Die kryptographische Sicherungsschicht

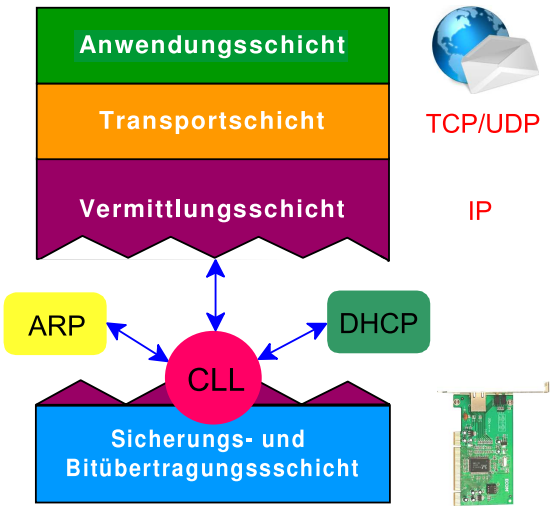

Abbildung 2.1.: CLL im Protokollstapel [8]

2.1. Vorstellung der Software

Die kryptographische Sicherungsschicht ermöglicht Nachrichtenauthentifikation und Vertraulichkeit zwischen Endsystemen im lokalen Netzwerk auf Ebene der Sicherungsschicht, das *Address Resolution Protocol (ARP)* [17] sowie das *Dynamic Host Configuration Protocol (DHCP)* [5] eingeschlossen. Damit verhindert CLL eine Reihe von

möglichen Angriffen wie *Denial of Service (DoS)*, *Man in the Middle (MiM)*, Identitätsmissbrauch als auch unerlaubtes Abhören von Nachrichten [8]. Da CLL sich transparent gegenüber existierenden Protokollen verhält, müssen keine Veränderungen am Protokollstapel vorgenommen werden. Die Software stellt über die Botan-Kryptobibliothek eine breite Auswahl an aktuellen kryptographischen Algorithmen sowie optionale Komprimierungsverfahren zur Verfügung, welche dem Sicherheitsbedürfnis angemessen zur Nutzung konfiguriert werden können. So ist es insbesondere leicht möglich, einen gebrochenen Algorithmus gegen einen als weiterhin sicher angenommenen auszutauschen oder bei Verwendung von Vertraulichkeitsmechanismen höherer Schichten aus Performancegründen auf Schicht-2-Verschlüsselung zu verzichten. CLL verwendet für die Nachrichtenauthentifikation je nach Pakettyp digitale Signaturen oder sog. Nachrichtenauthentifizierungscodes *(Message Authentification Codes)* nach dem HMAC-Verfahren [9], die optionale Verschlüsselung verwendet eine Blockchiffre im Modus *Cipher Block Chaining (CBC)*. Um *Replayangriffe* zu verhindern, werden an geeigneter Stelle sog. Noncen bzw. UNIX-Zeitstempel sowie Sequenznummern eingesetzt.

2.2. Funktionsweise von CLL im Detail

Die nachfolgenden Unterkapitel geben einen umfassenden Überblick über die wichtigsten Funktionalitäten von CLL.

2.2.1. Identifikation und Verifikation teilnehmender Endsysteme

Endsysteme werden in CLL über ihr IP/MAC-Adressenpaar identifiziert. Jeder Rechner benötigt zudem ein Schlüsselpaar aus öffentlichem und privatem Schlüssel sowie ein persönliches Hostzertifikat der zuständigen lokalen Zertifizierungsstelle *(Certificate Authority, kurz CA)*, über welches sich die Verbindung zwischen öffentlichem Schlüssel, IP- und MAC-Adresse verifizieren lässt. Um fremde Zertifikate überprüfen zu können, benötigt jeder Host zusätzlich den öffentlichen Schlüssel der CA. Im Falle einer Konfiguration für DHCP wird dem Host anstelle eines vollständigen Hostzertifikates ein Basiszertifikat ausgestellt, welches die Verbindung zwischen öffentlichem Schlüssel und MAC-Adresse überprüfbar macht. Ein Hostzertifikat wird in diesem Fall erst beim DHCP-Handshake vom DHCP-Server ausgestellt, welcher als zusätzliche CA agiert.

2.2.2. Zustand gesicherter Verbindungen: Die Sicherheitsbeziehung

Auf jedem teilnehmendem Endsystem verwaltet CLL die gesicherten Kommunikationsverbindungen über sog. *Sicherheitsbeziehungen (Security Association, kurz SA)*. Eine SA wird im Zuge eines ARP-Handshakes generiert und kann vollständig oder unvollständig sein, wobei nur über eine vollständige SA eine Kommunikation möglich ist. Die vollständige SA stellt einen verifizierten Zustand für den Nachrichtenaustausch dar, welche u. a. Informationen über die verwendeten Algorithmen und deren Konfiguration sowie die zugehörigen Schlüssel, das Zertifikat des Kommunikationspartners und verwendete Sequenznummern vorhält.

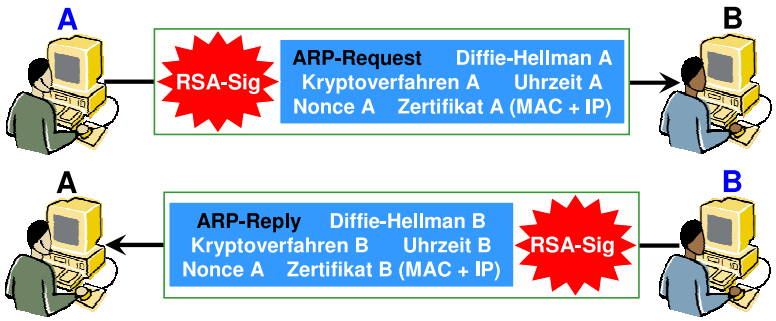

Abbildung 2.2.: ARP-Handshake, erweitert um Diffie-Hellman-Schlüsselaustausch und RSA-Signaturen [8]

Erstellung einer Sicherheitsbeziehung mittels ARP-Handshake

CLL verwendet zur Ermittlung der für die Authentifikation sowie Chiffrierung notwendigen Sitzungsschlüssel einen Diffie-Hellman-Schlüsselaustausch [18] und bedient sich zur Übertragung sowohl der kryptographischen Parameter als auch der öffentlichen Diffie-Hellman-Werte in transparenter Weise des ARP-Mechanismus. Ein bei Host A ausgehender ARP-Request wird dazu von CLL lokal abgefangen, unter anderem um das Hostzertifikat, oben genannte Parameter sowie die gewählten Diffie-Hellman-Werte erweitert, digital signiert und wie ein normaler ARP-Request als Broadcast im Netzwerk verbreitet. Sobald ein derartiger *Handshake Request* vom adressierten Host B empfangen und die Signatur über das zugehörige Zertifikat verifiziert werden konnte, wird aus den empfangenen sowie lokalen Daten eine (noch unvollständige) Sicherheitsbeziehung erstellt. Im

Anschluss wird das ursprüngliche ARP-Paket wiederhergestellt und an das ARP-Modul von Host B weitergereicht, welches einen ARP-Tabelleneintrag für Host A vornimmt und ein ARP-Reply-Paket erzeugt. Dieser *Handshake Reply* wird ebenfalls von CLL lokal abgefangen, analog zum Handshake Request erweitert, signiert und dann per Unicast an Rechner A verschickt. Die durch den Versand des Handshake Requests erstellte unvollständige SA auf Host A wird durch den Erhalt des korrespondierenden Handshake Reply nach dessen erfolgreicher Verifikation vervollständigt. Nachdem das ARP-Modul von Host A über das rekonstruierte ARP-Reply-Paket einen Tabelleneintrag im ARP-Cache vorgenommen hat, kann mittels der neuen Sicherheitsbeziehung kommuniziert werden. Host B muss allerdings zum Vervollständigen seiner SA auf das erste empfangene Datenpaket von Host A warten und hat ggf. vorher anfallende Nachrichten solange zwischenzuspeichern.
CLL verfolgt eine Cache-basierte Übertragungswiederholungsstrategie, um unnötige, rechenintensive Kryptooperationen bei Paketverlust während des Handshake zu vermeiden. Um sicherzustellen dass der ARP-Mechanismus zuverlässig für den Handshake verwendet werden kann, muss der ARP-Cache während der Initialisierung von CLL vollständig gelöscht werden, da ein bestehender Eintrag sonst den Aufbau einer Sicherheitsbeziehung verhindern würde. Der Handshake wird aus Perspektive des Initiators mittels Nonce, aus der Perspektive seines Kommunikationspartners über einen Zeitstempel gegen einen Replayangriff geschützt.

Aufrechterhaltung existierender Sicherheitsbeziehungen

Eine SA hat typischerweise eine festgelegte Gültigkeitsdauer von weniger als einer Stunde sowie eine Mindestgültigkeit von 15 min. Dadurch lässt sich (insbesondere im Fall von DHCP) auch eine Verwendung kurzlebiger Zertifikate umsetzten sowie ein Sequenznummernüberlauf verhindern. Falls während der Lebensdauer einer SA keine Datenpakete gesendet wurden, wird diese nicht für die Dauer einer weiteren Gültigkeitsperiode erneuert. Eine Neuaushandlung der SA impliziert einen erneuten Diffie-Hellman-Schlüsselaustausch sowie die daraus resultierende Ableitung neuer Sitzungsschlüssel für HMAC, Blockchiffren und Sequenznummern und geht darüber hinaus mit einem Neuaustausch sowie einer Revalidierung der aktuellen Zertifikate beider Teilnehmer einher.
Sollte Host A das Ablaufen der Sicherheitsbeziehung zu Host B feststellen und sind in der vergangenen Gültigkeitsperiode Datenpakete gesendet oder empfangen worden, so

initiiert A eine Neuaushandlung der SA in dessen Rahmen ein *Renegotiation Request* bzw. *Renegotiation Reply* ausgetauscht wird. Diese Pakete haben analogen Inhalt zum o. g. Handshake und werden über die alte SA per Unicast versendet und mittels HMAC abgesichert. Nach Erhalt des Reply steht Host A eine vollständige, neue Sicherheitsbeziehung zur Verfügung. Da nicht wie beim Handshake davon ausgegangen werden kann, dass A kurz nach Erstellen der SA Pakete an B zu versenden gedenkt, muss Rechner A zur Vervollständigung der SA auf Rechner B ein explizites *Renegotiation ACK* über die neue SA schicken. Beide Rechner haben somit gemeinsam die Neuaushandlung erfolgreich ausgeführt und können die SA nun für eine weitere Gültigkeitsperiode verwenden. Für den Fall eines Neustarts oder kurzzeitigen Ausfalls eines oder beider Kommunikationspartner speichert CLL periodisch die gültigen, aktuellen Sicherheitsbeziehungen auf der Festplatte, um eine Wiederaufnahme der Verbindung zu erleichtern.

Dynamische Zertifikatausstellung im Zusammenhang mit DHCP

Über CLL ist es möglich DHCP nicht nur gegen unautorisierte Eindringlinge, wie z. B. Hosts, die selber als (böswillige) DHCP-Server auftreten *(Rogue DHCP Server)* als auch gegen eine sog. *DHCP Starvation Attacke*, bei der Hosts durch permanente *DHCP-Requests* den Adresspool des angegriffenen DHCP-Servers blockieren, abzusichern. Dazu überwacht und modifiziert CLL auf den entsprechenden Hosts bzw. dem DHCP-Server in transparenter Art und Weise den vollständigen DHCP-Verkehr. Jedes DHCP-Paket wird von der Software um einen HMAC ergänzt und ist dadurch authentifizierbar, zusätzlich finden schichtübergreifende Konsistenzprüfungen statt.
Falls also eine dynamische Adressvergabe vorgesehen ist, wird dem betreffenden Host von der lokalen CA ein Basiszertifikat für die Benutzung seiner MAC-Adresse im Zusammenhang mit seinem öffentlichen Schlüssel ausgestellt. Für die Authentifikation des DHCP-Verkehrs mittels HMAC werden dem Client zusätzliche *Pre-Shared Keys* mitgeteilt. Dies macht einen initialen Schlüsselaustausch oder rechenintensive Digitale Signaturen überflüssig, da DHCP-Verkehr nur zwischen Clients und einem einzelnen DHCP-Server stattfindet.
Sobald dem Host eine IP-Adresse via DHCP-Offer angeboten wurde, verschickt er einen DHCP-Request erweitert um sein Basiszertifikat an den Server. Nach Konsistenzcheck sowie Authentifikation des Pakets wird vom DHCP-Server ein vollständiges Hostzertifikat für das IP/MAC-Paar und den korrespondierenden öffentlichen Schlüssel erstellt.

Der Server antwortet dem anfragenden Host über ein um das Hostzertifikat erweitertes DHCP-ACK-Paket. Der Host ist nun mittels dieses Zertifikates in der Lage eine Sicherheitsbeziehung zu anderen Hosts aufzubauen.

2.2.3. Datenverkehr über CLL

Abgesehen von ARP und DHCP sichert CLL insbesondere den normalen Unicast- und Broadcastverkehr zwischen den Endsystemen ab.

Unicast IP-Datenverkehr

Sobald eine SA zwischen zwei Hosts existiert, können beiderseits Unicast-IP-Pakete gesendet werden. Ein zu verschickendes IP-Paket wird dazu auf dem sendenden Host von CLL abgefangen, optional verschlüsselt, mit einer Sequenznummer versehen, mittels HMAC authentifiziert und danach zur Netzwerkkarte weitergeleitet. Ein derartiges Paket wird bei Empfang anhand der Sequenznummer sowie durch Überprüfung des HMAC verifiziert, ggf. entschlüsselt und nach Rekonstruktion des ursprünglichen Pakets an das IP-Modul weitergereicht. Die beim Sendevorgang inkrementell erhöhte Sequenznummer schützt vor Replayangriffen. Sie wird im Klartext übertragen, um ungültige Pakete ohne Aufwand verwerfen zu können. Um trotzdem keinen Schluss über die Anzahl der bereits gesendeten Pakete zuzulassen, wird der initiale Wert der Sequenznummer bei Erstellung oder Neuaushandlung einer SA zufällig gewählt. Jede SA speichert jeweils eine Sequenznummer für ausgehenden sowie eingehenden Verkehr.

Broadcastverkehr

Da beim Broadcastverkehr kein individueller Host angesprochen wird, sondern alle Rechner des betreffenden Subnetzes, ist das Konzept der SA an dieser Stelle nicht anwendbar. CLL verwendet zur Authentifikation von Broadcast-IP-Paketen eine digitale Signatur, zu deren Überprüfung den empfangenden Hosts das entsprechende Hostzertifikat bekannt sein muss. Dieses muss deshalb vor dem Sendevorgang - allerdings maximal einmal pro Minute - über ein separates *Certificate Packet* verbreitet werden. Ein eigenes Paket ist nötig, da damit zu rechnen ist, dass der Payload des betreffenden Broadcastpakets zu groß sein könnte, um noch das Zertifikat aufzunehmen. Um diesen Prozess zu unterstützen speichern alle Hosts des Subnetzes so empfangene Zertifikate. Broadcastpakete werden

über UNIX-Zeitstempel sowie einen Zähler gegen Replayangriffe geschützt. Der Zähler wird für jedes Paket inkrementiert, welches denselben Zeitstempel wie ein vorangegangenes erhalten hat. Die empfangenden Hosts speichern für jeden Sender den Zeitstempel und den Zähler des letzten Broadcastpakets.

2.3. Durchsatzsteigerung durch Parallelisierung

Da der Hauptaugenmerk von CLL auf der Absicherung des Unicast-Datenverkehrs liegt, konzentrieren sich auch die Anstrengungen dieser Arbeit zur Parallelisierung auf dieses Gebiet. Im Folgenden wird ein Überblick über zwei mögliche Ansätze auf jeweils unterschiedlichen Ebenen gegeben. Beide Verfahren begegnen der Problematik, dass eine Chiffrierung im CBC-Modus - wie in CLL angewandt - aufgrund der damit einhergehenden künstlichen Abhängigkeit eines Blocks von seinem Vorgänger in der verwendeten Variante nicht parallelisierbar ist [20].

2.3.1. Auf Paketebene

Für unverschlüsselte, potentiell komprimierte Daten wäre eine gleichzeitige Verarbeitung mehrerer Pakete auf jeweils unterschiedlichen CPUs ohne weitere Anstrengungen möglich, weil nur jeweils eine HMAC-Überprüfung stattfindet und es aus Sicht von CLL keine Abhängigkeiten zwischen diesen Paketen gibt. Es besteht zwar die Möglichkeit, dass auf höheren Schichten ein ungeordneter Datenstrom zu Unregelmäßigkeiten führen könnte, was allerdings in die Zuständigkeit eines reihenfolgeerhaltenden Transportprotokolls fiele. Trotzdem ist unter dem Gesichtspunkt der Optimierung die Einhaltung der Paketreihenfolge wünschenswert, um Komplikationen zu vermeiden.
Bei aktivierter Verschlüsselung verwendet CLL hingegen eine Blockchiffre im paketübergreifenden CBC-Modus, was eine Parallelisierung auf den ersten Blick ausschließen würde. Aufgrund von Toleranz gegenüber Paketverlusten stellt CLL Paketen allerdings stets den aktuellen *Initialisierungsvektor (IV)* voran und überprüft nicht, ob dieser IV tatsächlich mit dem letzten Block des vorangegangenen Paketes übereinstimmt. Aufgrund dessen kann eine parallelisierte Version von CLL abwärtskompatibel bleiben und trotzdem auf paketübergreifendes CBC verzichten, solange für jedes Paket ein eigener, zufälliger IV für paketinternes CBC verwendet wird.

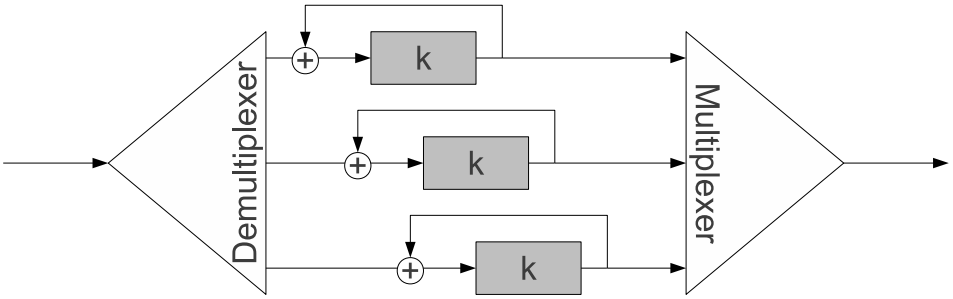

Abbildung 2.3.: Interleaved Cipher Block Chaining mit 3 Strömen

2.3.2. Auf Blockebene: Interleaved Cipher Block Chaining

Der Vollständigkeit halber soll an dieser Stelle auf ein Verfahren [20] hingewiesen werden, welches mittels Verschränkung eine parallele Anwendung des CBC-Modus zulässt. Auf Blockebene erzeugt CLL in der unmodifizierten Version einen Datenstrom welcher paketübergreifend im CBC-Modus verschlüsselt wurde.
Dieses Methode an sich lässt sich nicht parallelisieren. Es ist jedoch möglich eine Chiffrierung im CBC-Modus gleichzeitig für verschiedene Teile des Klartextes mit demselben Schlüssel vorzunehmen. So ließe sich auf einem System mit N Kernen eine Nachricht parallel verschlüsseln, indem diese in N Ströme, blockweise verschränkt, aufgeteilt wird, wobei jeder Strom separat aber gleichzeitig im CBC-Modus verarbeitet wird (siehe Abbildung 2.3). Zu Beginn werden N Initialisierungsvektoren benötigt, welche mit dem ersten Paket übertragen werden müssen.
Dieses *Interleaved Cipher Block Chaining* genannte Verfahren würde allerdings nur die Verschlüsselung beschleunigen [4], für Kompression und HMAC-Generierung lässt sich keine Verbesserung erreichen. Da es außerdem mit der Forderung nach Abwärtskompatibilität brechen würde, kommt es in dieser Form nicht zur Anwendung.

Kapitel 3.

Designüberlegungen zur Parallelisierung

3.1. Verschiedene Parallelisierungsmechanismen

Die im Folgenden vorgestellten Parallelisierungsmechanismen realisieren die nebenläufige Ausführung von Instruktionen über die Verwendung von *Threads*. Ein oder mehrere Threads existieren im Kontext eines ihnen übergeordneten Prozesses und teilen sich dessen Ressourcen, verwalten jedoch jeweils einen eigenen Zustand, Befehlszähler, Stack sowie einen Registersatz. Das Betriebssystem kann einem Thread eine freie CPU zum Ausführen von Anweisungen zuteilen *(scheduling)* [23, 2].

3.1.1. Manuelle Parallelisierung über Threads

Für diese Arbeit wurde die in CLL existierende, bereits auf Windows (mittels Win32-API) und Linux (mittels PThreads-API) portierte Abstraktion einer Threadimplementierung verwendet. Diese bietet über die Klasse `Thread` essentielle Funktionalität an. Dem Threadobjekt wird bei Konstruktion die auszuführende Funktion zugeordnet, welche dann über die Methode `Thread::Run(void*)` gestartet werden kann. Diese Funktion wird dann nebenläufig zum Hauptprogramm ausgeführt.
Darüber hinaus existieren bereits folgende Synchronisationskonstrukte:
Die Klasse `ThreadMutex` realisiert das Konzept des wechselseitigen Ausschlusses *(mutual exclusion)* über den ein *kritischer Abschnitt* geschützt wird, welcher nur von einem Thread zur Zeit betreten werden kann.
Über `ThreadEvent` und die Methoden `ThreadEvent::WaitForEvent()` sowie

`ThreadEvent::SignalEvent()` wird das bedingte Warten umgesetzt.
Die manuelle Parallelisierung eines Programmteils erfolgt nach Identifikation von parallel ausführbarem Quellcode über dessen Kapselung in einer separaten Funktion, welche einem Threadobjekt bei der Initialisierung übergeben wird. Dies ist ein nicht-trivialer Prozess [23], da z. B. der Zugriff auf gemeinsam genutzte Ressourcen mittels wechselseitigem Ausschluss geschützt werden muss, wobei es im Falle fehlerhafter Implementierung bei der Ausführung leicht zu Verklemmungen *(deadlock)* kommen kann.

3.1.2. Inkrementelle Parallelisierung mit OpenMP

OpenMP stellt eine compilerübergreifende Programmierschnittstelle [16, 10] zur inkrementellen Erweiterung existierenden Programmcodes um Nebenläufigkeit dar. Da die Schnittstelle compilerseitig implementiert ist, wird an dieser Stelle von einem hohen Optimierungsgrad der Umsetzung ausgegangen, außerdem wird durch die Benutzung keine weitere Abhängigkeit eingeführt. Zur Parallelisierung vorgesehene Code-Teile werden über eine kleine Zahl einfacher Präprozessordirektiven im vorhandenen Quelltext gekennzeichnet; der Compiler setzt diese dann nebenläufig um. Inkrementell bedeutet in diesem Zusammenhang, dass der ursprüngliche Quelltext idealerweise nicht oder nur geringfügig angepasst werden muss, so dass ein Compiler ohne OpenMP-Unterstützung die ihm unbekannten Direktiven einfach ignorieren und trotzdem ein funktionsfähiges Programm erstellen kann.
Die zentrale Direktive stellt `#pragma omp parallel` dar, welche dazu dient, den folgenden Block als parallelisierbar zu markieren. Bei Eintritt in diesen Block wird eine Gruppe aus Threads erstellt, die sich die gegebenen Aufgaben teilen. Über Laufzeitbibliotheksfunktionen lässt sich die Anzahl der zu startenden Threads bei Bedarf manuell festlegen. Interessant sind im konkreten Fall auch die `#pragma omp single` sowie `#pragma omp task` Direktiven. Omp-Single führt dazu, dass der nachfolgende Block stets nur von einem einzigen Thread durchlaufen wird, während mittels Omp-Task auch innerhalb einer Omp-Single-Umgebung eine parallel ausführbare Aufgabe definiert wird. Darüber hinaus existieren noch weitere Direktiven, z. B. zur Definition atomarer Speicherzugriffsoperationen oder kritischer Abschnitte.
Die inkrementelle Parallelisierung wurde für die Modifikation von CLL in Betracht gezogen, da jedoch die verwendeten Compiler unterschiedliche Versionen vom OpenMP unterstützen, wurde der manuellen Parallelisierung der Vorzug gelassen. Insbesondere

fehlt dem verwendeten Microsoft Visual C++ 2010 mit der OpenMP-Version 2.0 die Omp-Task Direktive, welche im Zusammenhang mit Omp-Single als besonders wichtig für eine Reihenfolge erhaltende Paketverarbeitung angesehen wird.

3.1.3. Automatische Parallelisierung

Der Vollständigkeit halber soll hier das Verfahren der Autoparallelisierung [21] erwähnt werden, über welches bestimmte Compiler angewiesen werde können, selbstständig nach parallelisierbaren Schleifenaufrufen zu suchen und diese nebenläufig umzusetzen. Da aber diese Funktionalität nicht vom verwendeten Microsoft Visual C++ 2010 Compiler unterstützt wird und das betrachtete Problem als zu komplex für eine triviale Parallelisierung angesehen wird, kommt dieses Verfahren nicht zur Anwendung.

3.2. Einführung in Threadmodelle

Zur sinnvollen Organisation der benötigten Threads gibt es verschiedene Ansätze. Die folgenden Unterkapitel widmen sich der Vorstellung und Bewertung zweier für geeignet befundener Modelle.

3.2.1. Boss-Worker-Modell

Im sog. Boss-Worker-Modell [15] delegiert eine vorgeschaltete Entität, der *Boss-Thread*, in Dauerschleife die anfallenden Aufgaben an untergeordnete *Worker-Threads*, welche diese dann nebenläufig und weitgehend unabhängig voneinander bearbeiten. Eine Verbesserung des Gesamtdurchsatzes gegenüber sequentieller Ausführung wird hier durch jeweils gleichzeitige Bearbeitung vollständiger Aufgaben durch mehrere Threads erreicht. Das Boss-Worker-Modell ist in zwei Ausführungen bekannt.
In der Variante *Thread-per-Request* wird pro Arbeitsauftrag dynamisch ein neuer Thread erstellt und nach Abschluss der Aufgabe wieder zerstört. Bei hohem Anfrageaufkommen kann dies allerdings zu Leistungseinbußen führen [19], da das Erzeugen eines Threads keine triviale Operation ist und mit einem gewissen Mehraufwand zur Laufzeit verbunden ist.
Demgegenüber wird beim *Threadpool* bereits während seiner Initialisierung eine Anzahl an Worker-Threads gestartet, welche dann für die Existenzdauer des Pools zur Verfügung

stehen. Diese so erstellten Worker-Threads pausieren ihre Ausführung nach Beendigung einer Aufgabe (*schlafen legen*) und warten auf die Signalisierung eines neuen Auftrages durch den Boss-Thread (*aufwecken*). Worker-Threads entnehmen ihre Arbeitsaufträge einer Warteschlange, welche durch den Boss-Thread befüllt wird.
Im Allgemeinen eignet sich das Boss-Worker-Modell zur Parallelisierung, solange keine oder nur geringfügige Datenabhängigkeiten zwischen den einzelnen Aufgaben bestehen [15]. Der Ansatz Thread-per-Request lässt sich verhältnismäßig leicht implementieren, ist aber aufgrund der o. g. Skalierungsproblematik nicht im hochlastigen Bereich einsetzbar. Das Konzept Threadpool hingegen ist in seiner Implementierung aufgrund einer notwendigen Verwaltung der Worker-Threads deutlich komplexer, kann aber die vorhandenen Ressourcen besser ausnutzen, da nicht fortwährend neue Threads erstellt bzw. zerstört werden müssen.

Das Boss-Worker-Threadmodell ist für den Einsatz in CLL geeignet, da die einzige aufgabenübergreifende Abhängigkeit die Forderung nach reihenfolgeerhaltender Weiterleitung bereits bearbeiteter Pakete pro SA darstellt. Hierfür wird Synchronisierungsaufwand in Kauf genommen. Aufgrund des im Netzwerkverkehr variablen, potentiell sehr hohen Anfrageaufkommens kommt hier nur die Variante als Threadpool in Frage. Die Worker-Threads können prinzipiell eingehende wie ausgehende Pakete bearbeiten.

3.2.2. Pipeline-Modell

Das sog. Pipeline-Modell funktioniert nach dem Prinzip der Fließbandarbeit [11]. Die Pipeline ist in mehrere Zwischenstufen aufgeteilt, denen jeweils ein spezifischer Teil der Gesamtaufgabe fest und in korrekter Reihenfolge zugeordnet ist. Jeweils ein Thread führt die Arbeit einer solchen Stufe aus, leitet das Resultat an die nächsthöhere Station weiter und empfängt seinerseits neue Daten von der vorangegangenen. Eine Steigerung des Durchsatzes wird durch gleichzeitiges Durchlaufen aller Stationen pro Arbeitstakt für aufeinander folgende Aufgaben erreicht [15].
Parallelisierung nach dem Pipeline-Modell ist im Allgemeinen möglich, falls sich die anfallenden Aufgaben in immer gleicher Art und Weise in feste Teilaufgaben zerlegen lassen. Die Synchronisation zwischen den Stufen ist vergleichsweise einfach zu realisieren, da stets derselbe Thread seinen zu Beginn festgelegten Nachfolger mit Daten versorgt. Für einen effizienten Betrieb wird ein ausreichend langer Strom an Eingabeda-

ten vorausgesetzt, sowie dass den einzelnen Stationen jeweils eine zeitlich gleichwertige Teilaufgabe zugeordnet wurde, da der Durchsatz der Pipeline sonst durch die langsamste Station bestimmt wird.

Auch das Pipeline-Modell ist prinzipiell für die Anwendung in CLL geeignet, da sich das Verarbeiten der Pakete (Kompression, Verschlüsselung, HMAC-Generierung) in einzelne Teilschritte zerlegen lässt. Die Einhaltung der Paketreihenfolge pro SA passiert implizit, da kein Paket in der Pipeline ein anderes „überholen“ kann. Gegen einen Einsatz in CLL spricht allerdings, dass die unterschiedlichen Teilschritte in ihrer Dauer nicht gleichwertig sind. Aufgrund der Unidirektionalität des zugrundeliegenden Konzeptes wird außerdem jeweils eine Pipeline für jede Verarbeitungsrichtung benötigt.

Kapitel 4.

Details der Implementierung

Die folgenden Unterkapitel befassen sich ausführlich mit der konkreten Implementierung einer nebenläufigen Version von CLL. Die modifizierte Variante der Software läuft im Test stabil und wurde unter Verwendung des Threadpoolmusters um parallele Verarbeitung des Unicast-Datenverkehrs erweitert. Um Störungen auf höheren Schichten zu minimieren, welche die Transparenz für existierende Protokolle schwächen könnte, wird jeweils die strikte Einhaltung der Paketreihenfolge für Unicast-Kommunikationsbeziehungen erzwungen. Die Anwendung hat sich im Test wie konzipiert als vollständig abwärtskompatibel erwiesen.

4.1. Überblick über parallelisierbaren Code

CLL verwendet einen Paketfilter auf Sicherungsschichtebene, um eingehende sowie ausgehende Pakete abfangen und intern weiterverarbeiten zu können. Dazu wird dem Paketfilter im Hauptthread der Software periodisch, sofern vorhanden, ein einzelnes Paket entnommen und je nach Ziel an `CLL::ProcessPacketTx(...)` (ausgehend) oder `CLL::ProcessPacketRx(...)` (eingehend) übergeben. Innerhalb dieser Methoden wird das Paket im Fehlerfall bereits verworfen oder je nach Typ weitergereicht, im Falle des regulären Unicast-Verkehrs an die Methoden `CLL::HandleDataTx(...)` sowie `CLL::HandleDataRx(...)`. Aufgrund des symmetrischen Aufbaus und da sich innerhalb dieser beiden Methoden parallel ausführbarer Code identifizieren lässt, konzentrieren sich die weiteren Anstrengungen konkret auf die Modifikation dieser. Es folgt eine Erläuterung der beiden Methoden sowie des parallelisierbaren Codes.

4.1.1. Ausgehende Pakete: HandleDataTx

Diese Methode hat die paketweise Verarbeitung ausgehenden Unicast-Datenverkehrs zur Aufgabe. Zu Beginn wird anhand der Ziel-Macadresse die zugehörige SA ausgewählt. Im Falle einer vorausgegangenen Neuaushandlung wird abhängig davon, ob die SA bereits vollständig ist entweder diese oder die vorangegangene verwendet. Falls die SA zum ersten Mal ausgehandelt wurde und noch unvollständig ist, muss das Paket zwischengespeichert werden und die Ausführung der Methode wird an dieser Stelle abgebrochen, da nur über eine vollständige SA kommuniziert werden kann. Im nächsten Schritt werden verschiedene Header des Paketes vorbereitet und die aktuelle Sequenznummer zugeordnet. Anschließend kommt zuerst die Kompression und dann die Verschlüsselung zur Anwendung. Das Paket wird nun mittels HMAC authentifiziert und anschließend an den realen Netzwerkadapter ausgeliefert, welcher das Paket versendet. Bevor die Methode endet, werden globale sowie SA-spezifische Statistikinformationen gespeichert.

Es bietet sich mit Blick auf die parallelisierte Ausführung eine Dreiteilung der Methode an. Initial und noch sequentiell wird zuerst die zugehörige SA sowie die Sequenznummer ermittelt. Dem folgt der für die Parallelisierung vorgesehener Teil, welcher die rechenintensiven kryptographischen Operationen sowie die optionale Verschlüsselung enthält. Im letzten, zur Erhaltung der Reihenfolge wieder sequentiellen Teil wird der eigentliche Paketversand vorgenommen. Wichtig ist an dieser Stelle, dass die Rohdaten im ersten Teil in einer Datenstruktur gespeichert werden, welche später dem verarbeitenden Thread zugänglich ist, da die Methode nach Übergabe des Arbeitsauftrages an einen Thread endet und der Speicherplatz der Daten im Hauptthread mit dem nächsten Paket überschrieben wird. Im zweiten Teil ist zu beachten, dass die Berechnung des HMAC über zwei aufeinanderfolgende Schreiboperationen auf dem HMAC-Objekt der SA vollzogen wird, es besteht hier also die Möglichkeit einer *Race Condition* [1]. Ein Schützen des Vorgangs durch einen Mutex würde hingegen dessen Parallelisierung verhindern. Außerdem ist es an dieser Stelle nötig, einen geeigneten Mechanismus zur Generierung von individuellen Initialisierungsvektoren für die Pakete anzuwenden.

[1] Mit Race Condition wird der Versuch zweier Threads bezeichnet, zur gleichen Zeit auf einen Speicherbereich zugreifen zu wollen, wobei mindestens ein Schreibvorgang getätigt wird. Das Ergebnis ist undefiniert und führt in der Regel zu schwer einzugrenzenden Fehlern

4.1.2. Eingehende Pakete: HandleDataRx

In dieser Methode wird der eingehende Unicast-Verkehr paketweise bearbeitet. Analog zu Abschnitt 4.1.1 muss zuerst die zugehörige SA über die Quell-Macadresse identifiziert werden. Die anschließende HMAC-Überprüfung dient der Authentifikation des Paketes sowie der Feststellung, ob das Paket über die aktuelle SA verschickt wurde oder über die noch existierende vorangegangene. Nach einer Überprüfung der Sequenznummer wird das Paket je nach Konfiguration entschlüsselt und dekomprimiert. Im Anschluss daran wird überprüft, ob das Paket Teil eines SA-Neuaushandlungsvorgangs ist. In diesem Fall werden die Daten an eine Methode zur Neuaushandlung, andernfalls an den IP-Stack weitergereicht. Zum Schluss findet ein Update lokaler und globaler Statistiken statt, sowie eine Statusaktualisierung der SA.

Ähnlich wie in Abschnitt 4.1.1 lässt sich auch hier eine Dreiteilung der Methode vornehmen. Der erste, sequentielle Part ermittelt die korrespondierende SA und überprüft die Sequenznummer. Im zweiten, zu parallelisierenden Teil werden die rechenintensiven Operationen wie HMAC-Überprüfung, Entschlüsselung sowie Dekompression vorgenommen sowie eine eventuelle Neuaushandlung der SA. Der letzte, wiederum sequentielle Teil ist sowohl für die Weiterleitung des Pakets an den IP-Stack, als auch für das SA-Statusupdate und die Statistiken zuständig. Problematisch in Bezug auf Nebenläufigkeit ist - wie im vorherigen Unterkapitel - an dieser Stelle wieder die HMAC-Überprüfung, ebenso wie das Sichern der Paketdaten für den zuständigen Workerthread. Zusätzlich ist die Ermittlung der zugehörigen SA im Gegensatz zur eingehenden Methode komplexer, da die Möglichkeit besteht, noch Pakete über eine alte SA zu empfangen. In diesem Fall ist eine zweite HMAC-Überprüfung nötig.

4.2. Implementierungsdetails des Threadpools

Auf Basis der in Abschnitt 3.2 herausgearbeiteten Argumente fiel die Wahl des Parallelisierungskonzeptes auf das Boss-Worker-Modell in Form eines Threadpool, da sich dieser ohne weiteres für bidirektionale Paketverarbeitung umsetzen lässt. Im Vergleich zur Pipeline ist der Threadpool mit Blick auf unterschiedlich konfigurierte Sicherheitsbeziehungen flexibler und kann die vorhandenen Ressourcen effektiver ausnutzen. Ein Paket, welches z.B. zur Verschlüsselung aber nicht zur Kompression vorgesehen wurde,

müsste die entsprechende Stufe einer Pipeline trotzdem durchlaufen.
Die folgenden Unterkapitel widmen sich den Implementierungsdetails des Threadpools im Zusammenhang mit einer Paketwarteschlange, welche als Paketpuffer und zur Erhaltung der Reihenfolge innerhalb einer Kommunikationsbeziehung eingeführt wurde.

4.2.1. Aufbau

Im Zuge der Initialisierung des Threadpool wird zuerst die Poolgröße festgelegt, welche die Anzahl der Workerthreads bestimmt. Dieser Wert wird, falls nicht explizit angegeben, auf die Anzahl der Kerne des vorhandenen Prozessors festgelegt und bleibt für die Dauer der Ausführung konstant. Im nächsten Schritt werden der Poolgröße entsprechend Workerthreads initialisiert und gestartet. Mit *Arbeitsauftrag* wird im Folgenden die Anweisung zur Bearbeitung eines einzelnen Unicast-Pakets in einem Workerthread bezeichnet. Unter *geschütztem* Zugriff wird ein Lese- oder Schreibvorgang verstanden, welcher über wechselseitigen Ausschluss gegen Race-Conditions gesichert wird.

Kommunikationsdatenstruktur zwischen Boss und Worker

Die Workerentity stellt eine jedem Thread fest zugeordnete Datenstruktur dar, welche zu dessen Instruktion verwendet wird. Dort werden vor jedem Arbeitsauftrag alle in diesem Zusammenhang relevanten Daten vom Bossthread für den jeweiligen Workerthread abgelegt. Durch die feste Zuordnung einer Workerentity zu einem Workerthread kann auf zeitraubende Synchronisierung zur Bearbeitungszeit der Aufgabe verzichtet werden, da sichergestellt ist, dass sich die übergebenen Daten nach Anstoß des Auftrages nur noch innerhalb des Threads ändern.

Zuweisung eines Arbeitsauftrages

Die Methode `ThreadPool::StartJob(...)` dient dazu, einen Arbeitsauftrag an einen Workerthread zu delegieren. Im Detail muss dazu ein untätiger Thread ermittelt werden, welcher über die ihm zugeordnete Workerentity instruiert und im Anschluss daran aufgeweckt wird. Der Threadpool hält, geschützt durch wechselseitigen Ausschluss, den Aktivitätszustand seiner Workerthreads in einem Bitset vor. Zusätzlich steht (ebenfalls geschützt) die Anzahl der aktuell untätigen Threads zur Verfügung, so dass sich schnell und einfach überprüfen lässt, ob ein Auftrag vergeben werden kann oder nicht.

Falls dies nicht der Fall sein sollte, blockiert die Methode und wartet auf Signalisierung aus einem der Workerthreads. Um die Handhabung der sehr allgemeinen Startjob-Methode zu vereinfachen, gibt es für unterschiedliche Aufgaben je eine eigene Wrappermethode, im Folgenden zusammenfassend mit *AssignJob-Methoden* bezeichnet.

Der Workerthread

Im Workerthread findet die eigentliche Unicast-Paketverarbeitung statt. Jedwede als parallelisierbar identifizierte Operation aus einer der HandleData-Methoden ist hier untergebracht worden, wobei ein Thread bidirektional konzipiert ist und somit je nach aktuellem Auftrag eingehende oder ausgehende Daten verarbeiten kann. Nach dem ersten Start tritt der Workerthread in eine Dauerschleife ein und legt sich, ebenso wie nach erfolgreicher Beendigung eines Auftrages, schlafen. Aus diesem Zustand wird der Workerthread bei Bedarf aus dem Boss-Thread des Programms über das Erteilen eines Arbeitsauftrages mittels einer der für `ThreadPool::StartJob(...)` vorgesehenen Wrappermethoden wieder aufgeweckt. Nach erfolgreicher Bearbeitung des Auftrages ändert der Thread seinen Zustand im Bitset des Threadpools geschützt auf „untätig". Er überprüft, ob bereits andere Threads untätig sind. Ist das nicht der Fall signalisiert er dem Boss-Thread, dass wieder ein Workerthread frei ist und legt sich im Anschluss daran schlafen. Einen Sonderfall stellt hier der Auftrag vom Typ *end* dar. Ein Thread erhält auf diesem Wege die Anweisung, sich zu beenden und verlässt die Endlosschleife. Damit im Boss-Thread beim Beenden keine Verklemmung auftreten kann, signalisiert der letzte sich beendende Thread, die Blockade aufzuheben.

4.3. Implementierungsdetails der Paketwarteschlange

Die Paketwarteschlange stellt einen Mechanismus dar, über den sowohl Paketdaten zwischengespeichert werden, um in einem Workerthread zur Verfügung zu stehen, als auch das Reihenfolge erhaltende Versenden über die Netzwerkkarte bzw. Weiterreichen an den IP-Stack ermöglicht wird. Die Warteschlange ist in Form zweier symmetrischer Ringpuffer implementiert, so dass einerseits Paketdaten von außen an einen Workerthread weitergegeben, andererseits aber auch die fertig bearbeiteten Paketdaten von einem anderen Thread versendet werden können. Über diese Funktionalität wird die Verarbeitung der Daten und deren Weiterleitung (an die Netzwerkkarte oder den IP-Stack) im Sinne der

Reihenfolgeerhaltung entkoppelt, so dass nicht ein rechenintensives Paket den Thread eines darauf folgenden eventuell sehr kleinen Paketes blockieren kann. Unter *Paketversand* wird im Folgenden entweder der Versand von fertigen Paketen über die Netzwerkkarte oder die Weiterleitung an den IP-Stack des Betriebssystems verstanden. Diese Generalisierung ist möglich, da die Warteschlange eingehende wie ausgehende Pakete nahezu gleich behandelt.

4.3.1. Aufbau

Die Paketwarteschlange besteht wie eingangs erwähnt aus zwei Ringpuffern mit fester Größe, Quell- und Zielpuffer, deren Speicher jeweils über denselben Index angesprochen wird. Darüber hinaus wird der Zustand der Speicherplätze über zwei geschützte Bitfelder erfasst, welche anzeigen, ob ein Bereich gerade in Benutzung ist oder bereits zum Versand bereit. Weiterhin wird die Position des nächsten zu vergebenden Speicherplatzes (Positionszeiger) und des ersten zu versendenden Speicherplatzes (Sendezeiger) geschützt festgehalten. Alle Inkrement-Operationen auf diesen Zeigern finden modulo Warteschlangengröße statt.

Zugriff auf Speicherplatz

Ein Zugriff auf den Speicherplatz erfolgt zuerst über den Versuch einer Reservierung eines Platzes in der Warteschlange. Die Methode `PacketQueue::ReserveBuffer()` gibt, falls der nächste zuzuweisende Platz frei ist, dessen Index zurück oder blockiert andernfalls. Der zugehörige Speicherplatz wird dabei geschützt als „in Benutzung“ markiert, ebenfalls geschützt wird der Positionszeiger inkrementiert.
Die Methode `PacketQueue::GetBuffers(...)` weist über einen angegebenen Index zwei Pointern jeweils den korrespondierenden Quell- sowie Zielspeicherplatz zu, welcher vorher reserviert wurde. Diese Operation wird, um Synchronisationsaufwand zu minimieren, nicht geschützt. Es existiert die Konvention, dass nur ein erfolgreich reservierter Bereich über dessen Index mittels dieser Methode manipuliert wird und dass nicht ein Workerthread auf den Speicherbereich eines anderen Threads zuzugreifen versucht, solange dieser noch in Benutzung ist. Da die Reservierung an sich geschützt wurde, kann also zuerst der Hauptthread den so erreichbaren Quellpuffer mit den Paketdaten initialisieren und danach einem Thread einen Arbeitsauftrag erteilen, was eine Übergabe des dem reservierten Platz zugeordneten Indexes einschließt. Der Thread wiederum kann

nach dem Aufwachen über diese Methode den ihm zugeordneten Speicher benutzten, ohne dass eine weitere Synchronisation nötig wäre.

Reihenfolgeerhaltender Paketversand

Sobald ein Thread mit der Arbeit auf seinem Speicherbereich fertig ist, markiert er den Bereich geschützt als „versendebereit“ und „nicht in Benutzung“. Im Falle eingehender Pakete kann es außerdem vorkommen, dass ein Paket verworfen werden soll, der Thread markiert dazu den Speicherplatz geschützt zusätzlich noch als „verworfen“.
Dem Paketversand unter Erhaltung der Reihenfolge geht folgende Konvention voraus: Es darf pro SA nur ein Thread zur Zeit Pakete senden; es beginnt stets der Thread, dessen Speicherplatzindex mit dem Sendezeiger übereinstimmt. Der Sendethread übermittelt nacheinander alle folgenden, als „versendebereit“ und nicht „verworfen“ markierten Pakete, währenddessen wird der Sendezeiger aktualisiert.
Diese Funktionalität ist in `PacketQueue::TrySend(...)` untergebracht, welche von jedem Workerthread am Ende einer Aufgabe aufgerufen wird. Intern wird überprüft, ob der Thread in der Pflicht wäre zu senden oder ob bereits ein anderer Thread sendet. In beiden Fällen wird die Methode wieder verlassen und der Thread legt sich schlafen, da bereits ein anderer Thread mit dem Versenden der Daten für diese SA beauftragt ist. Andernfalls sendet der Thread solange wie möglich fertiggestellte Pakete und löscht alle Markierungen der zugehörigen Speicherplätze, gibt diese also wieder frei.

Initialisierungsvektoren aus alten Paketen

Wie in Abschnitt 4.1.1 bereits ausgeführt ist es nötig, für jedes ausgehende Paket einen individuellen Initialisierungsvektor (IV) zu generieren. Um nicht bei jedem Paket eine Zufallsfunktion aufrufen zu müssen, wird hier eine Eigenschaft des Ringpuffers ausgenutzt: Die fertigen Pakete bleiben an ihrer Stelle im Zielpuffer erhalten, selbst wenn der Speicherplatz schon wieder frei gegeben wurde, da die Daten nicht explizit gelöscht werden. Es ist für einen Workerthread also ohne weiteres möglich, den IV für sein Paket aus dessen Zielpuffer zu extrahieren, weil dort noch ein altes Paket liegt, dessen Daten verwendet werden können. Um sicherzustellen, dass dieses Verfahren zu jedem Zeitpunkt funktioniert, werden alle Zielpufferplätze bei der Erstellung einer Warteschlange für ausgehende Pakete einmalig mit zufälligen Daten befüllt.

4.4. Parallelisierung des ursprünglichen Codes

Es soll nun näher ausgeführt werden wie CLL unter Zuhilfenahme des Threadpool und der Paketwarteschlange erfolgreich parallelisiert werden konnte. Die zentrale Klasse `CLL` wurde um einen Threadpool erweitert. Bei Programmstart wird dieser mit Werten aus der CLL-Konfigurationsdatei bzw. im Falle fehlender oder ungültiger Angaben mit sinnvollen Standardwerten für die Anzahl der Workerthreads initialisiert. Der Hauptthread von CLL übernimmt nun gegenüber den Poolthreads die Rolle des Boss-Threads. Um einen Reihenfolge erhaltenden Paketversand innerhalb einer Kommunikationsbeziehung zu ermöglichen, wurde die existierende Klasse `SecurityAssociation` um jeweils eine Paketwarteschlange in ausgehender sowie eingehender Richtung erweitert. Da nach vollständiger Neuaushandlung einer SA ausgehende Pakete nur noch über die neue SA versendet werden, erhält diese bei Initialisierung eine neue, eigene Paketwarteschlange. Da in eingehender Richtung immer noch Pakete über die alte SA ankommen können, teilen sich der Einfachheit halber alte und neue SA eine Paketwarteschlange; so wird Mehraufwand durch zusätzliches Umkopieren der Paketdaten vermieden.
Die beiden HandleData-Methoden wurden in ihrem Funktionsumfang stark verkleinert. Sie enthalten jeweils nur noch den bereits erwähnten (leicht umstrukturierten) sequentiellen Anteil der Ursprungsmethode und stoßen über eine der in Abschnitt 4.2.1 erwähnten AssignJob-Methoden einen Arbeitsauftrag an, nachdem sie einen Platz in der aktuellen SA zugehörigen Paketwarteschlange reserviert und die Paketdaten dorthin übertragen haben.
Der Workerthread übernimmt nun abhängig vom Aufgabentyp (empfangen und weiterleiten, senden, verschlüsseln und senden etc.) die vormals in den HandleData-Methoden als parallelisierbar identifizierte Funktionalität. Die in Abschnitt 4.1.1 angesprochene Problematik bei der Parallelisierung der HMAC-Generierung/Überprüfung wurde über Replikation gelöst. Jede SA hält für jeden Workerthread ein eigenes, zum original identisches HMAC-Objekt vor, so dass an dieser Stelle keine Race-Conditions auftreten können

Kapitel 5.

Evaluation

Die anschließenden Unterkapitel befassen sich mit den verschiedenen im Rahmen einer Qualitätsauswertung vorgenommenen Messungen, sowie deren Vergleich und Interpretation. Die Werte wurden auf zwei identisch konfigurierten Rechnern, im Folgenden mit *PC-A* und *PC-B* bezeichnet, erhoben, welche jeweils mit dem Vierkernprozessor Intel Xeon X3360, 2.83GHz [1], 8 GB RAM und gigabitfähiger Netzwerkkarte ausgestattet waren. Beide Rechner verfügten jeweils über die Betriebssysteme Windows 7 (64 Bit) sowie Ubuntu 11.04 (64 Bit) um eine Berücksichtigung der Testplattform in die Bewertung einfließen lassen zu können. Die angezeigten Messwerte stellen jeweils das arithmetische Mittel der real vorgenommenen Messungen dar. Alle in Tabellenform vorhandenen Daten werden in Anhang A zusätzlich noch grafisch dargestellt.

5.1. Durchsatzmessung im Netzwerk

Der erste vorgenommene Test soll einen Vergleich der ursprünglichen Version von CLL mit der modifizierten ermöglichen. Dazu wurden mehrere Messungen unter beiden Betriebssystemen via Gigabit-Ethernet, jeweils mit alter und neuer Version von CLL mit unterschiedlicher Threadanzahl unter Zuhilfenahme des Netzwerkbenchmarkprogramms Nuttcp [6] durchgeführt. Die Messungen dauerten jeweils 10 Sekunden; es wurde stets über 5 Messungen gemittelt. Die in Abbildung 5.1 dargestellten Resultate zeigen, dass die modifizierte Version von CLL unter beiden Betriebssystemen tatsächlich von den Parallelisierungsmechanismen profitieren kann. Wie für Vierkernprozessoren erwartet, steigen die Werte bis zur Verwendung von 4 Threads stark an. Das leichte Abflachen bei 16 Threads wird mit einer zwar besseren CPU-Auslastung aber stärkerem Scheduling-

aufwand gerechtfertigt, da mehr Threads unter Hochlast arbeiten als CPUs zur Verfügung stünden. Besonders auffällig an den Messergebnissen ist das deutlich schlechtere Abschneiden der Werte unter Linux im Vergleich zu Windows. Weiterhin liefert die modifizierte Version von CLL im Bereich 1-2 Threads allgemein sehr schlechte Ergebnisse.

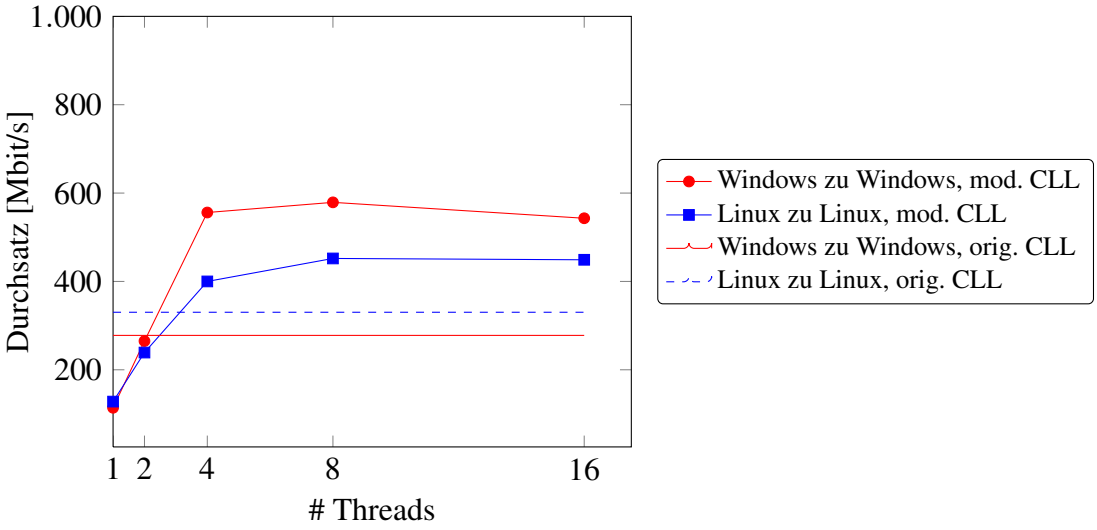

Abbildung 5.1.: Durchsatzvergleich im Gigabit-Netzwerk

5.2. Nachvollziehen der Ergebnisse in zweiter Messung

Aufgrund der im vorigen Unterkapitel festgestellten Performanceprobleme im Bereich kleiner Threadzahlen wird eine zweite Messung vorgenommen. Dazu wird in einer stark vereinfachten Testumgebung ein CPU-Benchmark lokal auf jeweils einem Rechner durchgeführt, welcher den Durchsatz anzeigen soll, den die CPU mit dem verwendeten Verfahren erreichen könnte. Es werden dazu ein sequentielles sowie ein durch den entwickelten Threadpool parallelisiertes Verfahren und eine stark vereinfachte, über OpenMP parallelisierte Variante gegenübergestellt. Um den Focus auf die Qualität der Verfahren zu legen, wurden Anforderungen wie Reihenfolgeerhaltung vernachlässigt. Die zu parallelisierende Aktion wird einzig durch eine Verschlüsselungsoperation im CBC-Modus auf zufälligen Daten fester Grüße repräsentiert, welche wiederholt ausgeführt wird. Daraus ergibt sich über eine Zeitmessung ein Durchsatzwert, der als Anhaltspunkt zur Qualitätsanalyse dient. Exemplarisch werden dazu die Messwerte von PC-B in Tabelle 5.1

betrachtet. Die Ergebnisse aus der Testumgebung sind jeweils über 20 Durchläufe gemittelt. Es wurde pro Test ein Datenvolumen von 100000 * 1024 Byte verarbeitet.
Wieder fällt auf, das die Threadpoolwerte beider Betriebssysteme weit unter denen der sequentiellen ebenso wie der OpenMP-Variante starten, wobei die Windowsversion zumindest im Bereich hoher Threadanzahl eine gute Leistung zeigt. Weiterhin steigt der über den Threadpool erzielte Durchsatz unter Linux bei höherer Threadzahl generell wesentlich schwächer an als unter Windows, was den Schluss zulässt, dass dessen Implementierung zumindest für Linux einer Überarbeitung bedarf. Die OpenMP-Werte steigen erwartungsgemäß bis zur Verwendung von 4 Threads jeweils an und halten den Wert bzw. sinken danach minimal. OpenMP wird hier nur als Orientierung verwendet um einen ungefähren Indikator für eine Lösung in Nähe des erreichbaren Optimums zu bieten; die Werte lassen sich nicht ohne weiteres auf das reale CLL übertragen.

Anzahl Threads	1	2	4	8	16
Sequentiell, Linux	157,90				
Threadpool, Linux	30,04	140,55	152,05	211,80	168,65
OpenMp, Linux	159,60	184,95	230,35	214,95	203,00
Sequentiell, Windows	162,73				
Threadpool, Windows	23,42	90,18	203,10	306,90	361,80
OpenMp, Windows	161,30	241,00	373,20	373,10	373,85

Tabelle 5.1.: Ergebnisse des CPU-Benchmarks, Angaben in [MB/s]

5.3. Laufzeitanalyse der Testumgebung

Aufgrund der durch die vorangegangenen Messungen belegten geringen Leistungsfähigkeit der Threadpoolimplementierung im Bereich kleiner Threadzahlen unter Windows sowie insgesamt unter Linux erschien es sinnvoll, eine abgewandelte Implementierung in Betracht zu ziehen. Um den problematischen Code zu identifizieren wurde eine Laufzeitanalyse der einzelnen Verfahren der Testumgebung unter Linux vorgenommen. Die verwendete Software Google-Perftools [7] zeigt auf, wie groß der prozentuale Anteil der Gesamtausführungszeit pro individueller Methode ist.
Ein Vergleich der Ergebnisse aller drei Verfahren (siehe Abbildungen A.3 - A.5) der Testumgebung mit jeweils einem Thread gibt einen Hinweis auf eine mögliche Verbesserung der Threadpoolimplementierung: Während das sequentielle sowie das OpenMP-

Verfahren jeweils ca. 80% der Ausführungsdauer mit der Verschlüsselung verbringen, erreicht der Threadpool nur 50%. Auffällig ist dabei, dass der Pool zu 16% mit der Eventsignalisierung und zu 23% mit dem Warten auf Events zubringt. Dies lässt den Schluss zu, dass die Verzahnung des Boss-Threads mit den zugehörigen Workerthreads zu eng sein könnte, was zu einer abgewandelten Version des Threadpool in der Testumgebung motiviert.

5.3.1. Anpassungen des Threadpools

Der aktuelle Threadpool verschwendet wertvolle Zeit mit Warten und Signalisierung, da ein Workerthread sich nach Abschluss eines Auftrages in jedem Fall schlafen legt, um danach wieder aufgeweckt werden zu können, sobald eine neue Aufgabe ansteht. Bei hohem Anfrageaufkommen kann dies einen Flaschenhals in Bezug auf den Gesamtdurchsatz darstellen.
Um diese Kopplung der Threadpoolkomponenten in sinnvollem Maße zu lockern, wurde der Pool um eine dynamische Aufgabenliste erweitert. Diese wird bei Erteilen eines Arbeitsauftrages geschützt mit dem jeweiligen Auftrag befüllt. Falls es einen untätigen Workerthread geben sollte, wird dieser im Anschluss daran aufgeweckt. Danach fährt der Boss-Thread fort, die Aufgabenliste zu befüllen und gegebenenfalls Threads aufzuwecken, bis diese eine festgelegte Größe erreicht hat. Die Liste gilt nun als voll und der Hauptthread wartet auf Signalisierung aus einem Workerthread. Um Verklemmungen zu vermeiden, überprüft der Boss-Thread periodisch, ob die Liste noch voll ist oder ob Platz für den nächsten Auftrag vorhanden ist.
Die Workerthreads wurden dahingehend modifiziert, das sie nach einmaligem Anstoß kontinuierlich Aufträge geschützt annehmen und bearbeiten, solange die Auftragsliste nicht leer ist. Sollte ein Thread feststellen, dass keine Aufträge mehr vorhanden sind, markiert er sich geschützt als „untätig“ und wartet auf Signalisierung aus dem Boss-Thread. In periodischen Abständen erwacht der Workerthread und überprüft geschützt den Zustand der Aufgabenliste. Sollte ein Auftrag vorhanden sein, kann der Thread auch ohne Signalisierung mit der Bearbeitung beginnen.

5.4. Auswertung der Modifikation in dritter Messung

Um einen Vergleich der alten mit der neuen Threadpoolimplementierung zu ermöglichen, wird ein weiterer CPU-Benchmark durchgeführt, sowie eine weitere Laufzeitanalyse durchgeführt. Die in Tabelle 5.2 sowie Abbildung A.1 und A.2 wiedergegebenen Messergebnisse zeigen allgemein eine deutliche Verbesserung auf beiden Plattformen bis 4 Threads auf. Der modifizierte Threadpool liefert unter Linux im Bereich 2 bis 8 Threads eine sehr gute Leistung, unter Windows hingegen knickt die Kurve nach 4 Threads ein und wird von der Originalimplementierung des Pools übertroffen. Die Messergebnisse lassen darauf schließen, dass sich die in Abbildung 5.1 dargestellten Netzwerkdurchsatzmessungen mit dem modifizierten Threadpool verbessern ließen. Für den Linux-Fall bestehen keine Zweifel, da sich der neue Threadpool der ursprünglichen Implementierung gegenüber in fast allen Messungen als leistungsfähiger erwiesen hat, sowie auch in der Laufzeitanalyse eine sichtbare Verbesserung erzielen konnte (siehe Abbildung A.6). Im Windows-Fall schneidet die neue Variante zwar im Vergleich ab mehr als 4 Threads schlechter ab, die Netzwerkdurchsatzmessung hat aber wiederum gezeigt, dass sich bei mehr als 4 Threads kaum eine nennenswerte Verbesserung erzielen lässt. Die Tatsache, dass die Ergebnisse der Threadpoolimplementierungen unter Windows sehr weit auseinandergehen und der modifizierte Pool unter Linux eine höhere Leistung erbringt als der unmodifizierte, wird mit der unterschiedlichen Implementierung der Synchronisations- und Nebenläufigkeitsmechanismen auf den beide Plattformen gerechtfertigt. Die Werte der stark auf der Verwendung von Mutexen beruhende modifizierte Version des Pools lassen den Schluss zu, dass die unter Linux verwendeten PThread-Mutexe effizienter implementiert wurden, als die über Pipes realisierten Events. Auf der anderen Seite können die Ergebnisse des unmodifizierten Threadpools, welcher stark auf Events basiert, einen Indikator dafür darstellen, dass Events unter Windows effizienter Arbeiten als Mutexe.

Anzahl Threads	1	2	4	8	16
mod. Threadpool, Linux	121,50	179,25	239,00	250,00	128,35
mod. Threadpool, Windows	137,95	219,55	236,60	112,25	85,84

Tabelle 5.2.: Ergebnisse des zweiten CPU-Benchmarks, Angaben in [MB/s]

Kapitel 6.

CLL als Dienst unter Windows

Auf Windowsplattformen wird mit einem *Dienst* eine Software bezeichnet, die für eine lange Laufzeit vorgesehen ist und währenddessen keine Benutzerinteraktion erfordert [14], ähnlich dem UNIX-Daemon. Ein Windows-Dienst wird im Hintergrund als Konsolenanwendung ohne Benutzeroberfläche ausgeführt und kann über ein zentrales Interface, den *Service Control Manager (SCM)*, gesteuert werden. Da unter Linux bereits die Möglichkeit besteht, die Software als Daemon zu betreiben und die genannten Anforderungen an eine Dienstanwendung von CLL erfüllt werden, ist dessen Erweiterung um eine Dienstfunktionalität unter Windows wünschenswert.

6.1. Struktur eines Windows-Dienstes

Ein Dienst folgt unter Verwendung des Win32-API einer vorgegebenen Grundstruktur [13]. Zu Beginn wird ein *Service Control Dispatcher* Thread gestartet, welcher sich mit dem SCM verbindet und die *Service Main* Funktion der Anwendung als Argument übergeben bekommt. Dieser Thread läuft für die Dauer der Ausführung des Programms und stellt die Kommunikationsschnittstelle zwischen Dienst und SCM dar. Im Anschluss wird die angegebene Service Main Funktion gestartet, welche die zugehörige *Service Control Handler* Funktion beim Service Control Dispatcher registriert und die *Service Code* Funktion startet, welche die eigentliche Funktionalität des Dienstes beinhaltet. Die Service Control Handler Funktion wird bei Bedarf aufgerufen, um die vom SCM an den Dienst gesendeten Kommandos zu verarbeiten. Der SCM wird während der Laufzeit mittels `SetServiceStatus(...)` über jede Statusänderung informiert.

6.1.1. Der Service Control Manager

Der Service Control Manager stellt ein einheitliches Interface zur Steuerung von Diensten dar. Dienstanwendungen müssen dazu initial über eine API-Funktion in die Datenbank des SCM eingetragen werden; dieser Vorgang wird im Folgenden *Installation* genannt. Ein derartiger Eintrag enthält im Wesentlichen den Namen des Dienstes, Zugriffsrechte, den Typ des Dienstes sowie dessen Starttyp (z. B. manuell oder automatisch). Eingetragene Dienste werden je nach Konfiguration automatisch gestartet oder manuell über den SCM. Parameter werden entweder beim manuellen Start über den SCM übergeben oder müssen für den automatischen Start bei der Installation bzw. Modifikation des SCM-Datenbankeintrags im Pfad der Anwendung enthalten sein.

6.1.2. Der Ereignisprotokolldienst

Fehler und Meldungen den Betrieb als Dienst betreffend können über den *Ereignisprotokolldienst (Event-Log)* des Betriebssystems kommuniziert werden. Während der Installation wird dazu eine - üblicherweise als *Dynamic Linked Library* vorliegende - *Message Text File* für die Anwendung registriert, über welche die verwendeten Meldungen definiert werden. Im Auslösungsfall wird vom Ereignisprotokolldienst auf Basis dieser Datei sowie gegebenenfalls vorhandener Parameter ein Eintrag im Anwendungsprotokoll vorgenommen.

6.2. Details der Implementierung

Zur Realisierung von CLL als Dienst wurde eine Wrapperklasse um die zahlreichen Win32-API-Funktionen geschrieben, um die entsprechende Funktionalität zu kapseln. Diese Klasse `NTService` wurde nach dem Singleton-Muster[1] [12] entworfen, da innerhalb der Anwendung nur eine einzige Instanz der Klasse sinnvoll ist. Die unter Linux bereits verwendeten Methoden zum Mitloggen der Statistikinformationen wurden minimal abgewandelt wiederverwendet. Eine persistente Parameterübergabe an die Anwendung lässt sich vor dem Start über den SCM mit Hilfe der Kommandozeile realisieren, so dass diese auch beim automatischen Start zur Verfügung stehen.

[1] Das Singleton-Entwurfsmuster dient dazu sicherzustellen, dass von einer Klasse nicht mehr als ein Objekt erstellt werden kann. Die einzige Instanz ist über eine statische Methode erreichbar.

Kapitel 7.

Resümee und Ausblick

Im Rahmen dieser Ausarbeitung wurde die Erweiterung der gegebenen kryptographischen Sicherungsschicht um Parallelisierungsmechanismen vorgestellt. Es konnte erfolgreich die Anwendung des Threadpoolmusters im Zusammenhang mit einer Paketwarteschlange zur Reihenfolgeerhaltung umgesetzt, sowie den aus Nebenläufigkeit resultierenden Synchronisierungsanforderungen begegnet werden. Die modifizierte Software läuft im Test stabil und konnte abwärtskompatibel umgesetzt werden.
Die in Kapitel 5 dargelegten Resultate belegen, dass CLL von der Parallelisierung profitieren kann. Die manuell um den ursprünglichen Threadpool erweiterte Version der Software liefert dafür eine gute Grundlage, da im Netzwerkdurchsatztest auf der verwendeten Hardware - je nach Plattform - eine Leistungssteigerung um den Faktor 1,4 - 2 messbar ist. Um einen qualitativen Vergleich zu ermöglichen, wurde der verwendete Threadpool in einer vereinfachten Testumgebung in Relation zu einer hypothetischen OpenMP-Lösung gesetzt. Auf Basis dieser Messergebnisse sowie nach weiterer Effizienzanalyse eröffnete sich in der Testumgebung eine Perspektive zur weiteren Optimierung der Software über eine modifizierte Version des Threadpools.

Weiterhin konnte CLL erfolgreich um die Funktionalität, als Windows-Dienst ausgeführt zu werden, erweitert werden. Die Software kann dazu über die Kommandozeile als Dienst installiert und konfiguriert werden und steht danach beim Systemstart automatisch zur Verfügung

Eine mögliche Weiterentwicklung der angewandten Parallelisierungsmaßnahmen würde in der Umsetzung des modifizierten Threadpools für die Endanwendung bestehen. Die vielversprechenden Ergebnisse aus der Testumgebung lassen auf bessere Resultate im

Bereich kleiner Threadzahlen sowie eine allgemein erhöhte Leistung hoffen. Auch der Versuch einer konkreten Parallelisierung der Software über OpenMP könnte die bereits geleistete Arbeit sinnvoll ergänzen.

Anhang A.

Zusätzliche Grafiken

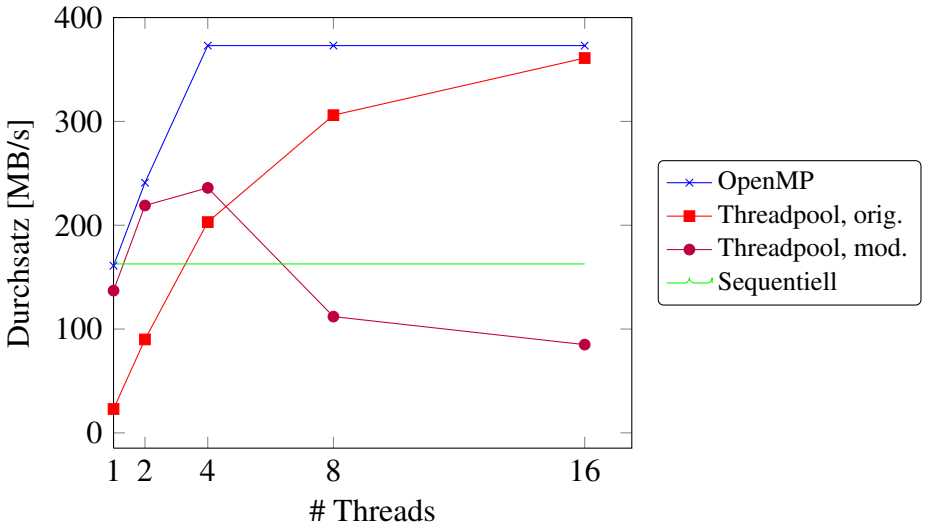

Abbildung A.1.: Grafische Darstellung der Resultate der beiden CPU-Benchmarks (PC-B, Windows)

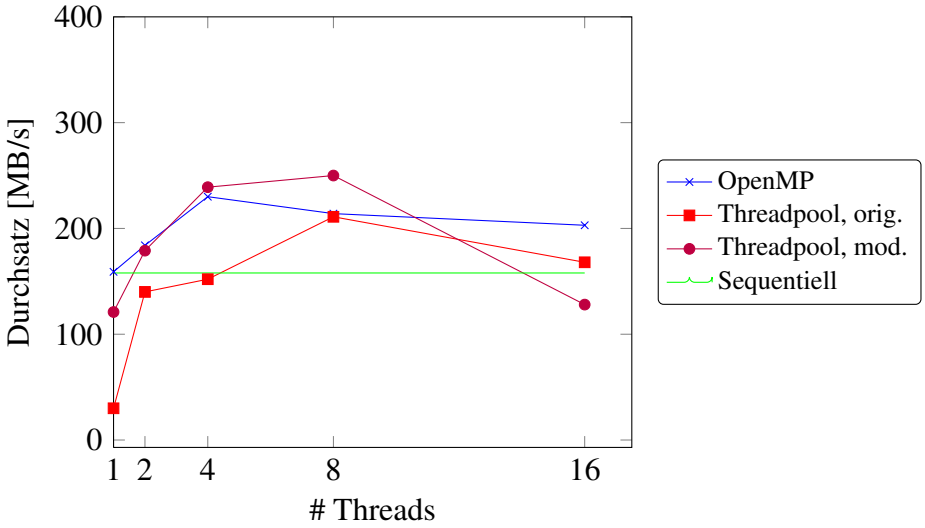

Abbildung A.2.: Grafische Darstellung der Resultate der beiden CPU-Benchmarks (PC-B, Linux)

../../src_tools/bin/CLL-SpeedProfile_New
Total samples: 153
Focusing on: 153
Dropped nodes with <= 0 abs(samples)
Dropped edges with <= 0 samples

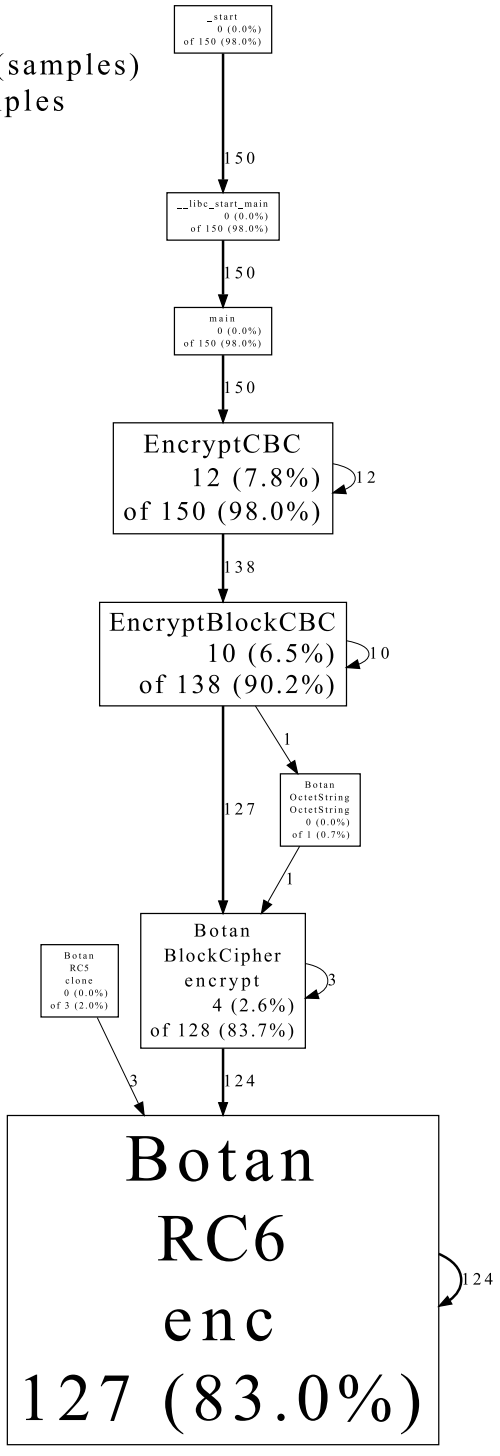

Abbildung A.3.: Laufzeitanalyse der sequentiellen Lösung mit einem Thread unter Linux

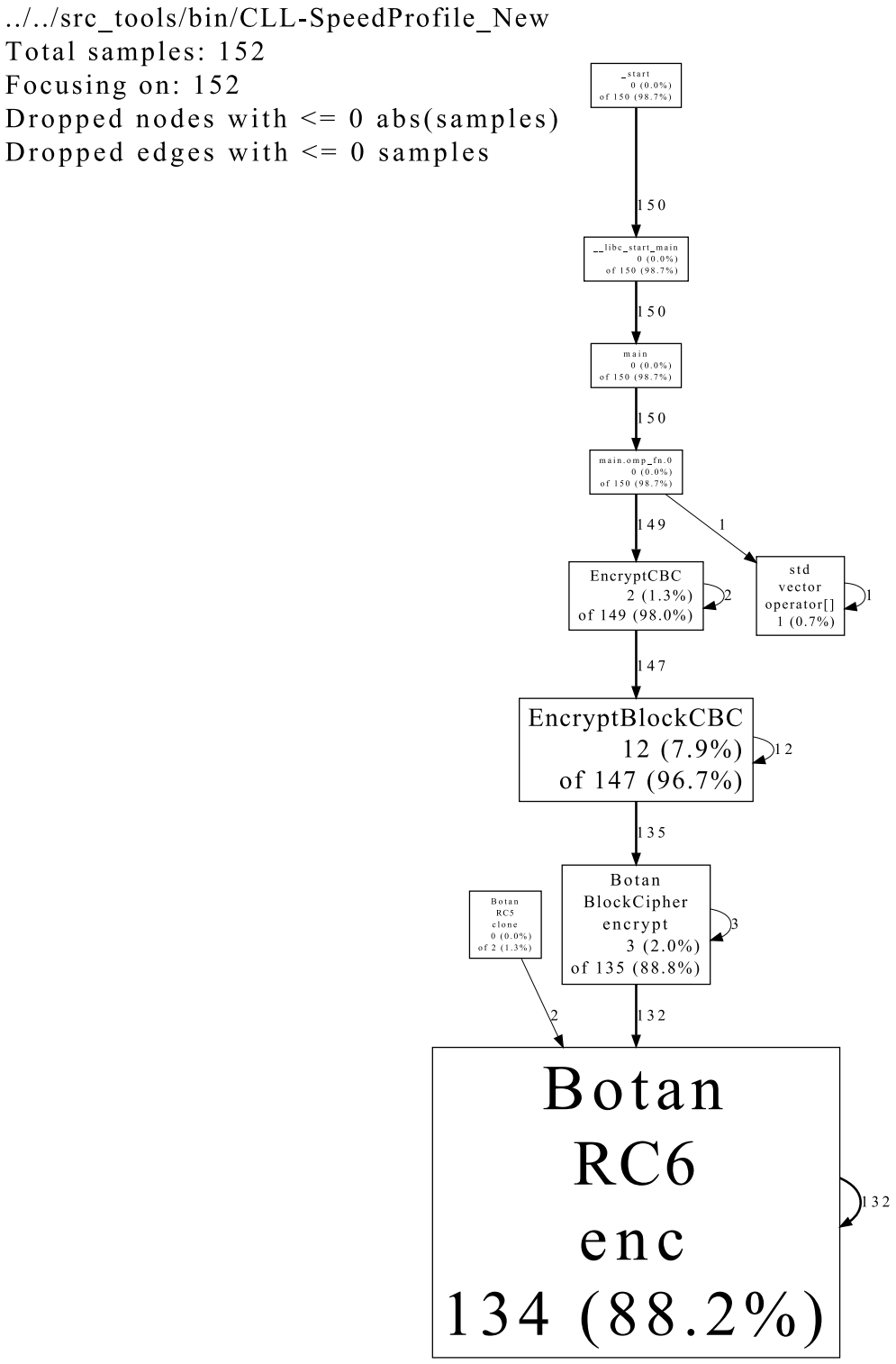

Abbildung A.4.: Laufzeitanalyse der OpenMP-Lösung mit einem Thread unter Linux

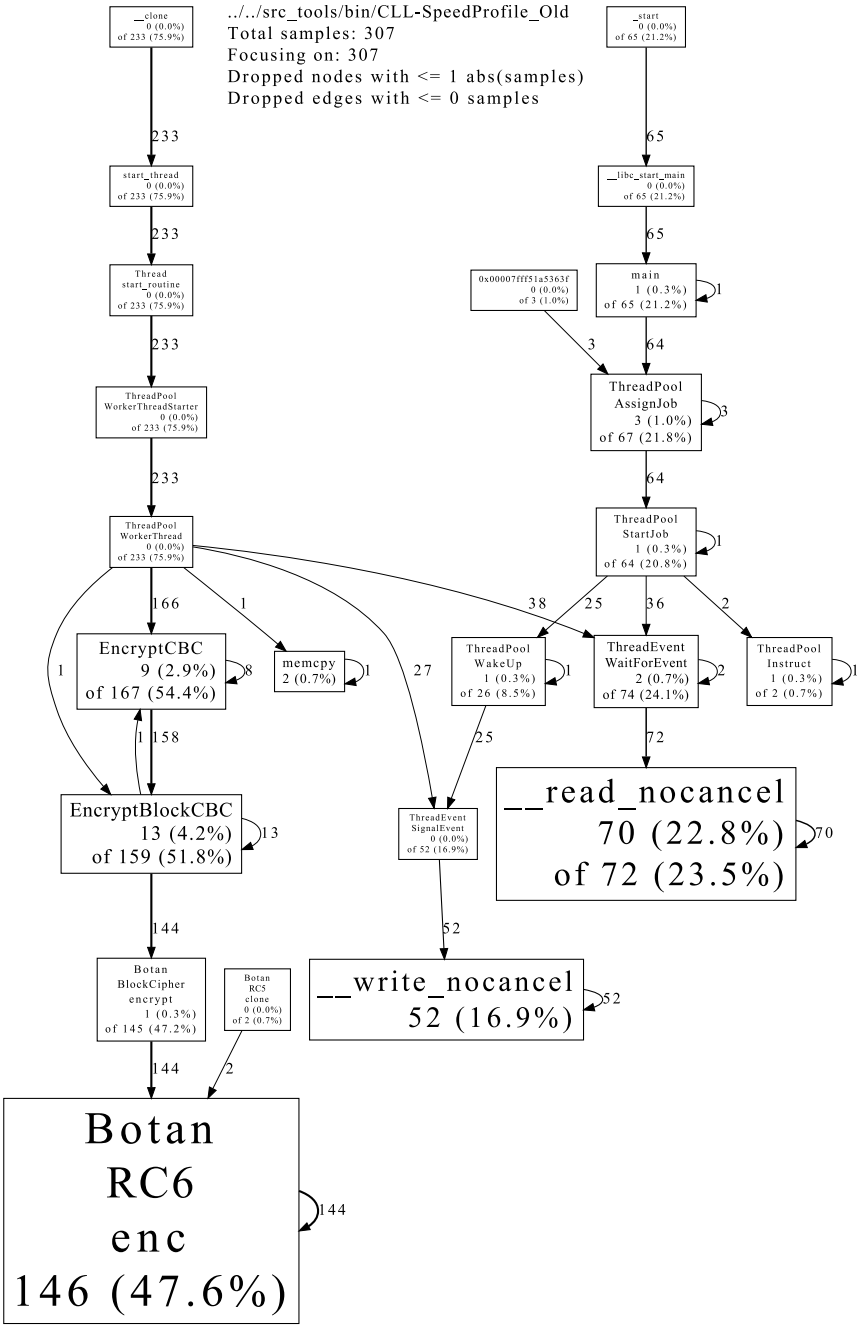

Abbildung A.5.: Laufzeitanalyse des unmodifizierten Threadpool mit einem Thread unter Linux

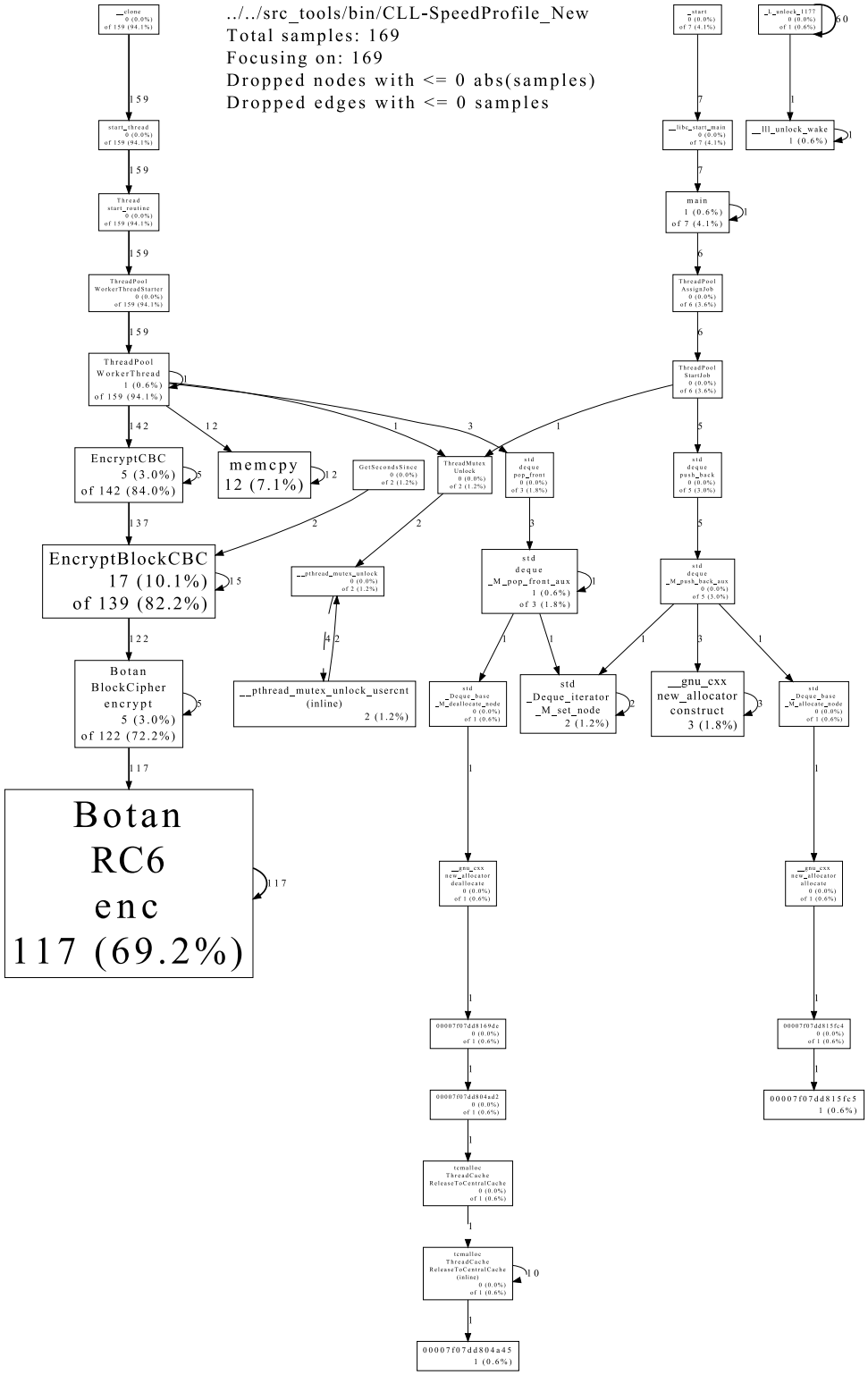

Abbildung A.6.: Laufzeitanalyse des modifizierten Threadpool mit einem Thread unter Linux

Literaturverzeichnis

[1] Intel Corporation: *Intel® Xeon® Processor X3360*. http://ark.intel.com/products/33933/Intel-Xeon-Processor-X3360-%2812M-Cache-2_83-GHz-1333-MHz-FSB%29

[2] BARNEY, Blaise: *POSIX Threads Programming*. Lawrence Livermore National Laboratory

[3] BENZ, Benjamin: Gigabit-LAN auf dem Vormarsch. In: *heise online* (2005), Februar. http://www.heise.de/newsticker/meldung/Gigabit-LAN-auf-dem-Vormarsch-136691.html

[4] DONGARA, Praveen ; VIJAYKUMAR, T. N.: Accelerating Private-Key Cryptography via Multithreading on Symmetric Multiprocessors. In: *Proc. IEEE Int'l Symp. Performance Analysis of Systems and Software (ISPASS 03), IEEE*, 2003, S. 58–69

[5] DROMS, R.: *Dynamic Host Configuration Protocol*. RFC 2131 (Draft Standard). http://www.ietf.org/rfc/rfc2131.txt. Version: März 1997 (Request for Comments). – Updated by RFCs 3396, 4361

[6] FINK, Bill ; SCOTT, Rob: *nuttcp - network performance measurement tool*. ftp://ftp.lcp.nrl.navy.mil/pub/nuttcp/nuttcp.html. Version: 2007

[7] GOOGLE INC. (Hrsg.): *Google-Perftools, Fast, multi-threaded malloc() and nifty performance analysis tools*. Google Inc., http://code.google.com/p/google-perftools/wiki/GooglePerformanceTools

[8] JERSCHOW, Yves I. ; LOCHERT, Christian ; SCHEUERMANN, Björn ; MAUVE, Martin: CLL: A Cryptographic Link Layer for Local Area Networks. In: *SCN 2008:*

Proceedings of the 6th Conference on Security and Cryptography for Networks, 2008, S. 21–38

[9] KRAWCZYK, H. ; BELLARE, M. ; CANETTI, R.: *HMAC: Keyed-Hashing for Message Authentication*. RFC 2104 (Informational). `http://www.ietf.org/rfc/rfc2104.txt`. Version: Februar 1997 (Request for Comments)

[10] LAU, Oliver: Abrakadabra, Programme parallelisieren mit OpenMP. In: *c't extra, Programmieren* (2009), Februar, S. 50–55

[11] LAU, Oliver: Arbeitsteilung, Grundlagen der Thread-Programmierung. In: *c't extra, Programmieren* (2009), Februar, S. 56–58

[12] *Kapitel* 26, Limiting the number of objects of a class. In: MEYERS, Scott: *More Effectve C++, 35 New Ways to Improve Your Programs and Designs*. Addison-Wesley Professional, 1996, S. 130–145

[13] MICROSOFT (Hrsg.): *The Complete Service Sample*. Microsoft, `http://msdn.microsoft.com/en-us/library/windows/desktop/bb540476%28v=vs.85%29.aspx`

[14] MICROSOFT (Hrsg.): *Introduction to Windows Service Applications*. Microsoft, `http://msdn.microsoft.com/en-us/library/d56de412.aspx`

[15] *Kapitel* 2, Designing Threaded Programs. In: NICHOLS, Bradford ; BUTTLAR, Dick ; FARRELL, Jacqueline: *PThreads Programming, A POSIX Standard for Better Multiprocessing*. O'Reilly Media, 1996, S. 31–38

[16] OPENMP ARCHITECTURE REVIEW BOARD (Hrsg.): *OpenMP, Application Program Interface*. OpenMP Architecture Review Board, July 2011

[17] PLUMMER, D.: *Ethernet Address Resolution Protocol: Or Converting Network Protocol Addresses to 48.bit Ethernet Address for Transmission on Ethernet Hardware*. RFC 826 (Standard). `http://www.ietf.org/rfc/rfc826.txt`. Version: November 1982 (Request for Comments). – Updated by RFC 5227

[18] RESCORLA, E.: *Diffie-Hellman Key Agreement Method*. RFC 2631 (Proposed Standard). `http://www.ietf.org/rfc/rfc2631.txt`. Version: Juni 1999 (Request for Comments)

[19] SCHMIDT, Douglas C. ; VINOSKI, Steve: Comparing Alternative Programming Techniques for Multi-threaded Servers: Thread-per-Request. In: *C++ Report* vol. 8 (1996), Februar

[20] *Kapitel* 9, Algorithmenarten und Betriebsmodi. In: SCHNEIER, Bruce: *Angewandte Kryptographie*. Pearson Studium, 2006, S. 223–249

[21] STILLER, Andreas: Hokuspokus, Automatische Parallelisierung mit den Intel-Compilern. In: *c't extra, Programmieren* (2009), Februar, S. 48–49

[22] SUTTER, Herb: The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. In: *Dr. Dobb's Journal* (2005), März. `http://drdobbs.com/architecture-and-design/184405990`

[23] *Kapitel* 2.2, Threads. In: TANENBAUM, Andrew S.: *Moderne Betriebssysteme*. 3. aktualisierte Auflage. Pearson Studium, 2009, S. 137–161

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Düsseldorf, 5. Januar 2012 Moritz Gericke