Entwurf und Implementierung lokaler Interprozesskommunikation für das Fuzzing von Kommunikationsprotokollen am Beispiel von AFLNet und libcoap

Design and Implementation of Local Interprocess Communication for the Fuzzing of Communication Protocols by Example of AFLNet and libcoap

von
Florian Bonetti

Bachelorarbeit im Studiengang Informatik
Universität Bremen

14. Februar 2022

1. Gutachter: Dr. Olaf Bergmann
2. Gutachter: Prof. Dr. Sebastian Maneth

Selbstständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Alle Stellen, die wörtlich oder sinngemäß aus anderen Werken entnommen sind, habe ich unter Angabe der Quellen als solche kenntlich gemacht.

Bremen, den 14. Februar 2022

Abstract

Fuzzer require a fast execution time for efficient use. The motivation of this bachelor thesis is to achieve improved execution time by enhancing interprocess communication in AFLNET and libcoap. The resulting measurement data from the fuzzing process with the adapted interprocess communication in AFLNET and libcoap are evaluated and discussed. This bachelor thesis brings an implementation of *UNIX* sockets in AFLNET and libcoap and indicates indication that the using a different means of interprocess communicationcan give us an advantage, but the execution time of the fuzzing process does not increase significantly.

Zusammenfassung

Fuzzer benötigen für einen effektiven Einsatz eine schnelle Anzahl an Ausführungen pro Sekunde. Die Motivation dieser Arbeit ist es, durch die Anpassung der Interprozesskommunikation an AFLNET und libcoap eine verbesserte Anzahl an Ausführungen pro Sekunde zu erreichen. Die daraus resultierenden Messdaten vom Fuzzing-Prozess mit der angepassten Interprozesskommunikation, wird evaluiert und diskutiert. Diese Bachelorarbeit bringt eine Implementation eines *UNIX-Sockets* in AFLNET und libcoap und weist darauf hin, dass die Verwendung unterschiedlicher Interprozesskommunikation einen Vorteil bringen kann, aber die Anzahl an Ausführungen pro Sekunde des Fuzzing-Prozess, sich nicht signifikant steigert.

Inhaltsverzeichnis

1 Einleitung

Diese Arbeit befasst sich mit der Evaluierung der Anzahl an Ausführungen pro Sekunde von libcoap [1] beim Testen durch AFLNET [2]. Dazu wird die Anzahl an Ausführungen pro Sekunde unter unterschiedlichen *Sockets* [3] geprüft. Beschrieben wird der Prozess vom Entwurf und die Implementierung von *UNIX Domain Sockets* (*UNIX-Sockets*) bis hin zur Evaluation der Anzahl an Ausführungen pro Sekunde gegenüber des *Internet Sockets* (*INET-Sockets*).

1.1 Motivation

Die Vernetzung der Gesellschaft steigt stetig an. Mit dieser Vernetzung steigt auch die Anzahl der Geräte, die zu diesem Netz gehören, an. Besonders die kleinen Geräte, auch IoT genannt, gehören mittlerweile schon zum Alltag. Daher ist es auch wichtig, sichere Software für diese Geräte bereitzustellen. Für die Kommunikation der kleinen Geräte mit dem Heimnetz oder mit dem Internet gibt es die Programmbibliothek libcoap[1]. Um sicherzustellen, dass die zur Verfügung gestellte Software möglichst korrekt funktioniert, bieten sich Tests an, die diese Software prüfen. Eine Testmethodik ist das *Fuzzing* (siehe Abschnitt 3.3), wodurch unterschiedliche Eingaben generiert werden und damit versucht wird, viele Programmpfade zu testen, auch Pfadabdeckung genannt. Weil es wichtig ist, viele verschiedene Eingaben testen zu können, ist es essenziell, dass der Prozess möglichst schnell abläuft.

[1] Webseite: `https://libcoap.net/` (17.05.2021); Github Projekt: `https://github.com/obgm/libcoap` (17.05.2021)

1.2 Ausgangssituation

Für libcoap (siehe Abschnitt 3.4) wurde im Rahmen einer Bachelorarbeit [4] ein Framework entwickelt, welches libcoap automatisiert mittels *Fuzzing* testet. In dem Framework wird AFLNET [2][2] für das Fuzzing verwendet. Bei der Evaluation wurde festgestellt, dass die Anzahl an Ausführungen pro Sekunde der Tests langsam ist. Da AFLNET mit *INET-Sockets* (siehe Abschnitt 3.1) arbeitet, wird vermutet, dass aus diesem Grund die Anzahl an Ausführungen pro Sekunde langsamer ist. Eine Alternative zum *INET-Socket* soll *UNIX-Domain-Socket* [3] oder *Shared-Memory* sein.

1.3 Zielsetzung

Ziel dieser Bachelorarbeit ist es, eine Anpassung an libcoap zusammen mit AFLNET umzusetzen, sodass der Fuzzing-Prozess über *UNIX-Domain-Socket* oder eventuell auch über *Shared-Memory* realisiert wird. Eine Anpassung an libcoap und AFLNET wird benötigt, da die Kommunikation nur über dem *INET-Socket* verläuft und keine Schnittstelle zum *UNIX-Domain-Socket* oder dem *Shared-Memory* vorhanden ist. Mit dieser Implementierung soll eine Evaluierung stattfinden, die die bisherige Anzahl an Ausführungen pro Sekunde mit der angepassten Version vergleicht.
Wie eine repräsentative Evaluierung von Fuzzing-Tests durchgeführt werden sollte und wo es zu Problemen oder Fehlinterpretationen kommen kann, wird in dem Artikel *Evaluating Fuzz Testing* von *Klees et al.* [5] beschrieben und kann als Orientierung dienen und der Einstiegspunkt für die Evaluation sein.

1.4 Abgrenzung

Diese Bachelorarbeit baut auf das von Marten Geuking[4] erstellte *Framework* auf, weswegen wir nur die dort verwendeten Testfälle nutzen und die zur Verfügung stehenden Protokolle, damit wir mit der Implementierung der Interprozesskommunikation in dieser Bachelorarbeit einen Vergleich herstellen können. Im Rahmen der Evaluierung fokussieren wir uns auf die Geschwindigkeit beim Fuzzing-Prozess und behandeln nicht die Ergebnisse, die beim Fuzzing-Prozess entstehen.

[2] Github Projekt: `https://github.com/aflnet/aflnet` (17.05.2021)
[3] Manual page: `https://man7.org/linux/man-pages/man7/unix.7.html` (letzter Aufruf 07.12.21)

2 Verwandte Arbeiten

Auch wenn wir auf die Bachelorarbeit von Marten Geuking[4] aufsetzen und die Einstellungen zusammen mit den vorhandenen Testfällen aus der Bachelorarbeit übernehmen, liegt unser Fokus in der Erweiterung von libcoap und AFLNET mit dem *UNIX-Socket* oder mit dem *Shared-Memory* und der Evaluation zusammen mit der daraus resultierenden Performanz. In dem Artikel *MultiFuzz: A Coverage-Based Multiparty-Protocol Fuzzer for IoT Publish/Subscribe Protocols* von *Yingpei et al.* [6] sehen wir, dass mittels AFLNET auch libcoap auf die Performanz prüft und aus diesem Artikel stammt auch unsere Fragestellung, ob die Performanz durch *UNIX-Sockets* verbessert werden kann. Deswegen unterscheidet sich diese Bachelorarbeit darin, dass wir die Performanz mit dem implementierten *UNIX-Socket* oder *Shared-Memory* prüfen. Zudem schaffen wir einen Vergleich der Performanz zwischen dem *UNIX-Socket* oder *Shared-Memory* gegenüber dem *INET-Socket*.

3 Grundlagen

In diesem Kapitel gibt es eine Übersicht über die Grundlagen, die für diese Arbeit verwendet werden oder worauf aufgebaut wird.

3.1 Kommunikation zwischen den *Sockets*

Sockets [3] ermöglichen eine Kommunikation zwischen zwei Endpunkten innerhalb eines Systems oder auch über die Systemgrenze hinweg. *Sockets* besitzen Dateinamen und können über den *File-Deskriptor* angesprochen werden, um Daten zu schreiben oder zu lesen. Um die Endpunkte über die Systemgrenze verwenden, können *INET-Sockets* verwendet werden, aber es kann auch innerhalb des Systems verwendet werden, wobei die *UNIX-Sockets* nur systemintern verwendet werden können. Je nachdem, welche *Sockets* genutzt werden, kann das über verschiedene Kommunikationstypen ablaufen. Die bekanntesten sind *SOCK_STREAM* und *SOCK_DGRAM*, wobei *SOCK_STREAM* für eine verbindungsorientierte, bidirektionale, zuverlässige und sequenzieller Kommunikation steht und *SOCK_DGRAM* für eine verbindungslose und unzuverlässigen Kommunikation mit fester Nachrichtenlänge. In *UNIX-Sockets* kann auch *SOCK_SEQPACKET* verwendet werden, das verbindungsorientiert ist und dabei mit sequenzierten Paketen arbeitet. *SOCK_SEQPACKET* wird bei der Implementierung jedoch nicht verwendet, da die Anpassung von *INET-Sockets* auf *UNIX-Sockets* mit *SOCK_STREAM* und *SOCK_DGRAM* einfacher ist, da die Infrastruktur für *SOCK_STREAM* und *SOCK_DGRAM* zum größten Teil vorhanden ist.

3.2 Shared Memory

Shared Memory [7] ist ein Speicherbereich, der von verschiedenen Prozessen verwendet werden kann, um so eine Kommunikation zwischen den Prozessen zu gewährleisten. *Shared Memory* unterscheidet sich zu *UNIX-Sockets* darin, dass es schneller ist, da z. B. keine Transportprotokolle benötigt werden oder der gemeinsame Speicher eher im *Cache* gehalten wird. Allgemein ist *Shared Memory* primitiver und bietet, im Gegensatz zu *UNIX-Sockets* nicht das Blocken von Lese- und Schreibkonflikten an.

3.3 Fuzzing und Fuzz-Testing

Das *Fuzzing* ist das Erstellen von zufällig generierten Eingaben durch einen *Fuzzer*. Dazu werden gültige Eingaben durch verschiedene Methoden verändert und in ein laufendes Programm eingegeben, auch *Program Under Test* (PUT) genannt. Damit soll ein unerwartetes Verhalten beim PUT erzeugt werden. Das *Fuzz-Testing* ist in dem Fall dann durch *Fuzzing*, ein PUT, auf Fehlverhalten zu testen. In der Bachelorarbeit von Marten Geuking [4] und in dem Fachartikel *The Art, Science, and Engineering of Fuzzing: A Survey* von *Manès et al.* [8] wird genauer auf das Thema *Fuzzing* eingegangen.

3.4 libcoap

libcoap [1] ist eine, in der Programmiersprache *C* geschriebene Implementierung des *Constraint Application Protocol's* (CoAP) [9] und bietet ein Web-Übertragungsprotokoll für eingeschränkte Geräte und eingeschränkte Netze an.

3.5 AFLNET

AFLNET [2] ist ein *Grey-Box-Fuzzer* (siehe Abschnitt 3.3) für Protokollimplementierungen. AFLNET basiert auf dem *American Fuzzy Lop (AFL)* [1]. Es agiert als *Client* und prüft durch einen Nachrichtenaustausch mit einem *Server* die Codeabdeckung und die Status, in dem sich das PUT befindet, welches aus den Antwortcodes des PUT ermittelt werden.

[1] American Fuzzy Lop: `https://github.com/google/AFL`

In der Bachelorarbeit von Marten Geuking [4] wird auf das Thema AFLNET genauer eingegangen.

3.6 Relevante Transportprotokolle in libcoap

Da in dieser Arbeit mit libcoap gearbeitet wird und auch die Übertragung zwischen den Endpunkten (*Sockets*) von libcoap und AFLNET angepasst wird, haben Transportprotokolle hier einen relevanten Anteil. Aus diesem Grund werden diese kurz beschrieben.

3.6.1 Transmission Control Protocol

TCP [10] ist ein Netzwerkprotokoll, das eine verbindungsorientierte Kommunikation zwischen Endpunkten ermöglicht. Es müssen also nur zum Beginn zusätzliche Daten mitgesendet werden wie der *Port* (englische Bezeichnung für eine Schnittstelle zu einer Anwendung), um Quelle und Ziel zu beschreiben. Zum Verbindungsaufbau wird ein *Handshake* ausgeführt. Ein *Handshake* wird benötigt, damit beide Kommunikationspartner die Verbindung korrekt aufbauen können. Dazu wird mit dem *SYN* ein Verbindungsaufbauwunsch gestellt und mit dem *SYN-ACK* vom anderen Kommunikationspartner bestätigt. Sendet nun der Kommunikationspartner mit dem Aufbauwunsch eine *ACK* auf das empfangene *SYN-ACK*, wissen beide Kommunikationspartner, dass nun eine Verbindung besteht.

3.6.2 User Datagram Protocol

UDP [11] ist ein Netzwerkprotokoll, das eine verbindungslose Kommunikation zwischen Endpunkten ermöglicht. Da keine direkte Verbindung besteht, müssen hier in jedem Paket zusätzliche Daten mitgesendet werden (*Port*), mit Ziel und Quelle aber kommt daher ohne einen Handshake aus. Bei verbindungsloser Kommunikation kann es vorkommen, dass die Pakete nicht in sequenzieller Reihenfolge beim Ziel ankommen oder auch verloren gehen.

3.6.3 Transport Layer Security

TLS [12] ist ein Protokoll, das auf TCP (siehe Abschnitt 3.6.1) aufsetzt, mit dem Zusatz, dass ein sicherer Kanal über das Internet oder *Local Area Network* (LAN) zur Verfügung

gestellt wird. Das wird damit erreicht, dass die übertragenen Daten verschlüsselt werden und somit für Dritte unkenntlich sind. Mit TLS werden die Ziele Vertraulichkeit, Integrität und Authentifizierung bereitgestellt.

- Vertraulichkeit: Die Daten, die zwischen zwei Endpunkten versendet werden, sind auch nur für diese erkennbar.
- Integrität: Versendete Daten können von Angreifern nicht unbemerkt verändert werden.
- Authentifizierung: Der Kanal auf der Server-Seite ist immer authentisiert. Optional kann zusätzlich auch der Kanal der Client-Seite authentisiert sein.

3.6.4 Datagram Transport Layer Security

DTLS [13] ist ein Protokoll, das auf TLS (siehe Abschnitt 3.6.3) basiert und die selbe Sicherheitsgarantie beibehält. Der wesentliche Unterschied zu TLS ist, dass die Datagrammsemantik, analog zu UDP, in DTLS beibehalten wird.

3.6.5 Serial Line Internet Protocol

SLIP [14] ist ein einfaches Protokoll, dass die Internet-Protocol-Pakete (IP-Pakete) für eine serielle Punkt-zu-Punkt-Verbindung kapselt. Das Protokoll bietet keine Adressierung, Fehlererkennung/-korrektur oder Kompression.

3.7 Single-User-Mode und Multi-User-Mode

Linux-Systeme können beim Starten in verschiedene *Runlevel* (englisch für Ausführungsebene) [2] starten. Darüber kann bestimmt werden, welche Services beim Start mit gestartet werden. Die *Runlevel* unterscheiden sich in sieben Stufen, wobei nicht alle für uns interessant sind. Beim *Single-User-Mode* befinden wir uns in dem ersten *Runlevel*, in dem nur wenige Services gestartet werden. Es werden z. B. keine Services für die graphische Benutzeroberfläche gestartet und auch kein Netzwerk-Manager. Beim *Multi-User-Mode*

[2] Manual page: `https://www.man7.org/linux/man-pages/man8/runlevel.8.html` (letzter Aufruf 07.12.21)

befinden wir uns beim *Runlevel* zwischen zwei und fünf, wobei bei jeder höheren Stufe auch mehr Services gestartet werden. Bei den meisten Linux-Systeme mit einer graphische Benutzeroberfläche befinden wir uns im *Multi-User-Mode*.

4 Entwurf

In dem Abschnitt beschreiben wir den Entwurf eines Systems, das libcoap und AFLNET so erweitern, dass die Fuzzing-Tests von AFLNET neben den Internet-*Sockets* (*INET-Sockets*) auch mit *UNIX-Domain-Sockets* (*UNIX-Sockets*) funktionieren.
Dafür müssen Änderungen an den Dateien von libcoap und AFLNET unternommen werden, die für die Kommunikation zwischen den *Sockets* verantwortlich sind. Wir benötigen für AFLNET und libcoap die Option, neue Endpunkte in Form von *UNIX-Sockets*, zur Verfügung zu stellen. Der Grund dafür ist, dass eine Auswahl vor der Ausführung getroffen werden kann, mit welchen Endpunkten gearbeitet werden soll. Zudem benötigen wir für UDP und TCP in libcoap die Option, *UNIX-Sockets* zu erstellen, die für das Senden sowie Empfangen von Daten in libcoap verantwortlich sind. Analog zu libcoap benötigen wir auch für AFLNET die Option, *UNIX-Sockets* vor der Ausführung auszuwählen.
Beim Übertragen kann die Schicht von TCP oder UDP noch in eine Schicht von SLIP (siehe Abschnitt 3.6.5) eingepackt werden, um zusammenhängende Pakete, die übertragen werden, zu identifizieren. Wenn die Übertragung durch *Shared Memory* realisiert werden soll, dann wird eine Anpassung von SLIP benötigt, da *Sockets* Lese- und Schreibkonflikte eigenständig auflösen, was bei *Shared Memory* nicht gegeben ist.
Zum Evaluieren der Anzahl an Ausführungen pro Sekunde können die Daten, die von AFLNET erzeugt werden, verwendet werden. Die Messdaten können wir noch mit eigenen benötigten Messungen ergänzen.

4.1 Anforderungen

Im folgendem Abschnitt werden die Anforderungen definiert, die zur Erreichung der Ziele benötigt werden.

1. **Endpunkte**: Die verwendeten Endpunkte sind wie in Abschnitt 1.3 fest definiert. *Shared Memory* ist dabei optional. Die Übertragung wird durch SLIP unterstützt.

2. **Messungen**: Die Messungen müssen vergleichbare Daten hervorbringen, um Unterschiede zwischen *UNIX-Sockets* (siehe Abschnitt 3.1) und *INET-Sockets* zu identifizieren. Dazu messen wir die Anzahl an Ausführungen von AFLNET beim Fuzzing-Prozess von libcoap in Sekunden, um eine Kennzahl zu bekommen, die uns etwas über die Geschwindigkeit aussagt.

3. **Integration**: Die umgesetzten Änderungen an libcoap sollen auch in Zukunft in dem Projekt von libcoap übernommen werden. Deswegen wird darauf geachtet, eine möglichst saubere Integration der zusätzlichen Endpunkte vorzunehmen und die Möglichkeit, diese zusätzlichen Endpunkte einfach zu konfigurieren.

4. **Performanz**: Im *AFL User Guide* [15] wird davon gesprochen, dass der Fuzzing-Prozess idealerweise eine Anzahl an Ausführungen pro Sekunde von über 500 haben soll. Im Quellcode von AFLNET in der Datei (afl-fuzz.c) wird die Anzahl an Ausführungen pro Sekunde als langsam deklariert, wenn der Wert unter 100 liegt. Beim Auswerten der erhobenen Daten werden wir einen Vergleich zwischen unseren gemessenen Ergebnissen mit den Werten aus dem *AFL User Guide* und dem Quellcode herstellen. Zudem stellen wir einen Vergleich der Performanz zwischen dem *UNIX-Socket* und dem *INET-Socket* her.

5 Implementation

In diesem Abschnitt befassen wir uns mit den wesentlichen Änderungen, die für die Integration der Endpunkte auf der Seite von libcoap und AFLNET gemacht wurden.

5.1 Anpassungen an libcoap

Die Programmierbibliothek libcoap nutzen wir mit der Version 4.3.0 und ist in der Programmiersprache *C* implementiert. Aus diesem Grund werden die Anpassungen ebenfalls in *C* durchgeführt.
Die ersten Änderungen werden an der Datei coap-server.c durchgeführt, in den auch die Änderungen von Marten Geuking [4] übernommen werden. Ursprünglich wurde im Server die IPv4- und IPv6-Adressen, sowie die Ermittlung der Familie der *Sockets* über die Funktion getaddrinfo() gelöst, um so die Endpunkte zu erstellen. Das wird für *UNIX-Sockets* von getaddrinfo() nicht unterstützt und musste daher gesondert angelegt werden.
Die Erstellung der *Sockets* und das Senden sowie Empfangen von Daten findet in der Datei coap_io.c für UDP und in der Datei coap_tcp.c für TCP statt. Also muss in beiden Dateien die Option für das Binden und das Verbinden zur Verfügung gestellt werden. Das Problem, das die *UNIX-Sockets* mit sich bringen, ist, dass die IPv4- und IPv6-Adresse sowie der *Port* beim Empfangen nicht mitgesendet wurden, sondern nur der Name des *Sockets*. Die Daten werden aber von libcoap erwartet, um die *Session* (englisch für Sitzung) zuzuordnen. Aus dem Grund werden die erforderlichen Daten vor den eigentlichen Daten mitgesendet. Das funktioniert, solange die Pakete nicht aufgeteilt werden und die zusätzlichen Daten nicht mehr eindeutig von den richtigen Daten unterschieden werden können. Um das zu verhindern, werden zum Versenden und Empfangen die gesendeten Datenpakete mit dem SLIP-Protokoll eingepackt, damit die Datenpakete identifiziert werden können. Auch für eine Erweiterung mit *Shared Memory* bräuchte es ein ähnliches Protokoll in einer *Mutex*-Variante, da auch hier die zusätzlichen Daten zur Zuordnung der

Session benötigt werden und darüber hinaus müssen Lese- und Schreibkonflikte behandelt werden, wofür die *Mutex*-Variante sorgen soll.

5.2 Anpassungen an AFLNET

Bei AFLNET nutzen wir die Version 2.56b und auch AFLNET wurde in der Programmiersprache *C* implementiert. Die Anpassungen werden ebenfalls in *C* implementiert. Eine Änderung, die gemacht wurde ist, wenn das Programm afl-fuzz.c gestartet werden soll, muss über die mitgegebenen Optionen dem Programm mitgeteilt werden, zu welchem *Sockets* die Verbindung aufgebaut werden soll. Diese Information wird beim Erstellen und Verbinden der *UNIX-Sockets* weitergegeben, welches ebenfalls in afl-fuzz.c passiert. So wie im Abschnitt 5.1 beschrieben, muss hier das SLIP-Protokoll implementiert werden und auch die zusätzlichen Daten mitgesendet werden, wie IPv4- oder IPv6-Adressen und *Port*, die von libcoap gefordert werden. Da momentan die zusätzlichen Daten in libcoap nur von UDP unterstützt werden, werden diese hier auch nur bei UDP mitgesendet. Vermutet wird ein Problem,dass durch die zusätzlichen Daten sowie durch das SLIP-Protokoll entstand. Die empfangenen Daten bei libcoap und auch der *Handshake* (siehe Abschnitt 3.6.3) werden durch dieselbe Funktion empfangen. In AFLNET werden die gesendeten Daten, die für die Tests verwendet werden und der *Handshake*, der nur für den Aufbau der Verbindung zuständig ist, separat an den Server von libcoap übertragen. Das bedeutet, dass auf Seiten von AFLNET nur die gesendeten Daten für den Testbetrieb in das SLIP-Protokoll eingepackt werden aber der *Handshake* wird nicht vom SLIP-Protokoll eingepackt. Deswegen würde libcoap die zusätzlichen Daten und das SLIP-Protokoll beim Empfangen nicht vorfinden und es entsteht eine Inkonsistenz, weil libcoap auch beim *Handshake* die zusätzlichen Daten und das SLIP-Protokoll erwartet. Aus dem Grund müssen die zusätzlichen Daten und das SLIP-Protokoll kurz vor dem Versenden verpackt werden und nicht beim Erstellen der Fuzz-Daten (siehe Abschnitt 3.3) selbst. Eine andere Vermutung ist, dass libcoap teilweise beim Endpunkt mit TCP eine kleine Datenmenge von vier Bytes erwartet. Bei den erwarteten vier Bytes könnte es sich um die *Capabilities and Settings Messages* (CSMs) handeln. CSMs sind Optionen, die zu Beginn eines Verbindungsaufbaus von allen Endpunkten gesendet werden müssen. Wenn wir das SLIP-Protokoll mit den zusätzlichen Daten übermitteln, übersteigen wir diese vier Bytes.

6 Evaluation

Als Orientierung zur Evaluation dient die wissenschaftliche Arbeit *Evaluating Fuzz Testing* von *Klees et al.* [5], in der beschrieben wird, wie die Performanz von *Fuzz-Tests* gemessen werden kann. Die Herausforderung beim Evaluieren von *Fuzz-Tests* besteht darin, dass zufällig generierte Daten versendet werden. Wenn die Datenlänge beim Versenden also variiert, kann keine genaue Aussage gemacht werden, wie schnell das Versenden, das Verarbeiten auf der Serverseite ist. Für das Problem wird in *Evaluating Fuzz Testing* von *Klees et al.* [5] empfohlen, dass mehrere Durchgänge durchgeführt werden sollen, um statistische Abweichungen zu eliminieren.

6.1 Erhebung der Messdaten

Wir messen die durchschnittliche Anzahl an Ausführungen pro Sekunde vom Programm AFLNET und wie lange die Bearbeitungszeit einer Anfrage vom *Fuzzer* bis zur Beantwortung durch den Server von libcoap dauert. Zu den durchschnittlichen Ausführungen gehören nicht nur die erfolgreichen Anfragen, sondern auch die, die zu einem Fehler oder zum Absturz des Servers führen würden. Grafik 6.1 zeigt das Vorgehen bei der Erhebung von Messdaten. Wir haben in dem Sequenzdiagramm 6.1 auf der linken Seite den AFLNET-Prozess, der den libcoap-Prozess auf der rechten Seite in einem Kindprozess startet. Die allgemeine Zeitmessung startet vor der Ausführung von libcoap. Die Zeitmessung wird verwendet, um die Laufzeit von AFLNET zu bestimmen, aber auch, um die durchschnittliche Anzahl an Ausführungen pro Sekunde von AFLNET zu berechnen. Das wird zusammen mit der Ausführungsvariable berechnet. Der Wert der Ausführungsvariable wird nach jedem Durchlauf erhöht. Dazu werden die gezählten Ausführungen durch die gelaufene Zeit in Sekunden geteilt.

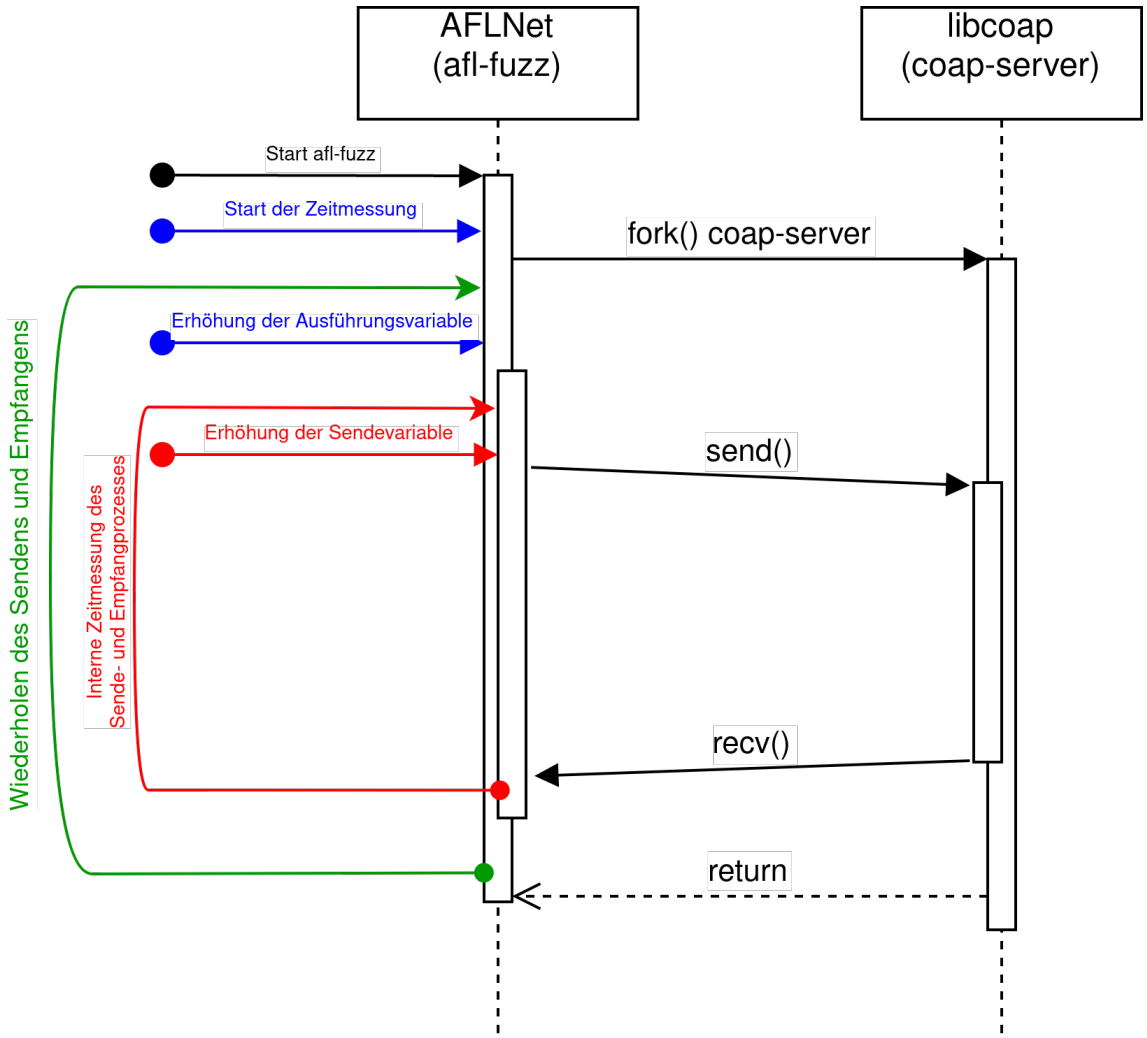

Abbildung 6.1: Sequenzdiagramm für die Erhebung der Messdaten.

Um zusätzlich einen präziseren Vergleich zwischen *UNIX-Sockets* und *INET-Sockets* zu bekommen, möchten wir noch wissen, wie schnell der Übermittlungsprozess zusammen mit der Bearbeitungszeit des libcoap-Prozesses ist. Dazu messen wir die durchschnittliche Anzahl der Übermittlungen pro Sekunde vom Sende-, Bearbeitungs- und Empfangsprozess. Der Wert der Sendevariable wird bei jeder gesendeten Nachricht erhöht, und die Zeit wird vor dem Senden bis nach dem Empfangen gemessen und aufaddiert. Die durchschnittliche Anzahl der Übermittlungen pro Sekunde wird analog zur durchschnittlichen Anzahl an Ausführungen pro Sekunde berechnet, indem wir die Anzahl der gesendeten Nachrichten, repräsentiert durch die Sendevariable, durch die aufaddierte Zeit in Sekunden teilen. Es muss jedoch berücksichtigt werden, dass zwischen dem Senden und Empfangen Speicher

bereitgestellt wird. Das würde die Geschwindigkeit verzögern, falls die Übermittlung und die Bearbeitungszeit durch den libcoap-Prozess schneller sind.
AFLNET zeigt uns während des Fuzzing-Prozesses die Anzahl an Ausführungen pro Sekunde, und auch, ob der angezeigte Wert langsam ist. Aus dem Grund fragen wir uns, wie schnell der Übertragungsprozess und die Bearbeitungszeit vom libcoap-Prozess sein müssen, damit die durchschnittliche Anzahl an Ausführungen pro Sekunde nicht mehr von AFLNET als langsam deklariert wird oder wie schnell die durchschnittliche Anzahl an Ausführungen pro Sekunde wäre, wenn der Übertragungsprozess und die Bearbeitungszeit aus der Berechnung entfernt werden. Dazu nehmen wir die Zeit, die beim Übertragungsprozess gemessen wird, und subtrahieren diese von der aktuell gemessenen Zeit, und berechnen dann aus der Differenz wieder die durchschnittliche Anzahl an Ausführungen pro Sekunde.
In AFLNET besteht das Problem, dass wir als durchschnittliche Anzahl an Ausführungen pro Sekunde nicht den korrekten Wert bekommen. Im Quellcodeausschnitt 6.1 sehen wir, dass bei einem starken Anstieg oder Abfall der durchschnittlichen Anzahl an Ausführungen pro Sekunde eine Glättung stattfindet. Das sehen wir im Quellcodeausschnitt 6.1 in Zeile 13 und 14. Da werden die durchschnittliche Anzahl an Ausführungen pro Sekunde (avg_exec) angepasst, falls der aktuelle Durchschnitt (cur_avg) einen fünffachen Anstieg oder Abfall hat. Zudem wird in Zeile 16 und 17 für eine weichere Änderung der durchschnittlichen Anzahl an Ausführungen pro Sekunde (avg_exec) gesorgt. Da in afl-fuzz.c mehrere Statistiken zum getesteten Server anzeigt werden, kann es sinnvoll sein, dass der Anstieg oder Abfall der durchschnittlichen Anzahl an Ausführungen pro Sekunde (avg_exec) weichere Übergänge hat. Bei einer Evaluierung würde dieser Vorgang jedoch die realen Daten verfälschen. Aus dem Grund beziehen wir auch zusätzlich noch die durchschnittliche Anzahl an Ausführungen pro Sekunde ohne das Glätten ein und weisen den Wert in (cur_avg) direkt der Variablen (avg_exec) zu, auch ohne eine Prüfung eines Anstiegs oder Abfalls, so wie wir es in Zeile 13 sehen.

```
1     if (!last_execs) {
2
3         avg_exec = ((double) total_execs) * 1000 / (cur_ms - start_time);
4
5     } else {
6
7         double cur_avg = ((double) (total_execs - last_execs)) * 1000 /
8                          (cur_ms - last_ms);
9
10        /* If there is a dramatic (5x+) jump in speed, reset the indicator
11       more quickly. */
12
13        if (cur_avg * 5 < avg_exec || cur_avg / 5 > avg_exec)
14            avg_exec = cur_avg;
15
16        avg_exec = avg_exec * (1.0 - 1.0 / AVG_SMOOTHING) +
17                   cur_avg * (1.0 / AVG_SMOOTHING);
18
19    }
20
21    last_ms = cur_ms;
22    last_execs = total_execs;
```

Listing 6.1: Berechnung der durchschnittlichen Anzahl an Ausführungen pro Sekunde in der Datei afl-fuzz.c. [2]:AFLNET *Quellcode*

Da wir auf die Bachelorarbeit von Marten Geuking [4] aufbauen, verwenden wir zum Vergleich auch seinen angepassten libcoap-Server. Da dieser libcoap-Server für den *INET-Socket* nur die Endpunkte für TCP und UDP unterstützt, können wir unsere Endpunkte TLS, DTLS, TCP und UDP nur im Verhältnis zu den Endpunkten TCP und UDP setzen.

6.2 Gesammelte Daten

Wie wir eingangs im diesem Kapitel besprochen haben, benötigen wir mehrere Durchgänge, um repräsentative Daten zu bekommen. Wir haben die erhobenen Daten, so wie in der folgenden Tabelle 6.2 dargestellt, erfasst. Dazu haben wir uns dafür entschieden, einen Durchlauf mit einer Stunde zu bemessen. Die Wiederholungen der Durchläufe haben wir auf elf festgesetzt. Es war uns wichtig, eine repräsentative Anzahl an erhobenen Daten zu

bekommen und der Zeitaufwand für die Erhebung der Daten soll in einen angemessenen Rahmen stattfinden. Für die Erfassung der Daten wurde folgendes System genutzt:

- **Betriebssystem:**
  Kubuntu 20.04 64-bit (Kernel: 5.4.0-91-generic)
- **Hardware:**
  Prozessor: *AMD Ryzen 9 3900X*
  Arbeitsspeicher: *15,6 GiB Arbeitsspeicher*
- **Software:**
  libcoap *Version 4.3.0*
  AFLNET *Version 2.56b*

Socket	INET		UNIX			
Protokoll	TCP	UDP	TCP	UDP	TLS	DTLS
Durchläufe	11	11	11	11	11	11
Dauer eines Durchlaufes in Stunden	1	1	1	1	1	1
Dauer eines Durchlaufes pro Socket in Stunden	22		44			
Dauer insgesamt in Stunden	66					

Abbildung 6.2: Übersicht des Zeitraums, in dem die erhobenen Daten erfasst werden.

AFLNET hat vorgesehen, dass alle fünf Sekunden die Daten zur Auswertung erfasst werden und bei besonderen Anlässen, wie z. B., wenn ein Fehler oder ein neuer Pfad im PUT gefunden wurde. Bei einem Durchlauf wären das dann mindestens 720 Einträge, wenn wir eine Stunde pro Durchlauf berechnen. Jedoch wurde dieser Wert bei keinem Protokoll erreicht, und die Anzahl der Einträge schwankt stark. Eine Vermutung, die wir auch überprüfen können, ist, dass die Schwankungen und die niedrige Anzahl an Einträgen dadurch entstehen, dass bei den Durchläufen die Anzahl der Prozesse des Systems, auf dem die Durchläufe durchgeführt werden, stören können. Aus dem Grund haben wir die Durchläufe für DTLS und TLS beim *UNIX-Socket* im *Single-User-Mode* durchgeführt, um die Anzahl an potenziell störenden Prozessen zu reduzieren. Im *Single-User-Mode* waren insgesamt fünf Prozesse aktiv, wohingegen nach einem normalen Start im *Multi-User-Mode* 254 Prozesse aktiv waren. Der Vergleich zwischen dem *Single-User-Mode* und dem *Multi-User-Mode* haben wir mit acht verschiedenen Durchläufen in Grafik 6.3 dargestellt.

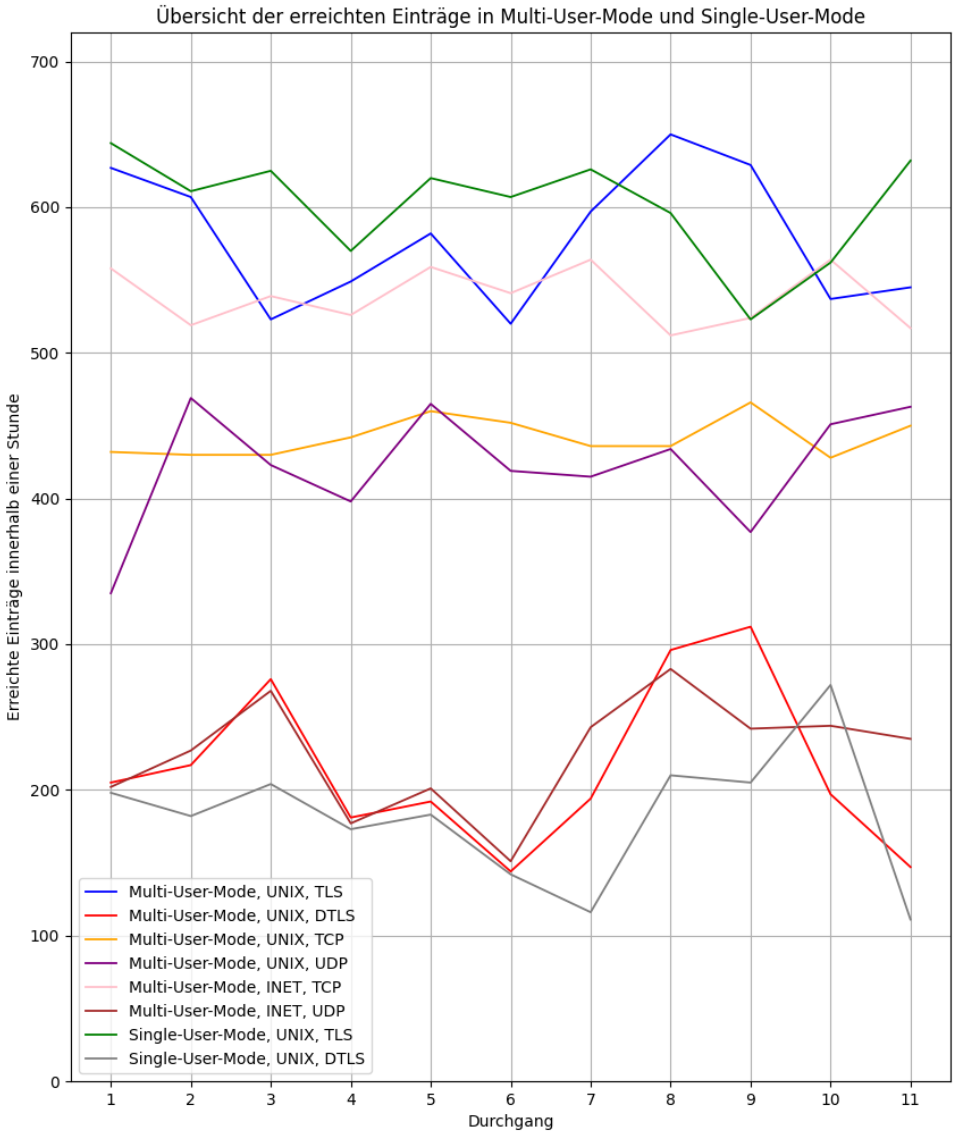

Abbildung 6.3: Übersicht der erreichten Einträge in Multi-User-Mode und Single-User-Mode .

Was wir zum einen auf der Grafik erkennen, ist, dass die Unterschiede zwischen dem *Single-User-Mode* und *Multi-User-Mode* nicht signifikant sind. Auffällig ist jedoch, dass wir drei Gruppen erkennen. Die Gruppen lassen vermuten, dass die Anzahl an Einträgen

innerhalb einer Stunde davon abhängen, ob der Durchgang mit den Protokollen UDP und DTLS oder mit TCP und TLS durchgeführt wurde. Eine Ausnahme sind im *Multi-User-Mode* und mit dem *UNIX-Socket* die Protokolle UDP und TCP. Die Ursache für die Unterschiede haben wir bisher noch nicht herausgefunden. Es könnte daran liegen, dass die Anzahl an Ausführungen pro Sekunde bei den Protokollen unterschiedlich ist. Das hat zufolge, dass es öfter zur Abfrage kommt, ob ein Eintrag vorgenommen werden soll, und somit kommt es auch schneller zu neuen Pfaden oder Fehlern, die ebenfalls zu einem Eintrag führen. Wir haben nicht jede mögliche Kombination zum Vergleich herangezogen, weil wir davon ausgehen, dass es nicht zu weiteren Hinweisen führen wird. Zum Beispiel fehlen die Kombinationen mit dem *Single-User-Mode* und dem *INET-Socket*.
In Tabelle 6.2 haben wir den Zeitraum gezeigt, indem wir die erhobenen Daten bezogen haben und in den folgenden Datensätzen 6.4 zeigen wir die Anzahl an Einträgen, die wir zum Auswerten heranziehen.

Socket	**INET**		**UNIX**			
Protokoll	**TCP**	**UDP**	**TCP**	**UDP**	**TLS**	**DTLS**
Erhobene Daten pro Protokoll und Socket	5923	2473	4862	4649	6366	2361
Erhobene Daten pro Socket	8396		18238			
Erhobene Daten insgesamt	26634					

Abbildung 6.4: Übersicht der Anzahl der erhobenen Daten.

Um die erhobenen Datensätze 6.4 grafisch darzustellen, benötigen wir jeweils für einen Vergleich eine gleiche Anzahl an Einträgen. Aus dem Grund nehmen wir für einen Vergleich den Datensatz mit den geringsten Einträgen und beschneiden alle anderen Datensätze auf dieselbe Anzahl an Einträgen. Dazu muss berücksichtigt werden, dass die fehlenden Einträge nicht dargestellt werden. Wenn wir die Beschneidung jedoch nicht durchführen, würden die Datensätze mit den geringsten Einträgen zum Ende hin nicht mehr berücksichtigt werden und wir hätten keinen statistischen Vergleich.

6.3 Auswertung der erhobenen Daten

Für die Auswertung der erhobenen Daten vergleichen wir die Anzahl an Ausführungen pro Sekunde. Wie aus dem Abschnitt 1.3 zur Zielsetzung hervorgeht, wollen wir wissen, ob die Nutzung von *UNIX-Sockets* dafür sorgt, dass die Anzahl an Ausführungen pro Sekunde erhöht wird. Dazu vergleichen wir die *Sockets*, also *UNIX* mit *INET*, einmal mit dem Protokoll TCP und UDP, aber auch mit TLS und DTLS. Grafik 6.5 zeigt die Auswertung

zwischen *UNIX* mit TLS und *INET* mit TCP, Grafik 6.6 zwischen *UNIX* mit DTLS und *INET* mit UDP, Grafik 6.7 zwischen *UNIX* mit TCP und *INET* mit TCP und Grafik 6.8 zwischen *UNIX* mit UDP und *INET* mit UDP. Eine Grafik enthält zwei Abbildungen. Die obere Abbildung zeigt uns die Ausführungen pro Zeit mit der Glättung, wie wir im Abschnitt 6 mit dem Quellcode 6.1 beschrieben haben. Aus dem Grund haben wir in der unteren Abbildung mit denselben erhobenen Daten die Ausführungen ohne Glättung dargestellt, um die unveränderten Ausführungen abbilden zu können. Die einzelnen Punkte in der Grafik sind die Ausführungen, die als Eintrag festgehalten wurden. Die durchgehenden Linien sind der Durchschnitt, der sich aus den elf Ausführungen berechnet. Die leicht transparente Fläche im Hintergrund der Linie ist die Standardabweichung, die aus den Ausführungen berechnet wird.

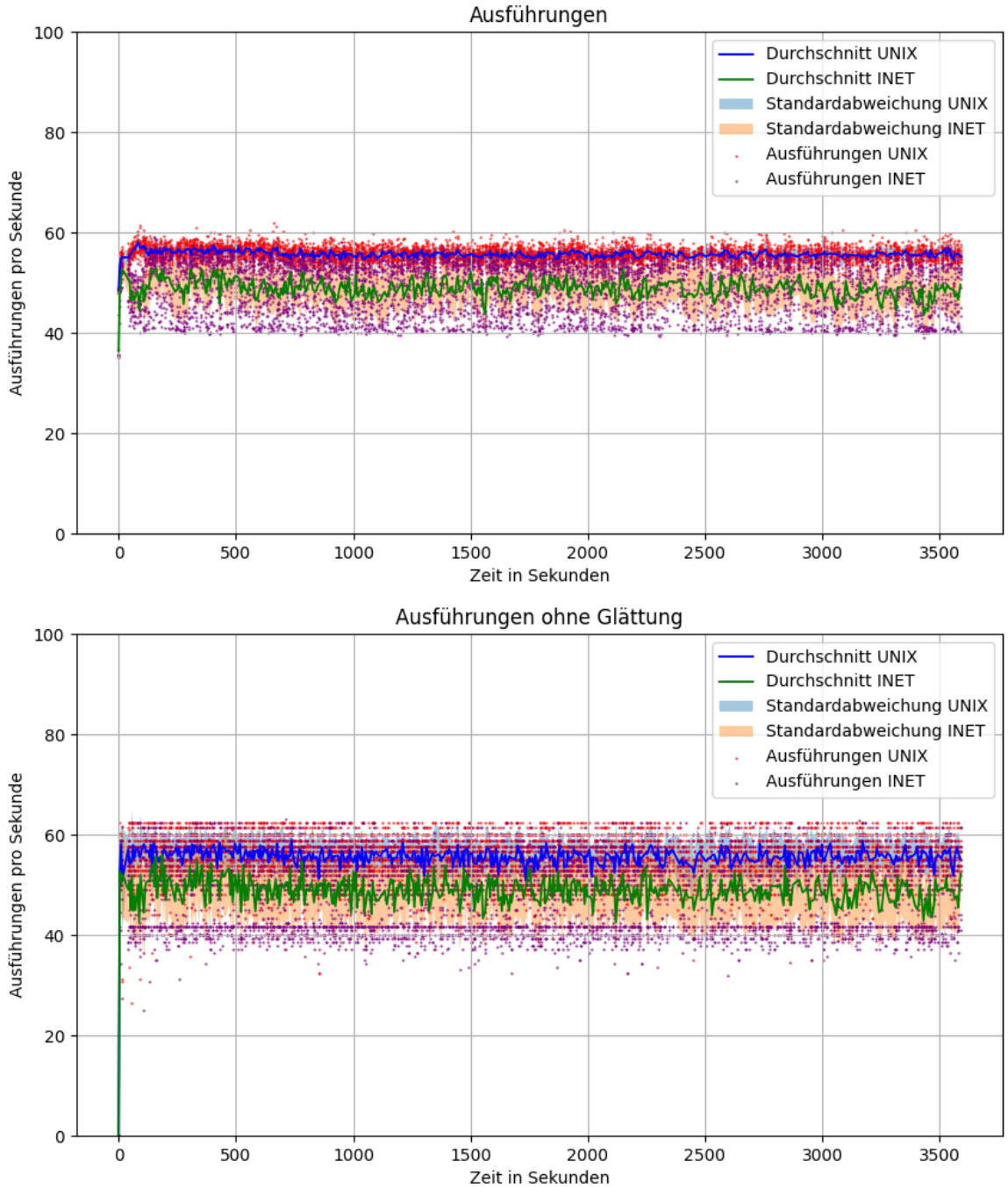

Abbildung 6.5: Anzahl an Ausführungen pro Sekunde zwischen UNIX mit TLS und INET mit TCP.

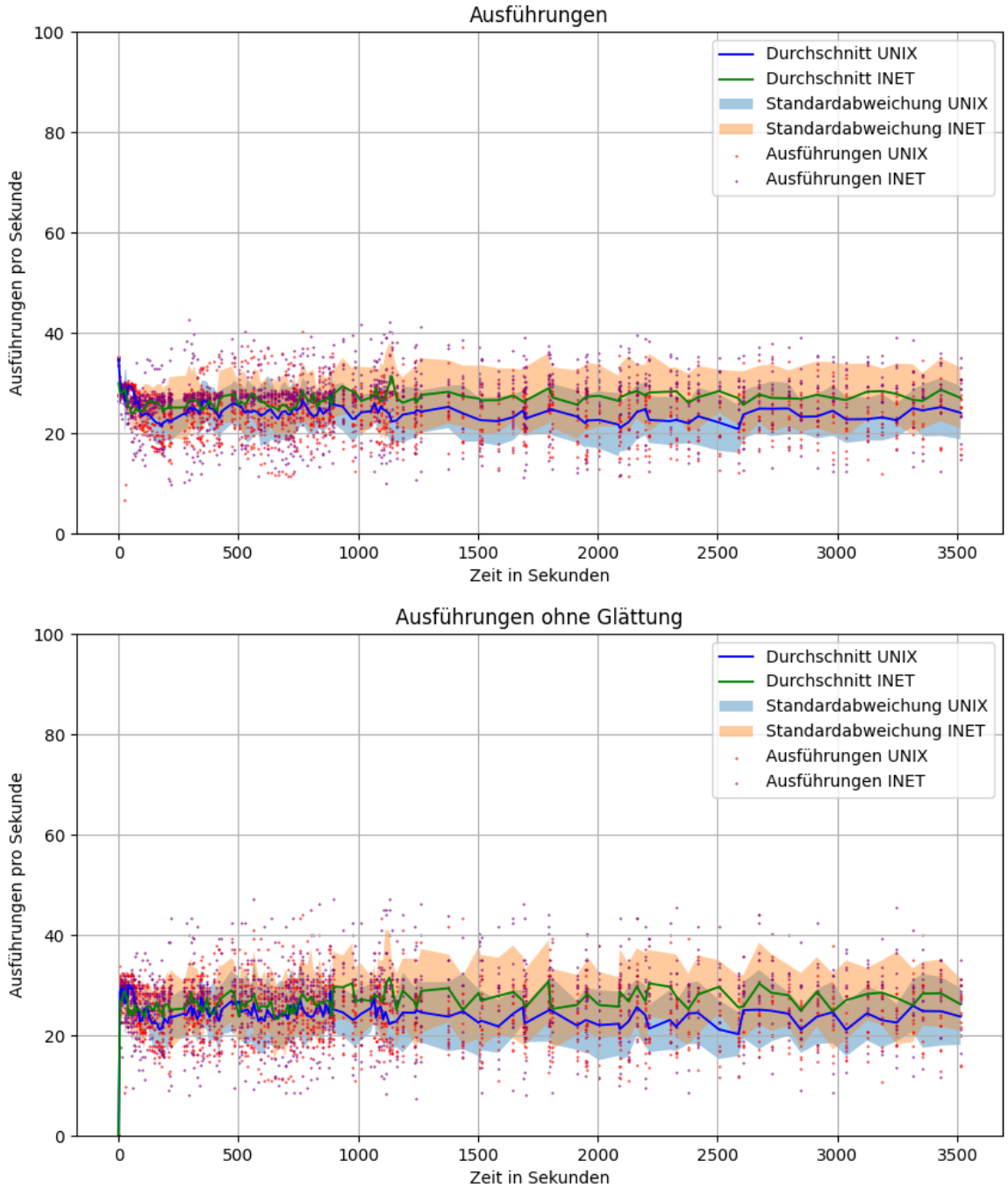

Abbildung 6.6: Anzahl an Ausführungen pro Sekunde zwischen UNIX mit DTLS und INET mit UDP.

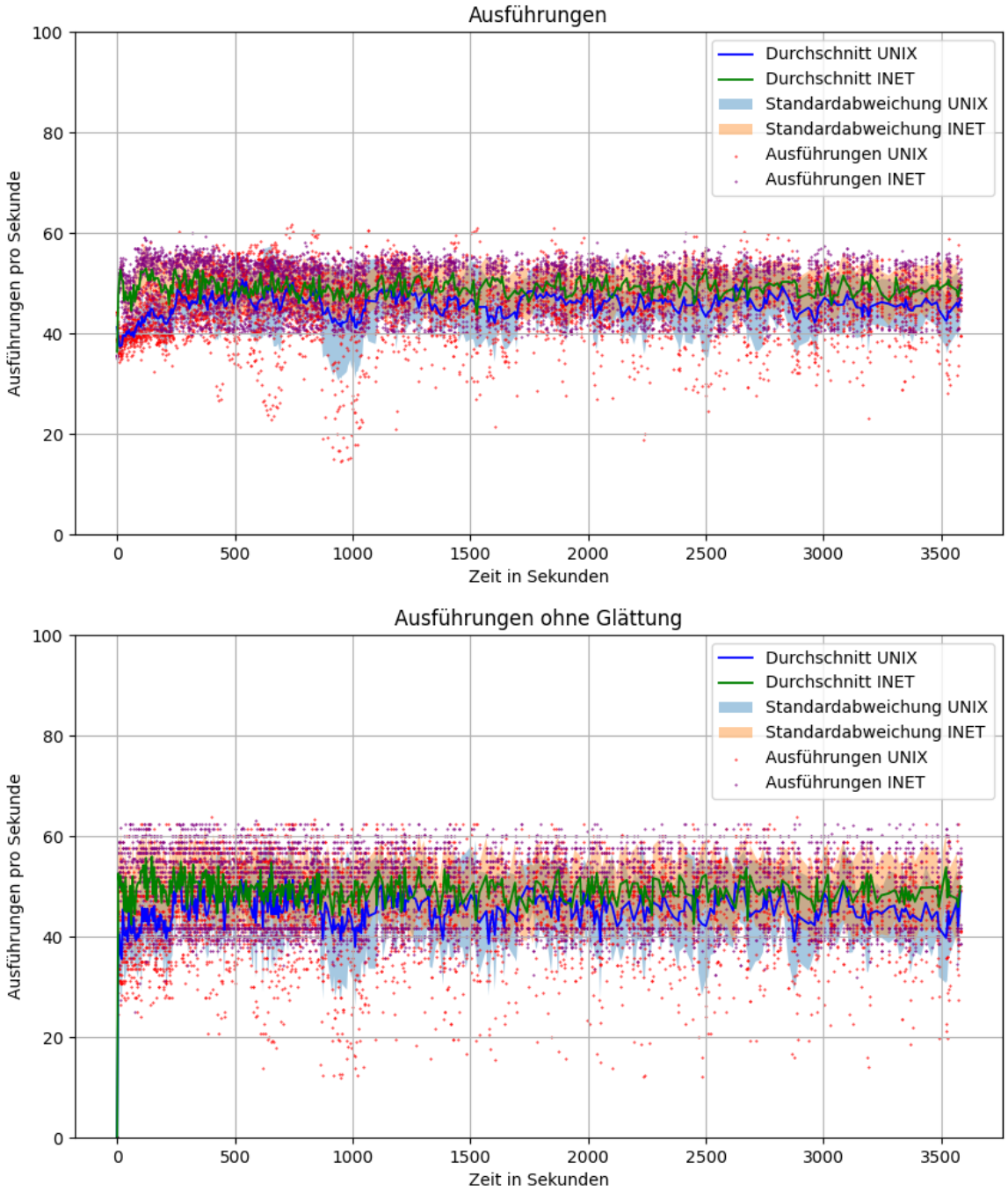

Abbildung 6.7: Anzahl an Ausführungen pro Sekunde zwischen UNIX mit TCP und INET mit TCP.

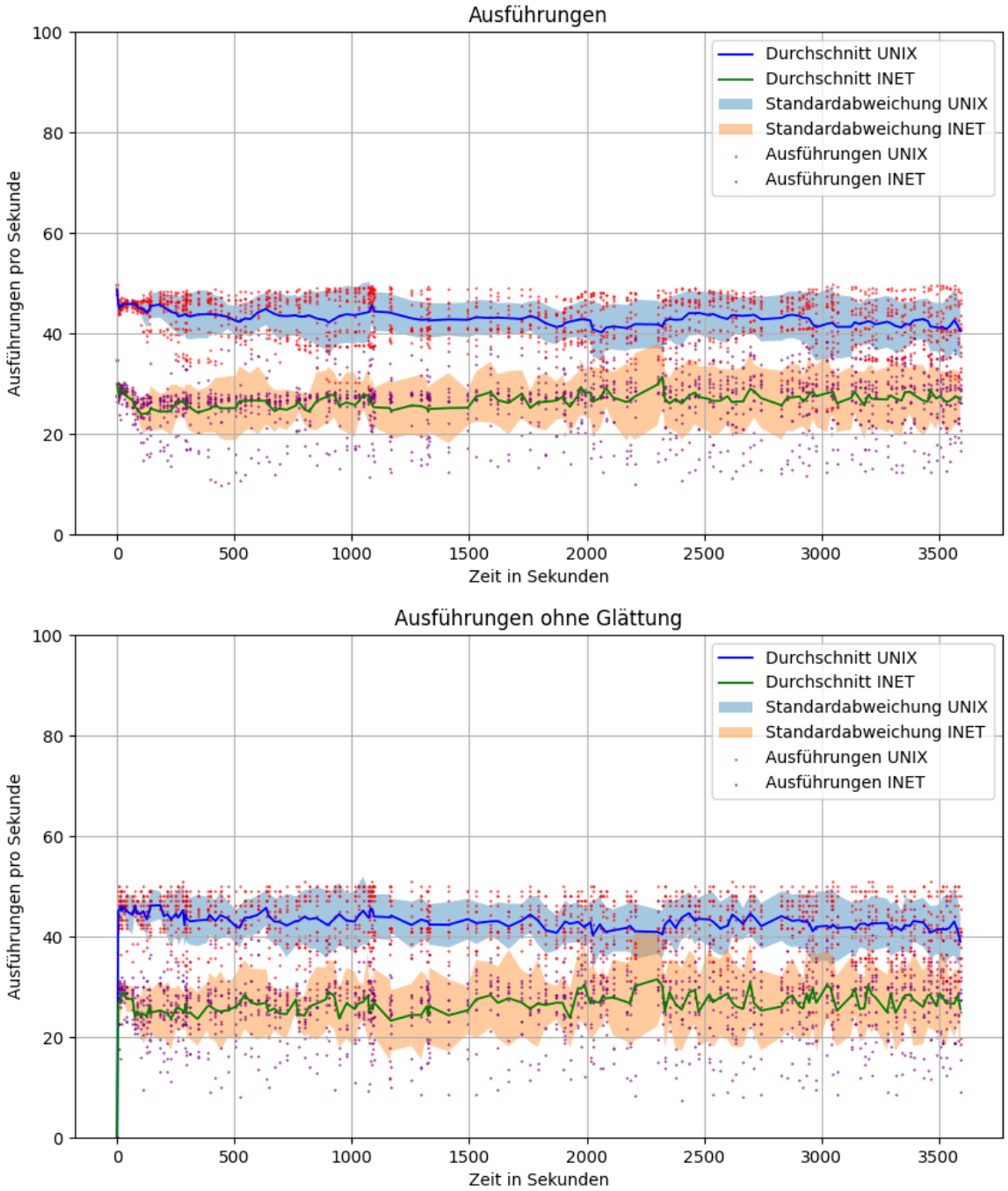

Abbildung 6.8: Anzahl an Ausführungen pro Sekunde zwischen UNIX mit UDP und INET mit UDP.

Wir betrachten zuerst Grafik 6.5. In beiden Abbildungen erkennen wir, dass die blaue Linie leicht über der grüne Linie verläuft und das sagt aus, dass durch den *UNIX-Socket* die Ausführungen pro Sekunde von AFLNET leicht höher sind. Auch auf beiden Abbildungen erkennen wir, dass die Streuung mit dem *UNIX-Socket* etwas geringer ist, was auch durch die Standardabweichung verdeutlicht wird. Aufgrund der allgemein geringen Streuung sticht die Standardabweichung nicht heraus. Wie erwartet ist der Verlauf der Ausführungen ohne Glättung ruckartiger als der Verlauf mit Glättung. Auf der unteren Abbildung mit den Ausführungen ohne Glättung können wir gut erkennen, dass sich durch die Ausführungen von *UNIX* eine Obergrenze abzeichnet. Das kann darauf hinweisen, dass der Prozessor hier die Ausführungen pro Sekunde begrenzt. Beim Überprüfen stellen wir fest, dass AFLNET beim Initialisieren den Prozess nur einem Prozessorkern zuweist und die Auslastung des Prozessorkerns sehr hoch ist. Wenn wir nun Grafik 6.6 betrachten, dann erkennen wir, dass die blaue Linie leicht unter der grünen Linie verläuft, was hier aussagt, dass die Ausführungen pro Sekunde mit dem *UNIX-Socket* niedriger sind, was jedoch überraschend ist. Eine Vermutung ist, dass es an dem Mehraufwand durch das *SLIP* und den zusätzlichen Daten, die mitgesendet werden müssen, liegt, was zur Folge hat, dass die Daten kopiert werden müssen. Eine andere Vermutung ist, dass es allgemein an dem Mehraufwand von DTLS gegenüber zu UDP liegen kann. Im Vergleich zu TLS und TCP ist bei DTLS und UDP eine leicht größere Streuung zu erkennen, was durch die Standardabweichungen erkennbar ist. Die Streuung resultiert vermutlich aus den unterschiedlichen UDP-Paketgrößen, die zum Übertragen genutzt werden, um die durch AFLNET generierten Daten zu übertragen. Interessant ist zu sehen, dass die Dichte an Ausführungen ab ungefähr 1250 Sekunden stark nachlässt. Das ist daran erkennbar, dass die Punkte ab diesem Zeitpunkt größere Lücken auf der Zeitachse aufweisen, was bedeutet, dass für ein Eintrag die Zeitabstände größer geworden sind. Aus diesem Grund ist die Anzahl der Einträge innerhalb von einer Stunde auch geringer. Wenn wir nun die Grafiken 6.5, 6.7 und 6.6 vergleichen fällt auf, dass die Ausführungen pro Sekunde bei TLS und TCP deutlich höher sind als bei DTLS. Wir haben bisher nicht herausgefunden, wodurch der Geschwindigkeitsunterschied entsteht. Wir vermuten, dass es an den unterschiedlichen Protokollabläufen liegt. Als Nächstes betrachten wir Grafik 6.7. Hier ist zu erkennen, dass die blaue Linie leicht unter der grünen Linie verläuft. Das bedeutet, dass die Anzahl an Ausführungen pro Sekunde hier beim *UNIX-Socket* etwas langsamer verläuft. Das ist ein unerwartetes Verhalten, besonders da in Grafik 6.5 mit *UNIX-Socket* und TLS die Anzahl an Ausführungen höher ist. Aus dem Grund vermuten wir hier ein Fehlverhalten mit TCP. Grafik 6.8 zeigt uns den Vergleich zwischen *UNIX* und *INET*

mit UDP. Hier sehen wir eine deutliche Steigerung vom *UNIX-Socket* gegenüber dem *INET-Socket*. Die Anzahl an Ausführungen pro Sekunde beim *UNIX-Socket* ist ungefähr doppelt so hoch wie beim *INET-Socket*, und die Standardabweichung ist leicht geringer.
Wir können festhalten, dass wir unsere Eingangsfrage, ob die Anzahl an Ausführung pro Sekunde mit dem *UNIX-Socket* erhöht wird, nicht eindeutig beantworten können. Zwar wurde bei TLS die Anzahl an Ausführungen pro Sekunde etwas erhöht und bei UDP können wir eine deutliche Steigerung vermerken, aber bei DTLS und TCP zeigen die Grafiken mit *UNIX-Sockets* eine niedrigere Anzahl an Ausführung pro Sekunde. Interessant ist noch zu erwähnen, dass wir eine leichte Korrelation erkennen, wenn wir die Anzahl der Einträge (siehe Grafik 6.3) mit der Anzahl an Ausführungen pro Sekunde vergleichen. Also umso höher die Anzahl an Ausführungen pro Sekunde ist, desto mehr Einträge sind vorhanden. Das haben wir auch in Abschnitt 6.2 zur Erläuterung der Anzahl an Einträgen vermutet. Wenn wir die Ergebnisse mit der Performanz vergleichen, wie in unseren Anforderungen beschrieben (siehe Abschnitt 4.1), dann sehen wir sehr deutlich, dass wir nicht an die Anzahl an Ausführungen pro Sekunde von 100 oder 500 gelangen. Von der durchschnittlichen Anzahl an Ausführungen pro Sekunde unserer Referenzwerte 100 und 500 konnten wir folgende Ergebnisse erreichen:
Beim Fuzzing-Prozess konnten wir mit dem *UNIX-Socket* und dem Referenzwert 100 im Durchschnitt einen prozentualen Anteil von 55,56 % für TLS, 24,35 % für DTLS, 45.39 % für TCP und 42,79 % für UDP erreichen. Mit dem Referenzwert 500 konnten wir im Durchschnitt einen prozentualen Anteil von 11,11 % für TLS, 4,87 % für DTLS, 9,08 % für TCP und 8,56 % für UDP erreichen.
Da wir im Vergleich zum *INET-Socket* kein eindeutiges Ergebnis bekommen haben, haben wir uns überlegt, die Anzahl an Ausführungen pro Sekunde nur für den Sendeprozess, die Verarbeitung des libcoap-Servers und vom Empfangsprozess zu berechnen, um überprüfen zu können, ob ein signifikanter Unterschied zwischen *INET-* und *UNIX-Sockets* erkennbar ist. Das wird mit den folgenden Grafiken 6.9, 6.10, 6.11 und 6.12 in derselben Reihenfolge wie bei der Auswertung zu der Anzahl an Ausführungen pro Sekunden von AFLNET, dargestellt.

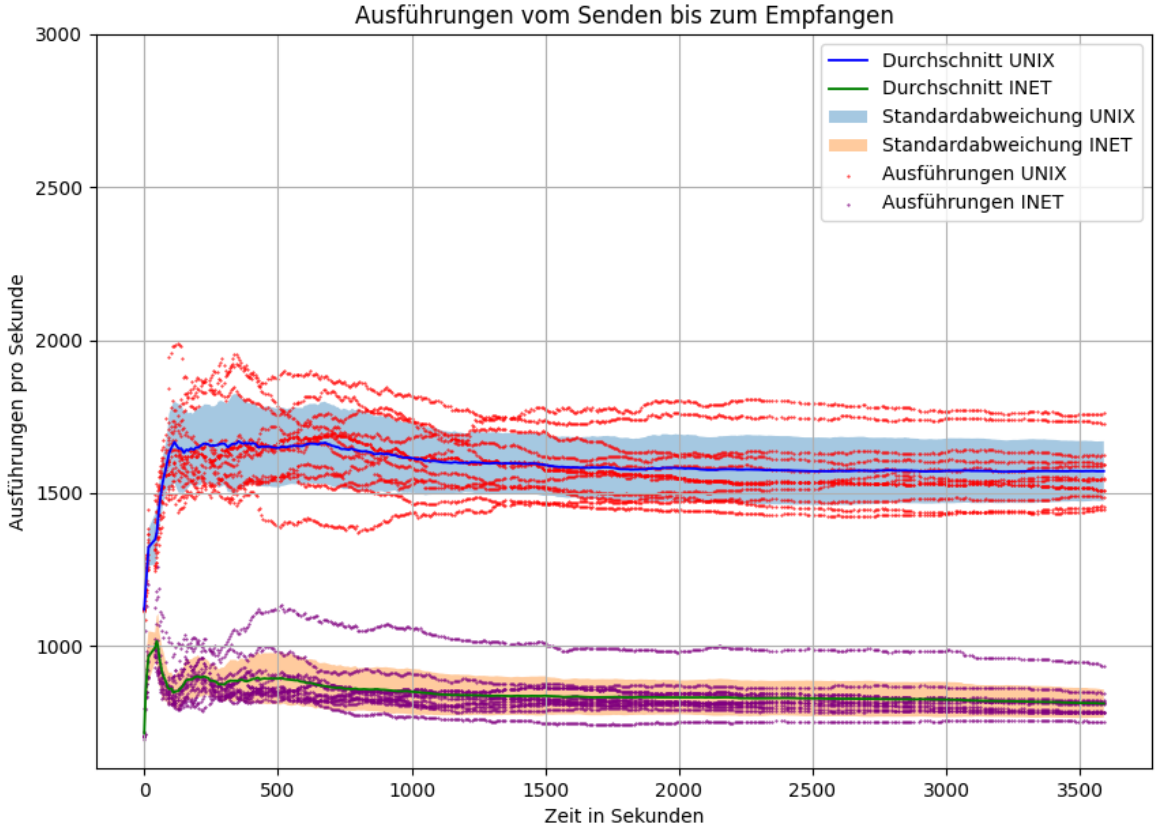

Abbildung 6.9: Ausführungen des Sende-, Bearbeitungs- und Empfangsprozesses zwischen UNIX mit TLS und INET mit TCP.

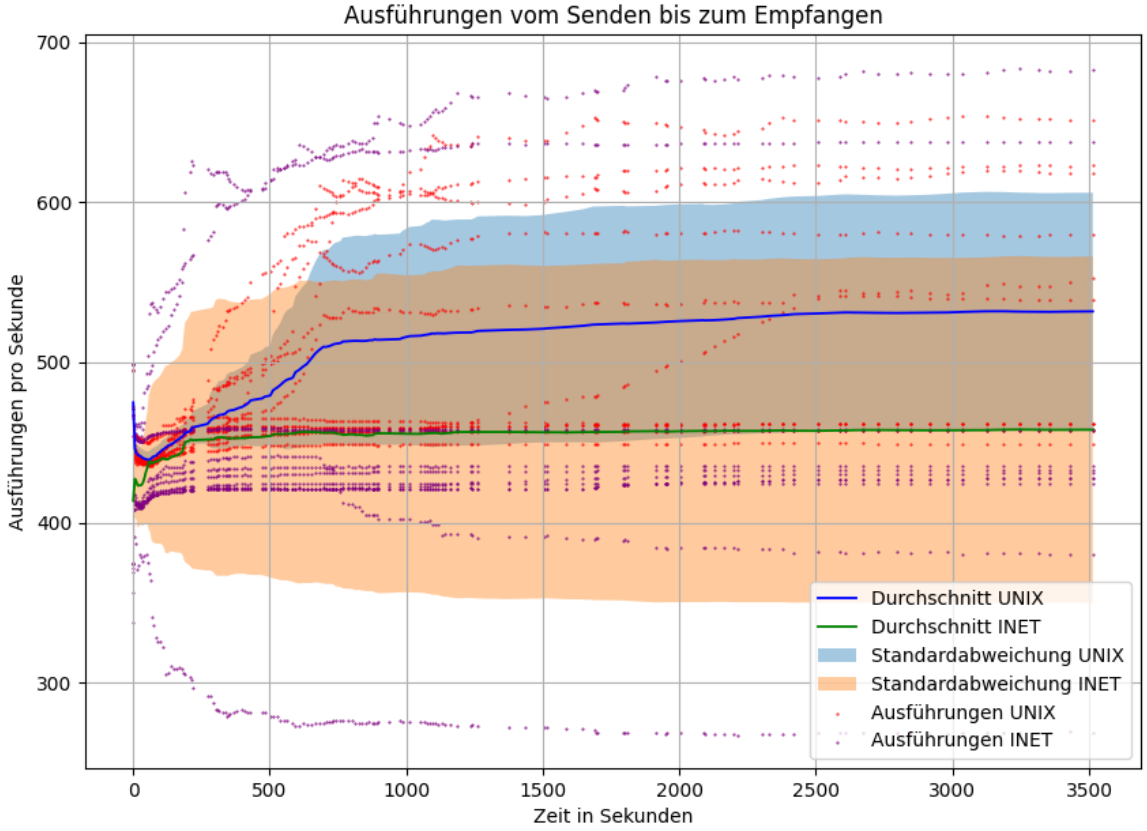

Abbildung 6.10: Ausführungen des Sende-, Bearbeitungs- und Empfangsprozesses zwischen UNIX mit DTLS und INET mit UDP.

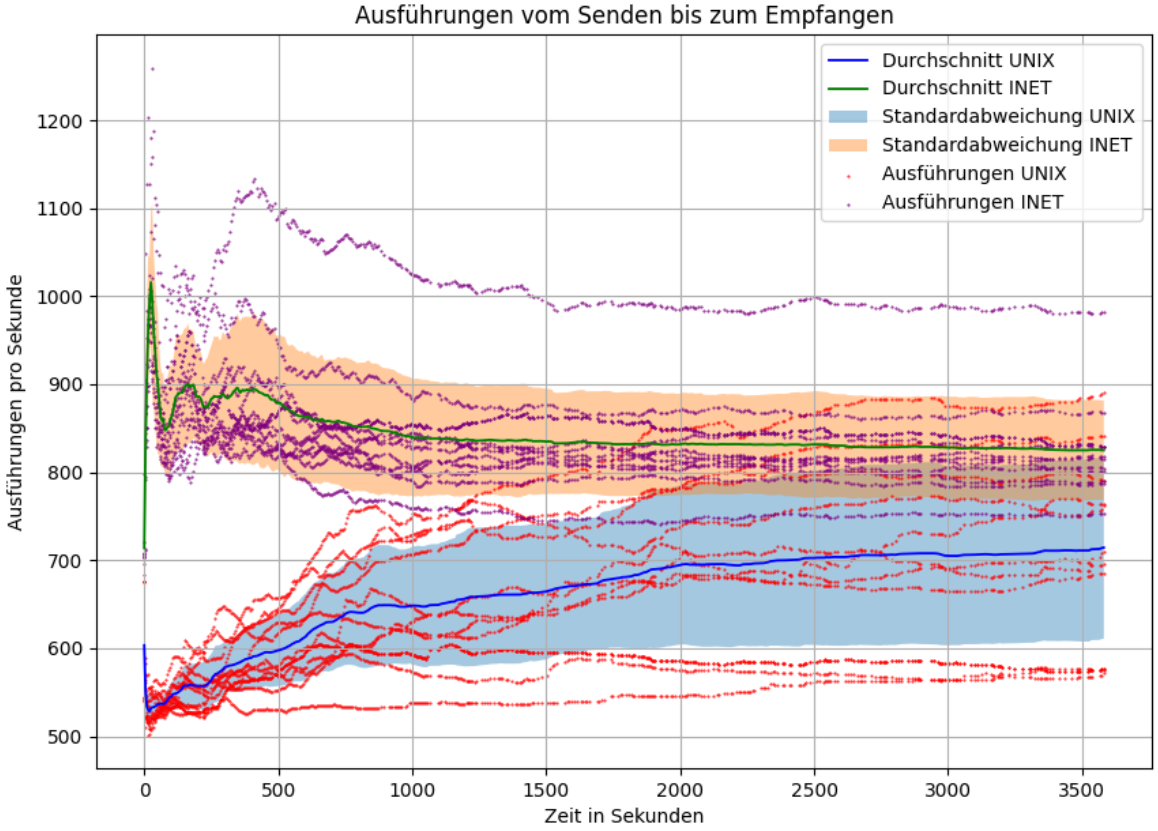

Abbildung 6.11: Ausführungen des Sende-, Bearbeitungs- und Empfangsprozesses zwischen UNIX mit TCP und INET mit TCP.

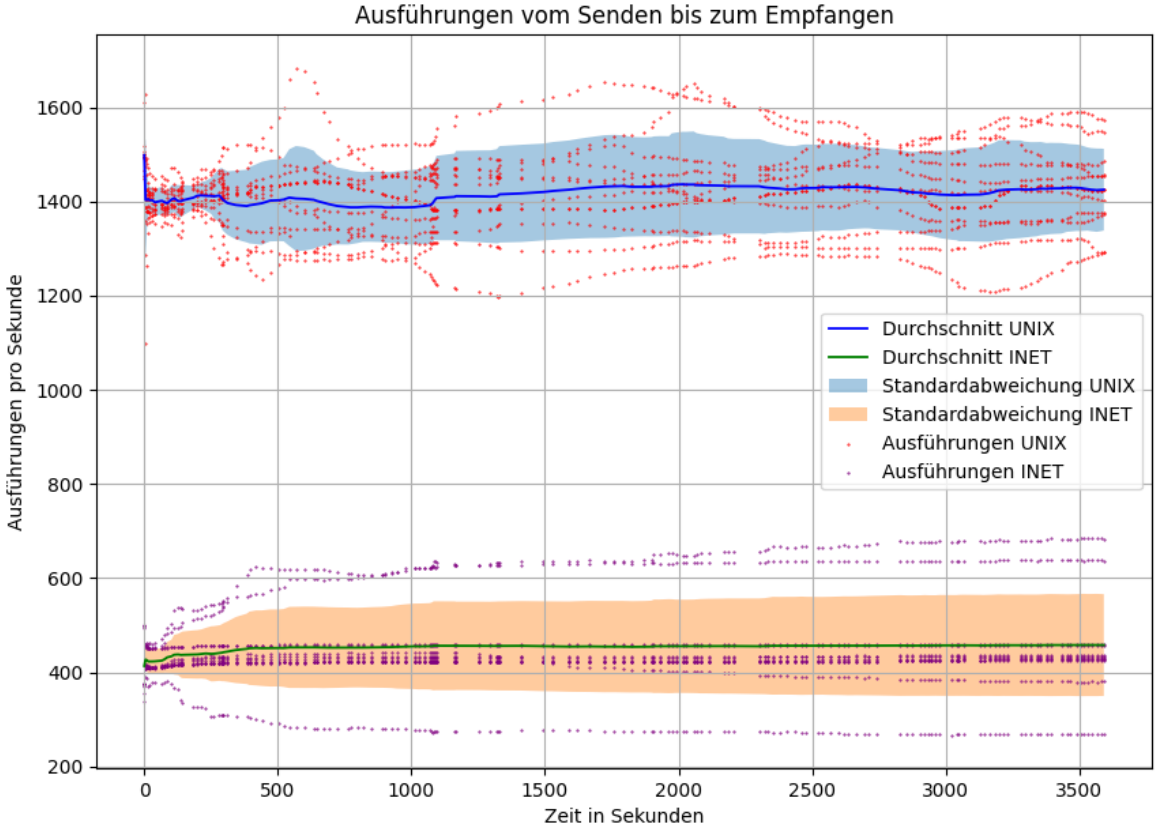

Abbildung 6.12: Ausführungen des Sende-, Bearbeitungs- und Empfangsprozesses zwischen UNIX mit UDP und INET mit UDP.

Wir betrachten wieder zuerst Grafik 6.9 für TLS. Es wird auf dieser Grafik sehr deutlich, dass die Anzahl an Ausführungen pro Sekunde für den Sende-, Bearbeitungs- und Empfangsprozess mit dem *UNIX-Socket* deutlich schneller verläuft. Dass mit dem *UNIX-Socket* die Anzahl an Ausführungen pro Sekunde höher ist, entspricht unseren Erwartungen und erreicht hier eine fast doppelte Anzahl an Ausführungen pro Sekunde, wenn wir die Linien betrachten, die den Durchschnitt beschreiben. Dass die durchschnittliche Linie weiter unten startet und dann schneller wird, ist vermutlich der in TCP verankerte *Slow-Start-Algorithmus*, der dafür sorgt, dass die Übertragungsmenge sich langsam an die optimale Übertragungsmenge annähert. Die einzelnen Durchläufe sind in dieser Grafik gut an den gepunkteten Linien erkennbar, da diese eine nicht so starke Streuung haben. Dass die Standardabweichung der einzelnen Durchgänge bei *UNIX-Socket* etwas höher ist als bei

INET-Socket, liegt vermutlich an störenden Prozessen oder es liegt daran, dass es Zufall ist, da die übertragenden Daten durch das *Fuzzing* von AFLNET generiert werden und damit unterschiedliche Datenmengen haben, wodurch dann die Anzahl an Ausführungen pro Sekunde schwankt.
Als Nächstes betrachten wir Grafik 6.10 für DTLS. Auch hier verläuft die durchschnittliche Linie vom *UNIX-Socket*, wie erwartet, deutlich über die vom *INET-Socket*. Was hier jedoch auffällt ist, dass die einzelnen Durchläufe mit der Anzahl an Ausführungen pro Sekunde stark schwankt. Besonders beim *INET-Socket* sehen wir in der Grafik einen Durchlauf, der eine niedrige Anzahl an Ausführungen pro Sekunde hat, die bei ungefähr 280 liegt. Wir sehen aber auch einen Durchlauf, bei dem die Anzahl an Ausführungen pro Sekunde bei ungefähr 770 Ausführungen pro Sekunde verläuft und damit ein Durchlauf ist, dessen Anzahl an Ausführungen höher ist, als es bei *UNIX-Socket* erreicht wurde. Zwischen 400 und 470 Ausführungen pro Sekunde können wir mehrere Durchläufe erkennen, die deutlich dicht zusammen verlaufen. Es sieht wieder danach aus, als wenn es durch den Prozessorkern zu einer Begrenzung kommt, jedoch sprechen die Durchläufe, die eine höhere Ausführung pro Sekunde haben, dagegen. Betrachten wir Grafik 6.11, sehen wir, was auch Grafik 6.7 vermuten ließ. Auch die Ausführungen pro Sekunde beim Sende-, Bearbeitungs- und Empfangsprozess sind langsamer. Wir können also eingrenzen, dass es beim Übertragungsprozess oder bei der Bearbeitung des libcoap-Servers zu dem Geschwindigkeitsverlust kommt. Auch die Standardabweichung wird beim *UNIX-Socket* mit der Zeit größer. Wir vermuten, dass wenn wir den Fuzzing-Prozess mit TCP durchführen, dann entsteht ein Fehlverhalten. Dieses Fehlverhalten könnte daran liegen, dass wir für TCP das *SLIP* mit den zusätzlichen Daten nicht mitsenden und libcoap die Session nicht richtig zuordnen kann. Grafik 6.12 zeigt uns, dass wir im Vergleich zum *INET-Socket* mit UDP, eine Steigerung an Ausführungen pro Sekunde von Faktor drei haben, was unseren Erwartungen entspricht.
Die Ausführungen pro Sekunde von AFLNET werden von AFLNET als langsam bezeichnet, wenn die Ausführungen pro Sekunde unter 100 liegen. Idealerweise soll der Wert über 500 liegen. Also fragen wir uns, wie schnell der Sende-, Bearbeitungs- und Empfangsprozess sein muss, um über diesen Wert zu gelangen, beziehungsweise wie schnell wäre AFLNET mit dem libcoap-Server ohne den Sende-, Bearbeitungs- und Empfangsprozess. Dafür haben wir den Sende-, Bearbeitungs- und Empfangsprozess herausgerechnet. Die folgenden Grafiken 6.13, 6.14, 6.15 und 6.16 haben wir wieder in derselben Reihenfolge dargestellt, wie bei der Auswertung zu der Anzahl an Ausführungen pro Sekunde von AFLNET.

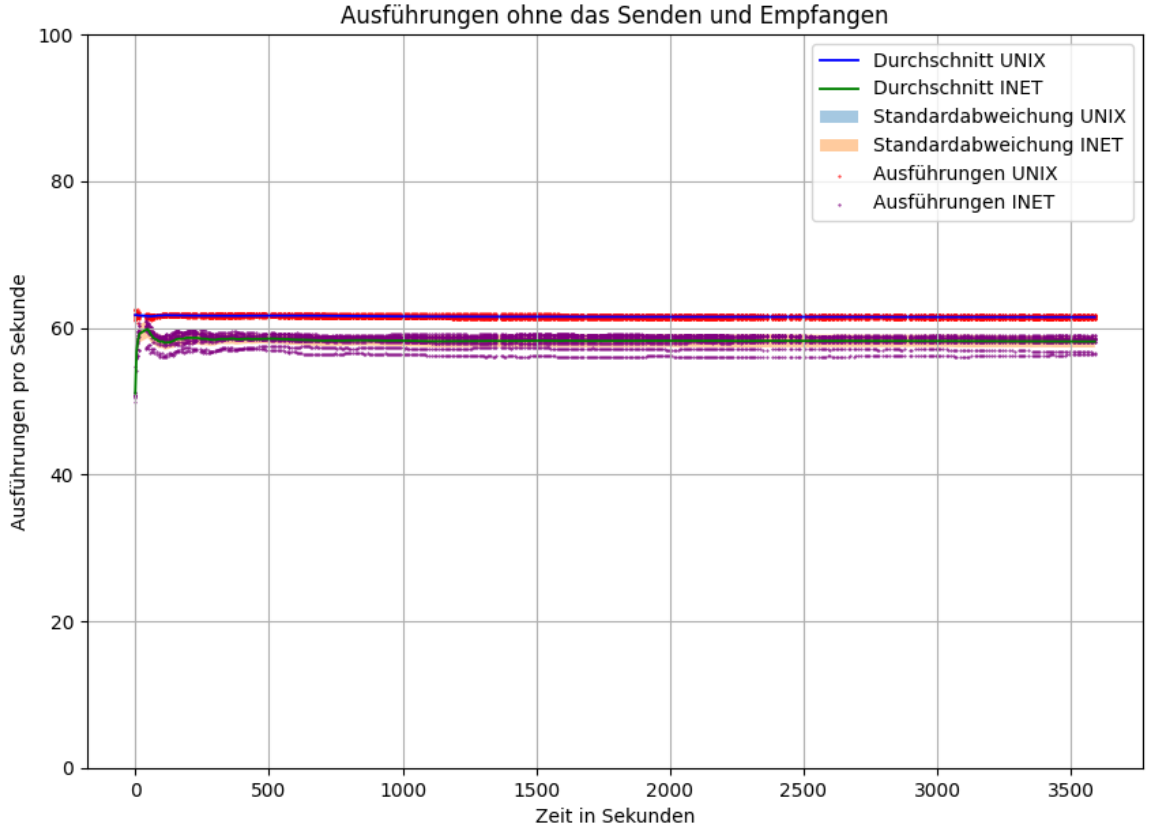

Abbildung 6.13: Ausführungen von AFLNET ohne den Sende-, Bearbeitungs- und Empfangsprozess zwischen UNIX mit TLS und INET mit TCP.

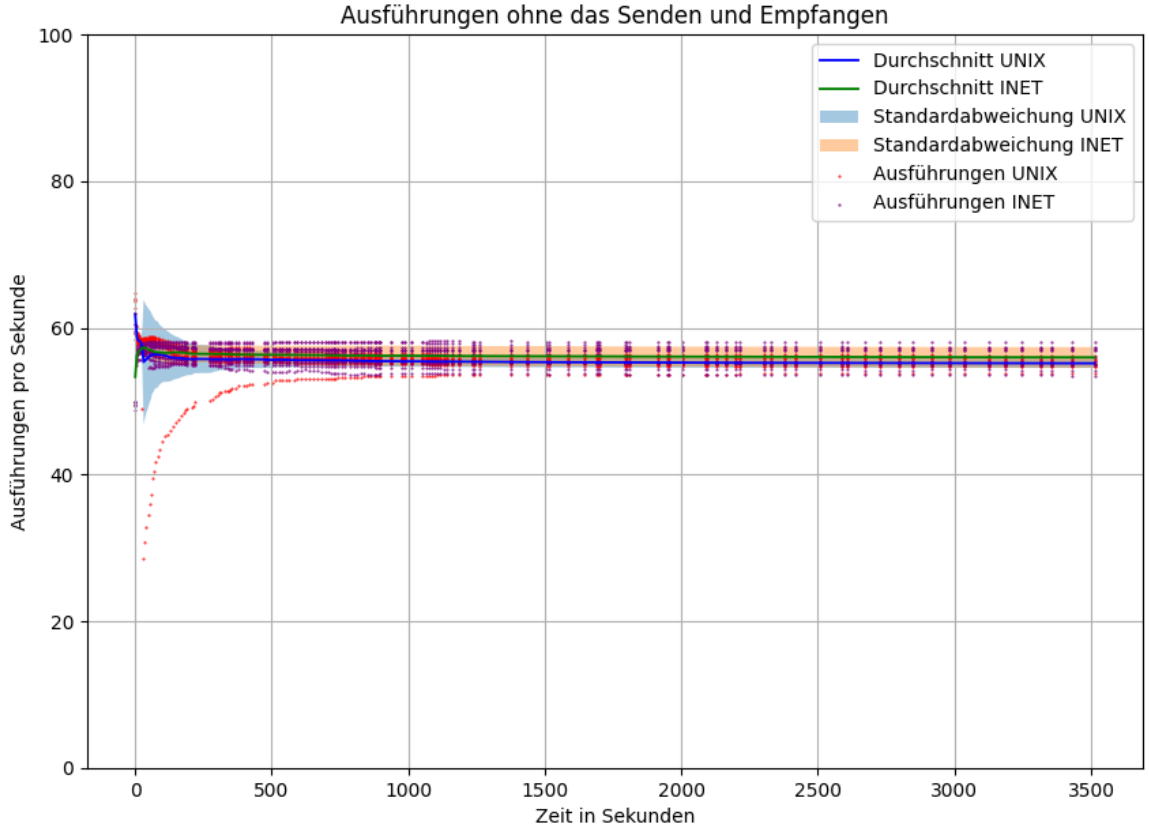

Abbildung 6.14: Ausführungen von AFLNET ohne den Sende-, Bearbeitungs- und Empfangsprozess zwischen UNIX mit DTLS und INET mit UDP.

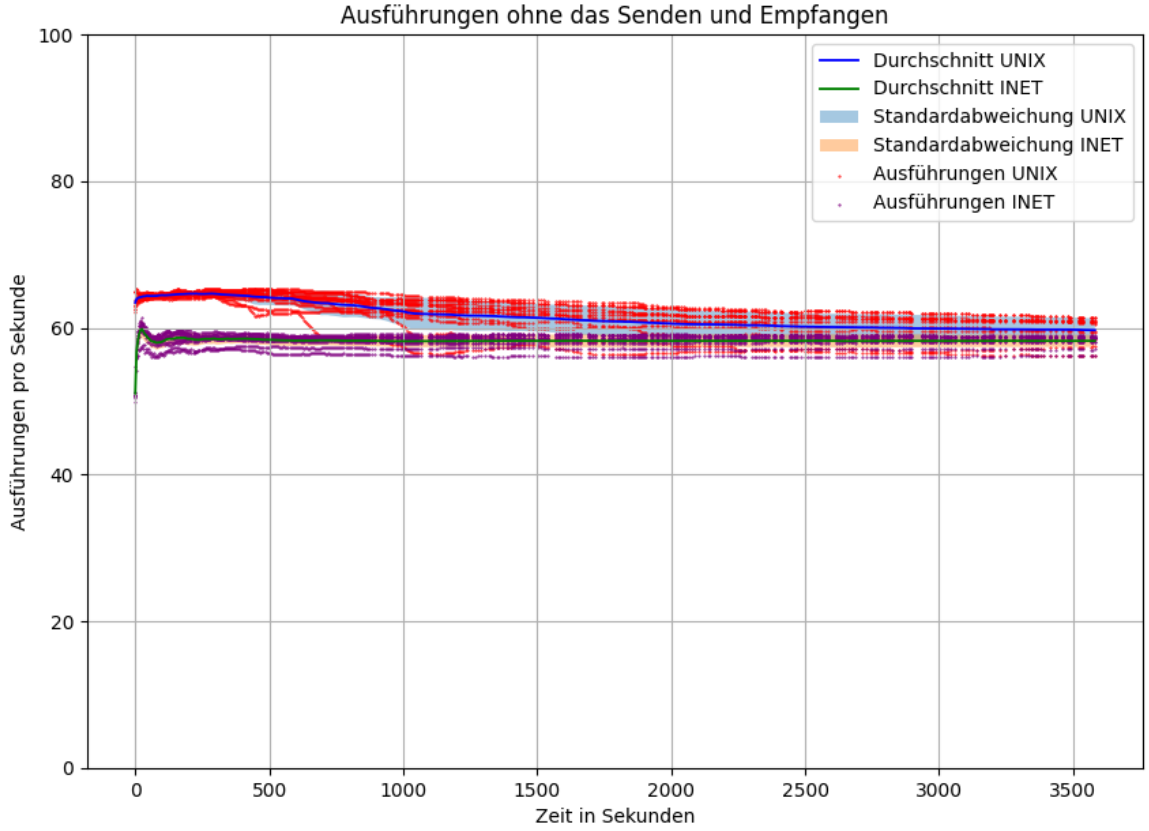

Abbildung 6.15: Ausführungen von AFLNET ohne den Sende-, Bearbeitungs- und Empfangsprozess zwischen UNIX mit TCP und INET mit TCP.

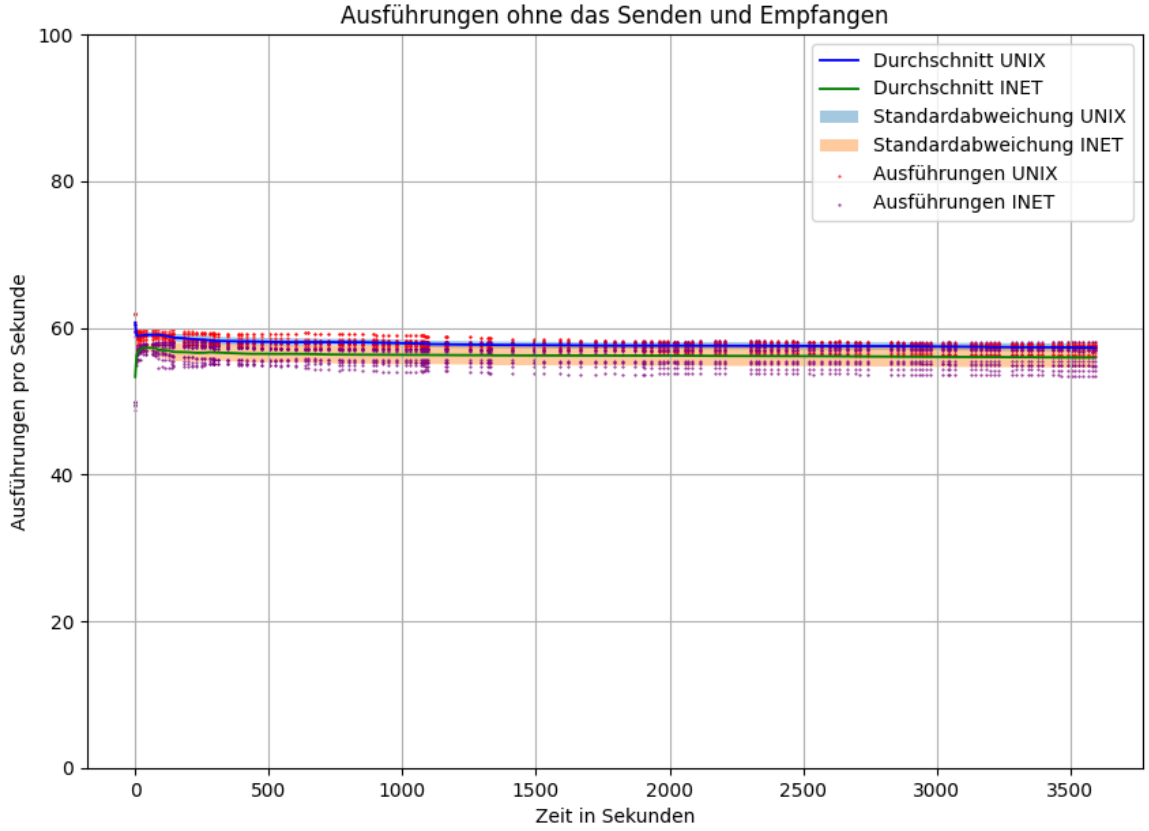

Abbildung 6.16: Ausführungen von AFLNET ohne den Sende-, Bearbeitungs- und Empfangsprozess zwischen UNIX mit UDP und INET mit UDP.

Wir erkennen direkt, dass die Anzahl an Ausführungen pro Sekunden in den Grafiken sehr konstant und fast gleich hoch sind. Das bedeutet, dass die Schwankungen, die wir bei der Anzahl an Ausführungen pro Sekunde von AFLNET bekommen, zum größten Teil durch den Sende-, Bearbeitungs- und Empfangsprozess verursacht werden. Den angestrebten Wert von 100 oder 500 konnten wir damit nicht erreichen. In unserem Fall konnten wir die maximalen Werte bei TCP in Grafik 6.15 erreichen. Die maximale Anzahl an Ausführungen pro Sekunde bei TCP war 65,38 und als maximale durchschnittliche Anzahl an Ausführungen pro Sekunde konnten wir den Wert 64,67 erreichen. Die Frage, wie schnell der Sende-, Bearbeitungs- und Empfangsprozess sein muss, damit wir über den Wert 100 beziehungsweise 500 gelangen, können wir damit beantworten, dass in diesem Fall die Performanz ohne den Sende-, Bearbeitungs- und Empfangsprozess nur bis zu einem gewissen

Punkt die Anzahl an Ausführungen pro Sekunde von AFLNET beeinflussen kann. Auch ohne den Sende-, Bearbeitungs- und Empfangsprozess konnten wir nur für unseren Referenzwert von 100 Ausführungen pro Sekunde im Durchschnitt den prozentualen Anteil von 61,55 % für TLS, 55,68 % für DTLS, 62,05 % für TCP und 57,84 % für UDP erreichen. Für den Referenzwert von 500 Ausführungen pro Sekunde konnten wir im Durchschnitt den prozentualen Anteil von 12,31 % für TLS, 11,14 % für DTLS, 12,41 % für TCP und 11,57 % für UDP erreichen.
Wir können nun aus den erhobenen Daten Folgendes zusammenfassen:

1. Unserer Eingangsfrage, ob die Anzahl an Ausführung pro Sekunde von AFLNET mit dem *UNIX-Socket* erhöht wird, kann bedingt zugestimmt werden. Wir können mit TLS eine leichte und mit UDP eine starke Erhöhung feststellen, vgl. Grafiken 6.5 und 6.8, aber bei DTLS und TCP, vgl. Grafiken 6.6 und 6.7, sehen wir, dass die Anzahl an Ausführungen pro Sekunde von AFLNET etwas geringer ist. Natürlich müssen wir beim Vergleich des *UNIX-Sockets* mit TLS und DTLS gegenüber dem *INET-Socket* mit TCP und UDP berücksichtigen, dass TLS und DTLS einen Mehraufwand haben, welcher sich auf die Anzahl an Ausführungen pro Sekunde von AFLNET auswirkt. Zudem gehen wir davon aus, dass das Verhalten beim Fuzzing-Prozess mit TCP beim *UNIX-Socket* fehlerhaft ist, und somit könnte die Anzahl an Ausführungen pro Sekunde von AFLNET höher sein.

2. Wir konnten die Performanz, wie wir im Abschnitt 4.1 beschrieben haben, nicht erreichen. Auch wenn wir den Sende-, Bearbeitungs- und Empfangsprozess herausrechnen, bekommen wir keine signifikante Erhöhung der Anzahl an Ausführungen pro Sekunde von AFLNET. Wir sehen, dass die Ausführung pro Sekunde von AFLNET sich besonders bei den Protokollen, die auf UDP basieren, durch den fehlenden Sende-, Bearbeitungs- und Empfangsprozess, profitieren. Das bedeutet natürlich auch, dass die Protokolle, die auf UDP basieren, einen langsameren Sende-, Bearbeitungs- und Empfangsprozess haben.

6.4 Diskussion

Im Abschnitt 6.3 haben wir die Bedeutung der Ergebnisse erläutert. In diesem Abschnitt reflektieren wir über einige Ergebnisse aus diesem Abschnitt.

- **Die Performanz des Fuzzing-Prozesses:** Wenn wir unsere Ergebnisse mit den Anforderungen in Abschnitt 4.1 zur Performanz vergleichen, erkennen wir, dass wir die Anzahl an Ausführungen pro Sekunde auch mit dem *UNIX-Socket* nicht erreichen konnten. Beim Start des Fuzzing-Prozesses wird uns die Warnung ausgesprochen, dass der libcoap-Server, den wir verwenden, langsam ist. Da wir durch den *UNIX-Socket* nicht die gewünschten Ergebnisse erreicht haben, wäre es interessant, die Tipps zu befolgen, die beim *AFL User Guide* [15] angemerkt werden. Auch in der Datei *README.llvm* [2] gibt es einige Verbesserungsmöglichkeiten. Da steht z. B. beschrieben, dass es sein kann, dass Server beim Starten viele Ressourcen initialisieren. Dadurch kann es zu Einbußen bei der Performanz kommen, wenn AFLNET den Server in einen neuen Prozess abspaltet. In so einem Fall könnte der Zeitpunkt, an dem der Prozess abgespalten wird, nach hinten verschoben werden, sodass die Initialisierung dann abgeschlossen ist. Das wäre vermutlich eine Herangehensweise, die die Anzahl an Ausführungen pro Sekunde erhöhen könnte. Mit *Shared-Memory* könnten wir vermutlich noch bessere Ergebnisse erreichen, aber wir gehen davon aus, dass die Ergebnisse nicht die angestrebte Anzahl an Ausführungen pro Sekunde von über 500 erreichen und wahrscheinlich auch nicht den Wert von 100 erreichen, was aus der Vermutung hervorgeht, dass ohne den Sende-, Bearbeitungs- und Empfangsprozess die Anzahl an Ausführungen pro Sekunde nicht signifikant gesteigert wurde. Wenn wir uns noch einmal Grafik 6.5 anschauen, dann sehen wir, dass sich auf der unteren Abbildung eine Grenze abbildet. Das erkennen wir an den roten Punkten, deren Werte eine Maximalgrenze bilden. Das könnte ein Hinweis sein, dass der Prozessorkern mit dem Fuzzing-Prozess langsamer ist als der Übertragungsprozess durch TLS. Da AFLNET dem Fuzzing-Prozess nur einen Prozessorkern zuweist, könnte ein Ansatz sein, dem Prozess nachträglich mehrere Prozessorkerne zuzuweisen oder einen schnelleren Prozessor zu verwenden, damit der Prozessor nicht der Flaschenhals ist.

- **Schwankungen unterhalb der einzelnen Durchläufe:** In Abschnitt 6.3 wurde die Vermutung geäußert, dass die Schwankungen durch störende Prozesse oder beim Sende-, Bearbeitungs- und Empfangsprozess durch die generierten Daten vom *Fuzzer* entstehen. Dies erkennen wir besonders daran, dass ohne den Sende-, Bearbeitungs- und Empfangsprozess die Streuung zwischen den Durchläufen deutlich geringer ist. Ob es an den generierten Daten vom *Fuzzer* liegt, könnte noch herausgefunden werden, wenn wir die Größe der gesendeten Daten speichern und mit den Schwankun-

gen desselben Durchlaufes abgleichen. Es könnte auch damit zusammenhängen, wie lange der libcoap-Server benötigt, um die unterschiedlichen generierten Daten, die von AFLNET kommen, zu bearbeiten. Dazu könnte auf Seiten des libcoap-Servers ebenfalls eine Zeitmessung eingerichtet werden, um beim Bearbeitungsprozess zu ermitteln, wie die Bearbeitungszeit im Vergleich zu den übertragenden Daten ist.

- **Menge der erhobenen Daten:** In den Grafiken 6.2 und 6.4 haben wir gezeigt, wie viele Stunden und erhobene Daten wir für die Auswertung herangezogen haben. Die erhobenen Daten geben uns schon einen guten Eindruck, wie die Tendenz der Anzahl an Ausführungen pro Sekunde verläuft, wodurch wir einige Aussagen machen konnten, aber wir sehen auch in Grafik 6.10, dass eine große Streuung zwischen den einzelnen Durchläufen besteht. Da könnten wir noch mehr Durchläufe benötigen, um eine bessere Tendenz zu erkennen.
- **Fehlverhalten und weitere Erkenntnisse:** In dieser Bachelorarbeit konnten wir nicht alle Fragen beantworten, die zu neuen Erkenntnissen führen oder zur Entdeckung beziehungsweise Behebung eines Fehlverhaltens. Das Problem dabei ist, dass wir viel Zeit benötigen, um einen Durchlauf zu bekommen. Das bedeutet, dass wir bei Anpassungen eines Fehlverhaltens oder für weitere Erkenntnisse zwischen elf und 66 Stunden benötigen. Für die Verbesserung des Fuzzing-Prozesses über den *UNIX-Socket* benötigen wir also weitere Zeitinvestitionen.

7 Fazit und Ausblick

Diese Arbeit hatte zum Ziel, eine Anpassung an libcoap zusammen mit AFLNET umzusetzen, sodass der Fuzzing-Prozess über *UNIX-Domain-Socket* oder eventuell auch über *Shared-Memory* neben den *INET-Sockets* realisiert wird. In Kapitel 5 zur Implementation haben wir besprochen, wie wir das Ziel für *UNIX-Domain-Socket* umgesetzt haben. Für UDP haben wir die Implementation mit SLIP umgesetzt, jedoch bei TCP nicht. Die optionale Implementation für *Shared-Memory* wurde nicht umgesetzt. Im Kapitel 6, besonders im Abschnitt 6.3 konnten wir einige Fragestellungen zur Performanz, wie im Abschnitt 4.1 beschrieben, beantworten. Wir konnten mit dem *UNIX-Socket* gegenüber dem *INET-Socket* bei UDP die Anzahl an Ausführungen pro Sekunde von durchschnittlich 26,52 auf 42,79 erhöhen. Bei TLS konnten wir die Anzahl an Ausführungen pro Sekunde von durchschnittlich 48,89 auf 55,56 anheben. In Abschnitt 6.3 haben wir festgestellt, dass eine Steigerung der Performanz bezogen auf den gesamten Fuzzing-Prozess erkennbar ist. Einen derart hohen Zuwachs der Performanz konnten wir bei TCP von durchschnittlich 49,03 auf 45,39 Ausführungen pro Sekunde und bei DTLS von durchschnittlich 26,48 auf 24,35 Ausführungen pro Sekunde mit dem *UNIX-Socket* gegenüber dem *INET-Socket* nicht verbuchen.
Aus den Ergebnissen, die wir bei der Evaluation der Performanz und beim Vergleich zwischen *UNIX-Sockets* und *INET-Sockets* gewonnen haben, können wir darauf aufbauend weitere Fragen für den Ausblick formulieren. Wir sehen, dass wir ohne den Sende-, Bearbeitungs- und Empfangsprozess keine signifikante Steigerung erkennen können und auch mit der Verwendung des *Shared-Memory* können wir unter diesen Bedingungen keine Steigerung erwarten. Jedoch haben wir auch bemerkt, dass sich eine Begrenzung der Anzahl an Ausführungen pro Sekunde abbildet. In Anhang A.3 haben wir einen Vergleich zwischen TCP und UDP mit dem *UNIX-Socket* und dabei haben wir dem Prozess mehrere Prozessorkerne nachträglich zugewiesen. Da ist ersichtlich, dass wir die Anzahl an Ausführungen pro Sekunde mit mehreren Prozesskernen gegenüber einem Prozessorkern deutlich erhöhen konnten, und der Prozessorkern begrenzt den Fuzzing-Prozess nicht

mehr in diesem Umfang. Also wäre es interessant, wie der Fuzzing-Prozess durch den *Shared-Memory* profitieren kann, wenn der Prozessor den Prozess nicht begrenzt. Wenn wir die Idee mit mehreren Prozessorkernen weiter verfolgen, könnten wir uns überlegen, den Fuzzing-Prozess mit einer Grafikkarte zu realisieren. In dem Fachartikel *Graphics Card Based Fuzzing* von *Mower et al.* [16] wurde mit einem *Fuzzer*, der auf dem *American Fuzzy Lop (AFL)* [1] basiert und den Fuzzing-Prozess über einen Grafikkartenprozessor realisiert, getestet. Die Ergebnisse zeigen zwar, dass der Prozessor noch zu bevorzugen ist, jedoch wird dort angegeben, dass mit verbesserter Parallelisierbarkeit der Grafikkartenprozessor Potenzial hat. Eine weitere interessante Frage wäre, ob wir die Ursache der langsamen Geschwindigkeit durch die Hilfestellung im *AFL User Guide* [15] und *README.llvm* [2] besser eingrenzen können, da wir auch eine Warnung von AFLNET bekommen, die besagt, dass der libcoap-Server langsam ist, worauf wir im Abschnitt 6.4 genauer eingegangen sind.

[1] American Fuzzy Lop: `https://github.com/google/AFL`

Literatur

[1] K. Kuladinithi, O. Bergmann, T. Pötsch, M. Becker und C. Görg, „Implementation of CoAP and its Application in Transport Logistics,“ 2011.

[2] V.-T. Pham, M. Böhme und A. Roychoudhury, „AFLNet: A Greybox Fuzzer for Network Protocols,“ in *Proceedings of the 13rd IEEE International Conference on Software Testing, Verification and Validation : Testing Tools Track*, 2020.

[3] *Definition of a socket*, RFC 147, Mai 1971. DOI: 10.17487/RFC0147. Adresse: https://rfc-editor.org/rfc/rfc147.txt (besucht am 01. 11. 2021).

[4] M. Geuking, *Entwurf und Implementierung eines Systems zum automatisierten Testen von Kommunikationsabläufen in libcoap mittels Fuzzing.* Bachelorarbeit, Universität Bremen, 2021.

[5] G. Klees, A. Ruef, B. Cooper, S. Wei und M. Hicks, „Evaluating Fuzz Testing,“ *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018.

[6] Y. Zeng, M. Lin, S. Guo, Y. Shen, T. Cui, T. Wu, Q. Zheng und Q. Wang, „MultiFuzz: A Coverage-Based Multiparty-Protocol Fuzzer for IoT Publish/Subscribe Protocols,“ *Sensors*, Jg. 20, Nr. 18, 2020, ISSN: 1424-8220. DOI: 10.3390/s20185194. Adresse: https://www.mdpi.com/1424-8220/20/18/5194 (besucht am 01. 11. 2021).

[7] Ion Gaztañaga (igaztanaga at gmail dot com), *Memory Mapped Files And Shared Memory For C++*, Juni 2006. Adresse: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2044.html (besucht am 01. 11. 2021).

[8] V. J. M. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz und M. Woo, „Fuzzing: Art, Science, and Engineering,“ *CoRR*, Jg. abs/1812.00140, 2018. arXiv: 1812.00140. Adresse: http://arxiv.org/abs/1812.00140 (besucht am 01. 11. 2021).

[9] Z. Shelby, K. Hartke und C. Bormann, *The Constrained Application Protocol (CoAP)*, RFC 7252, Juni 2014. DOI: 10.17487/RFC7252. Adresse: https://rfc-editor.org/rfc/rfc7252.txt (besucht am 01. 11. 2021).

[10] *Transmission Control Protocol*, RFC 793, Sep. 1981. DOI: 10.17487/RFC0793. Adresse: https://rfc-editor.org/rfc/rfc793.txt (besucht am 01. 11. 2021).

[11] *User Datagram Protocol*, RFC 768, Aug. 1980. DOI: 10.17487/RFC0768. Adresse: https://rfc-editor.org/rfc/rfc768.txt (besucht am 01. 11. 2021).

[12] E. Rescorla, *The Transport Layer Security (TLS) Protocol Version 1.3*, RFC 8446, Aug. 2018. DOI: 10.17487/RFC8446. Adresse: https://rfc-editor.org/rfc/rfc8446.txt (besucht am 01. 11. 2021).

[13] E. Rescorla und N. Modadugu, *Datagram Transport Layer Security Version 1.2*, RFC 6347, Jan. 2012. DOI: 10.17487/RFC6347. Adresse: https://rfc-editor.org/rfc/rfc6347.txt (besucht am 01. 11. 2021).

[14] *Nonstandard for transmission of IP datagrams over serial lines: SLIP*, RFC 1055, Juni 1988. DOI: 10.17487/RFC1055. Adresse: https://rfc-editor.org/rfc/rfc1055.txt (besucht am 01. 11. 2021).

[15] Google, *AFL User Guide*, 2019. Adresse: https://afl-1.readthedocs.io/en/latest/user_guide.html#stage-progress (besucht am 03. 02. 2022).

[16] R. Mower, B. Bernard und J. Straub, „Graphics Card Based Fuzzing,“ Nov. 2019, S. 111–115. DOI: 10.1109/MASSW.2019.00029.

Abbildungsverzeichnis

Listings

Glossar

Socket Ist ein Zwischenspeicher und dient als Endpunkt, um Daten zwischen Prozessen auszutauschen.

AFLNET Ein Grey-Box Fuzzer für Netzwerkprotokolle.

libcoap libcoap ist eine Implementierung des RFC7252 *The Constrained Application Protocol* (CoAP)[9] und bietet ein Web-Übertragungsprotokoll für eingeschränkte Geräte und eingeschränkte Netze an.

CoAP Constrained Application Protocol ist ein Web-Übertragungsprotokoll, das für eingeschränkte Netzwerke und Geräte entwickelt wurde.

DTLS Datagram Transport Layer Security ist ein auf TLS basierendes Protokoll, dass eine sichere Kommunikation bieten und analog zu UDP, die Datagrammsemantik beibehält.

IoT Internet of Things sind Geräte mit Software, die über Netzwerke kommunizieren können. Sie können Sensoren oder anderen Techniken beinhalten.

IP Internet Protocol ist ein verbindungsloses, unzuverlässiges und nicht sequenzielles Protokoll, das eine Adressierung über Netzwerke ermöglicht.

IPv4 Internet Protocol version 4 ist das Internetprotkoll, das für die Übertragung von Daten über Netzwerke verwendet wird. Die Version 4 nutzt einen 32-Bit Adressraum.

IPv6 Internet Protocol version 4 ist das Internetprotkoll, das für die Übertragung von Daten über Netzwerke verwendet wird. Die Version 6 nutzt einen 128-Bit Adressraum.

LAN Local Area Network beschreibt ein Netzwerk, dass Computer in einem näheren Bereich verbindet.

PUT Program Under Test ist das Programm, dass getestet wird.

SLIP Serial Line Internet Protocol ist ein einfaches Protokoll, dass über einer serielle Punkt-zu-Punkt-Verbindung kommuniziert.

TCP Transmission Control Protocol ein verbindungsorientiertes und zuverlässiges Netzwerkprotokoll.

TLS Transport Layer Security ist ein auf TCP basierendes Protokoll, das zum Ziel hat, einen sicheren Kanal zwischen zwei Kommunikationsendpunkten herzustellen.

UDP User Datagram Protocol ist ein verbindungsloses und verlustbehaftetes Netzwerkprotokoll.

Anhang

A Anhang 1

A.1 Ergebnisse des User-Single-Modes

In diesem Abschnitt vergleichen wir die Auswirkungen des *Single-User-Mode*. In dem Vergleich betrachten wir nur den *UNIX-Socket* mit TCP und UDP.

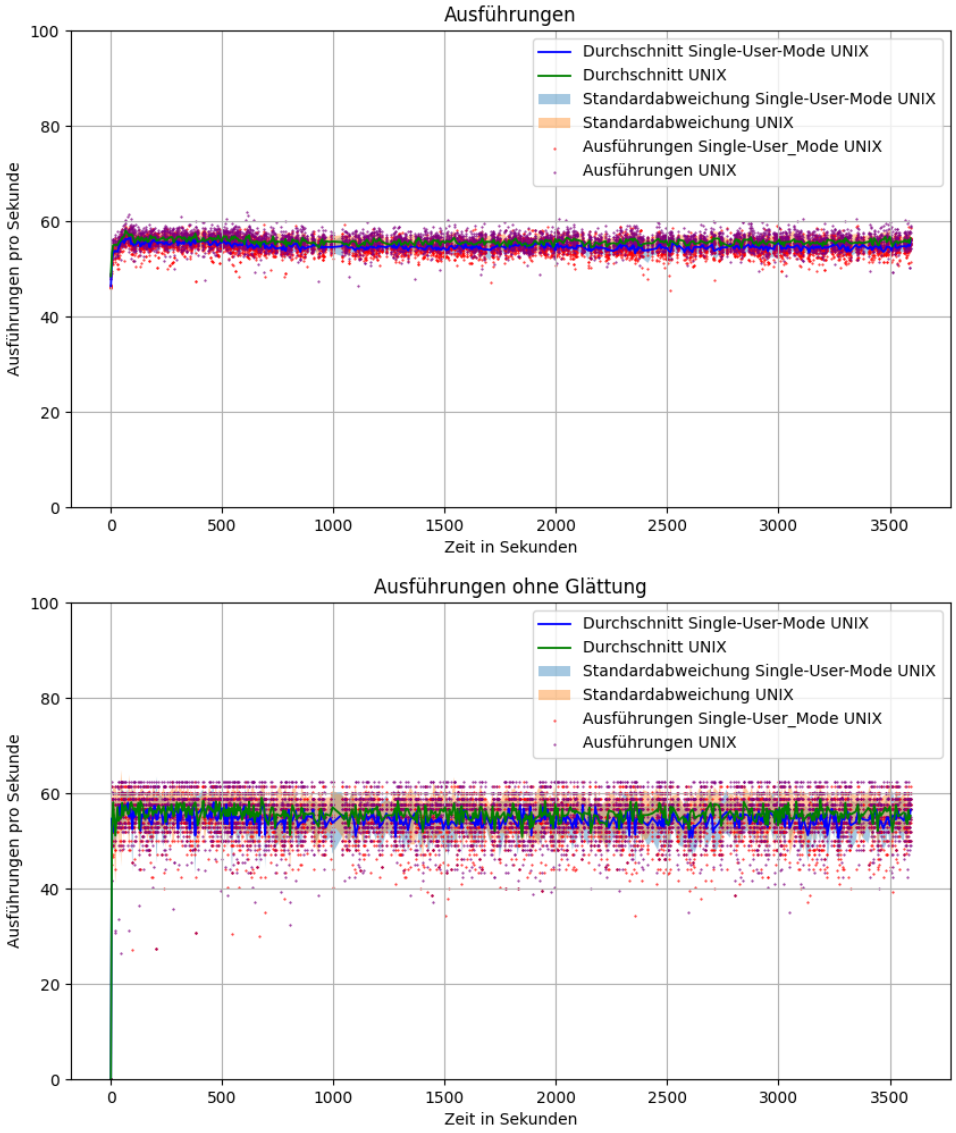

Abbildung A.1: Verleich für TCP mit *UNIX-Socket* zwischen *Single-User-Mode* und *Multi-User-Mode*.

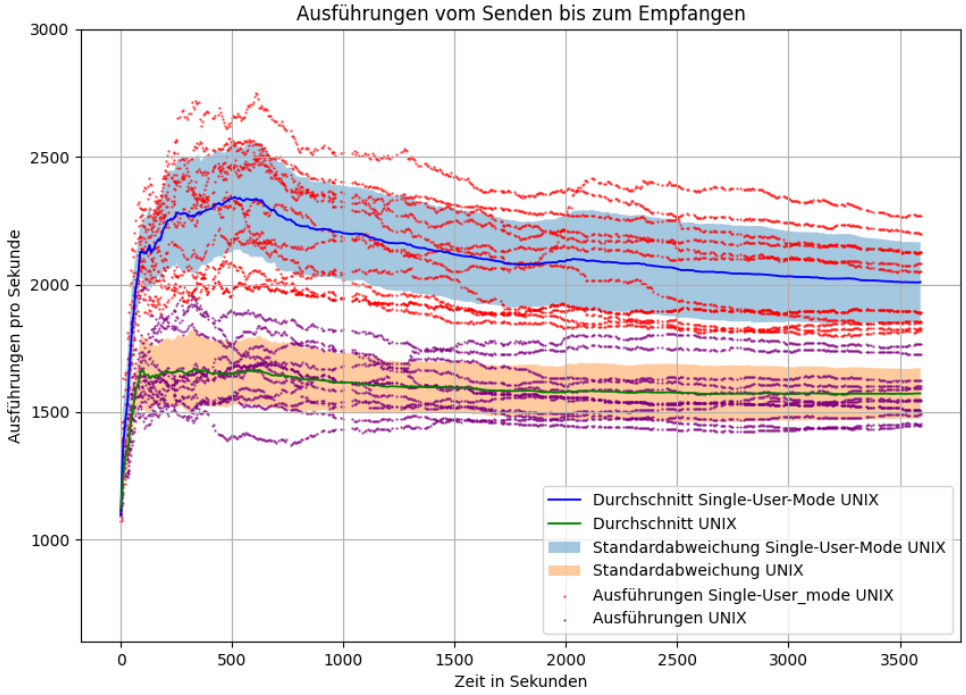

Abbildung A.2: Verleich des Sende-, Bearbeitungs- und Empfangsprozesses für TCP mit *UNIX-Socket* zwischen *Single-User-Mode* und *Multi-User-Mode*.

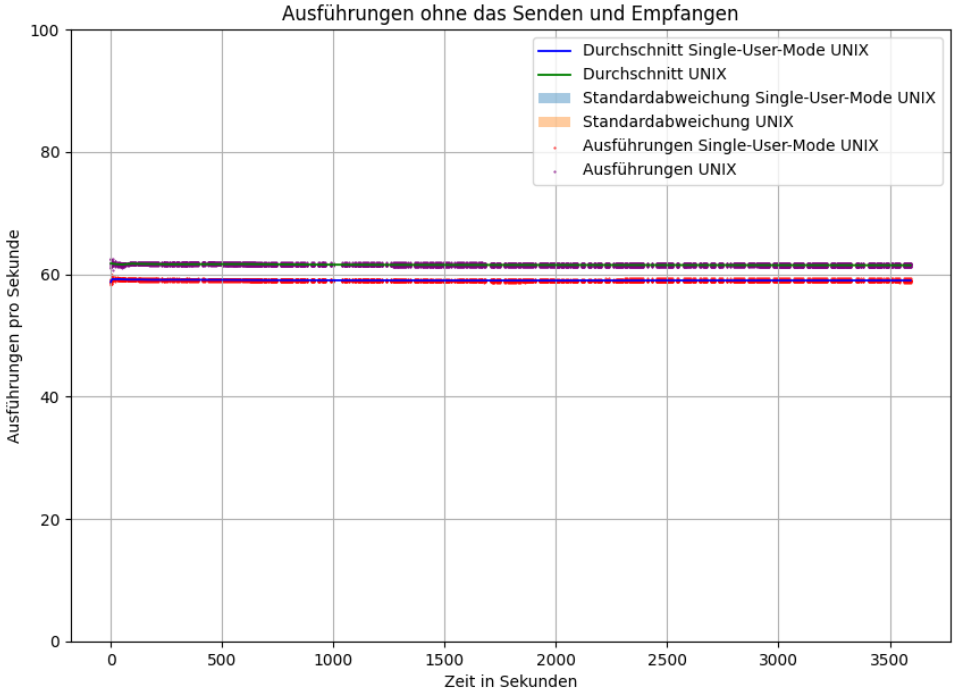

Abbildung A.3: Verleich ohne den Sende-, Bearbeitungs- und Empfangsprozess für TCP mit *UNIX-Socket* zwischen *Single-User-Mode* und *Multi-User-Mode*.

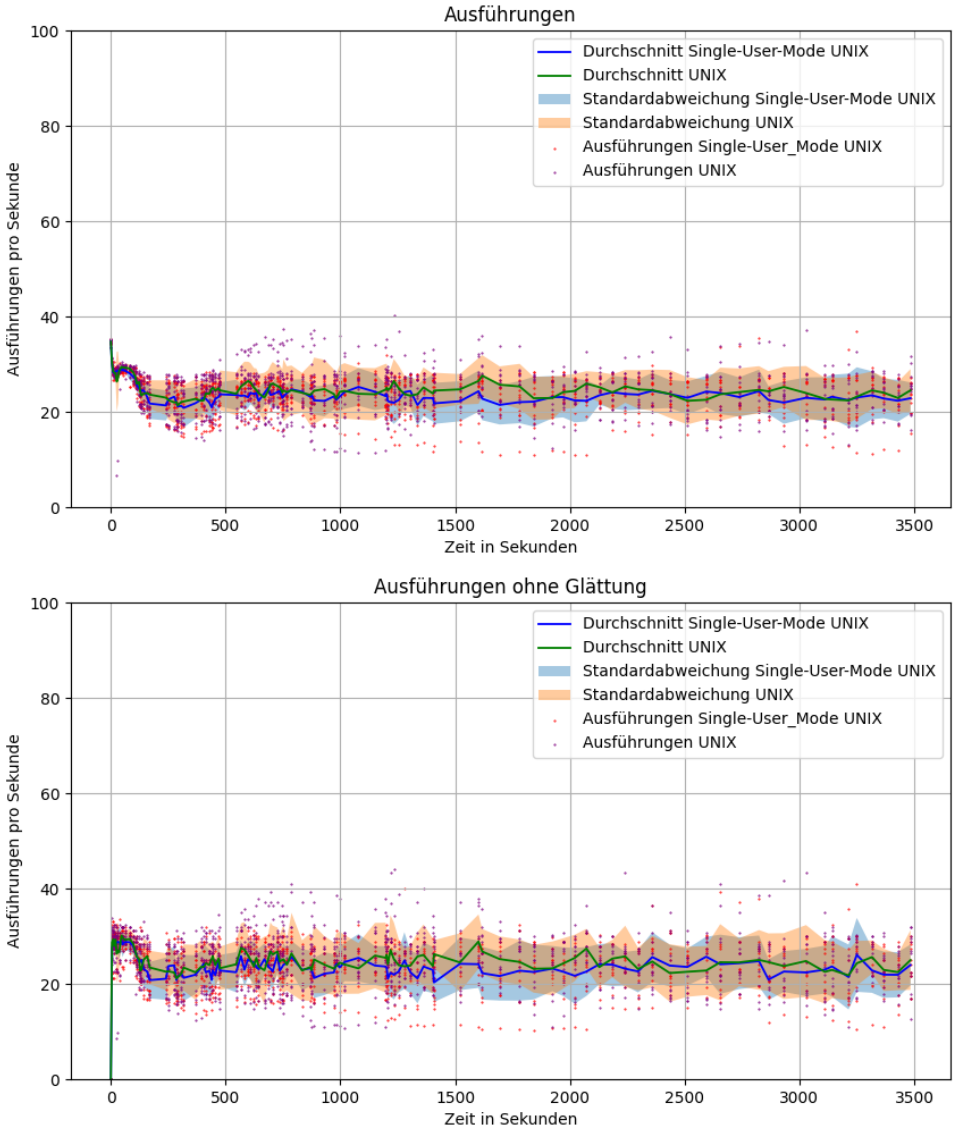

Abbildung A.4: Verleich für UDP mit *UNIX-Socket* zwischen *Single-User-Mode* und *Multi-User-Mode*.

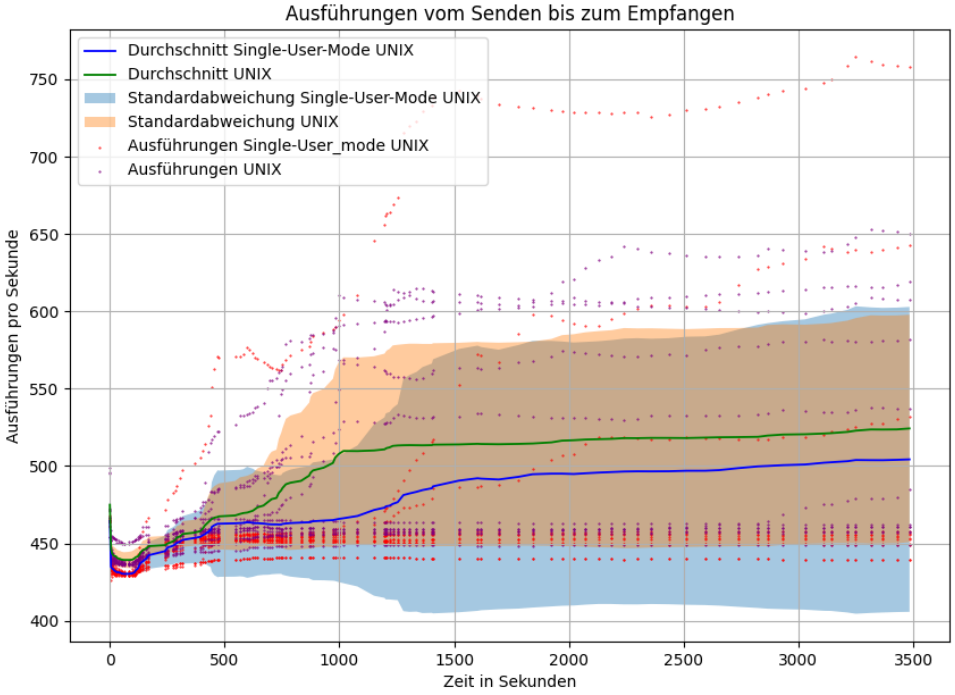

Abbildung A.5: Verleich des Sende-, Bearbeitungs- und Empfangsprozesses für UDP mit *UNIX-Socket* zwischen *Single-User-Mode* und *Multi-User-Mode*.

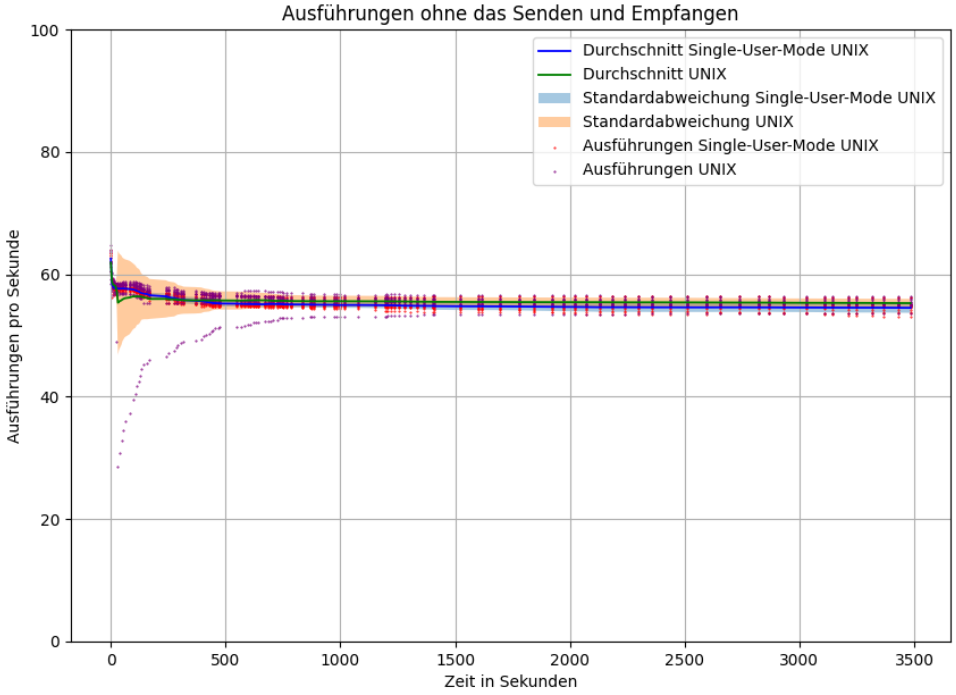

Abbildung A.6: Verleich ohne den Sende-, Bearbeitungs- und Empfangsprozess für UDP mit *UNIX-Socket* zwischen *Single-User-Mode* und *Multi-User-Mode*.

A.2 Ergebnisse mit blockierenden Sockets

In diesem Abschnitt zeigen wir die Ergebnisse von blockierenden *UNIX-Sockets* im Vergleich zu *INET-Sockets*. In dem Abschnitt Auswertung der erhobenen Daten haben wir die Grafiken gezeigt, die nicht blockieren. Diese Grafiken dienen nur zur Information.

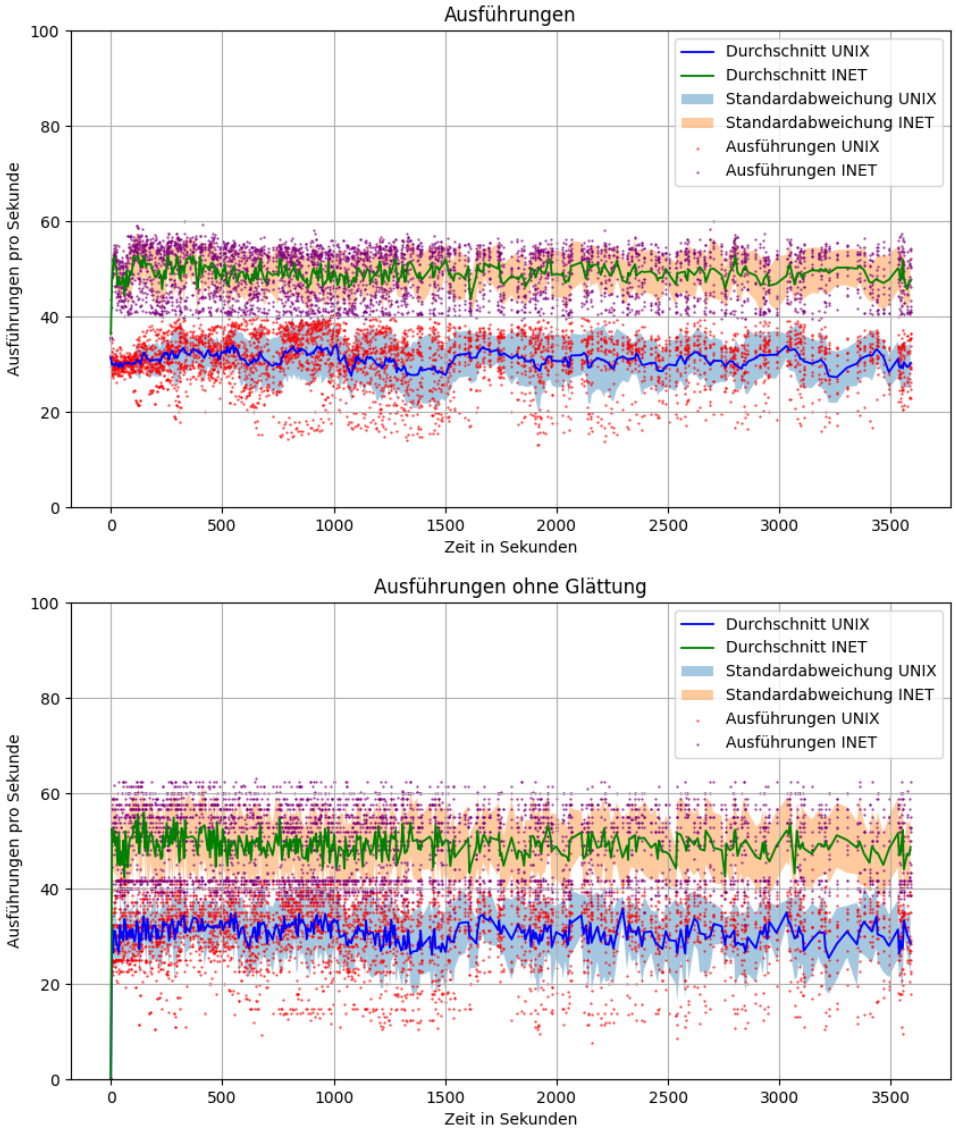

Abbildung A.7: Anzahl an Ausführungen pro Sekunde für TCP und der *UNIX-Socket* ist blockierend.

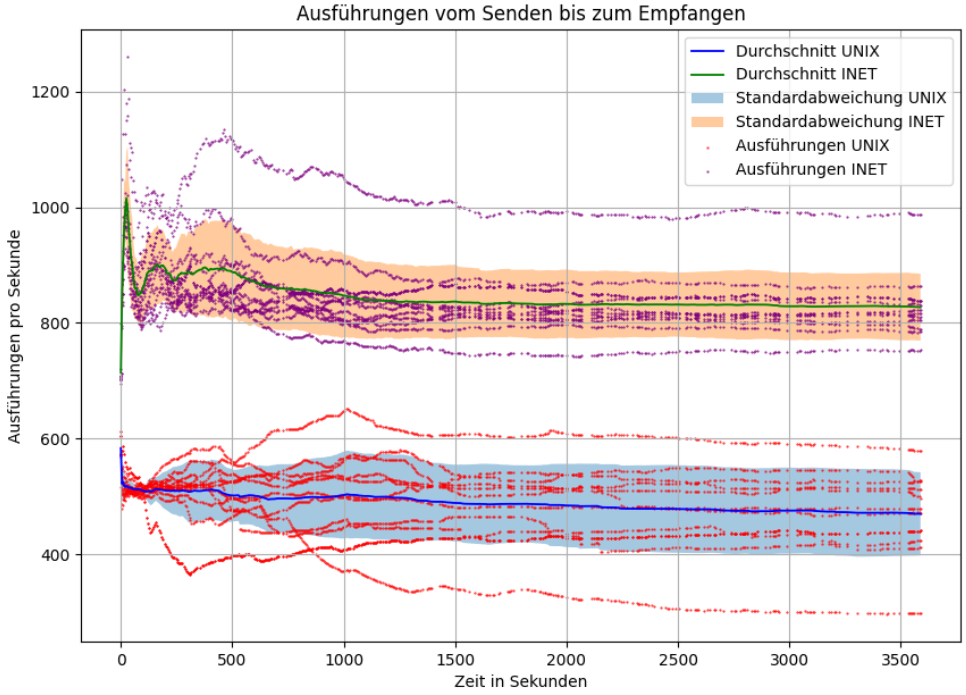

Abbildung A.8: Sende-, Bearbeitungs- und Empfangsprozess für TCP und der *UNIX-Socket* ist blockierend.

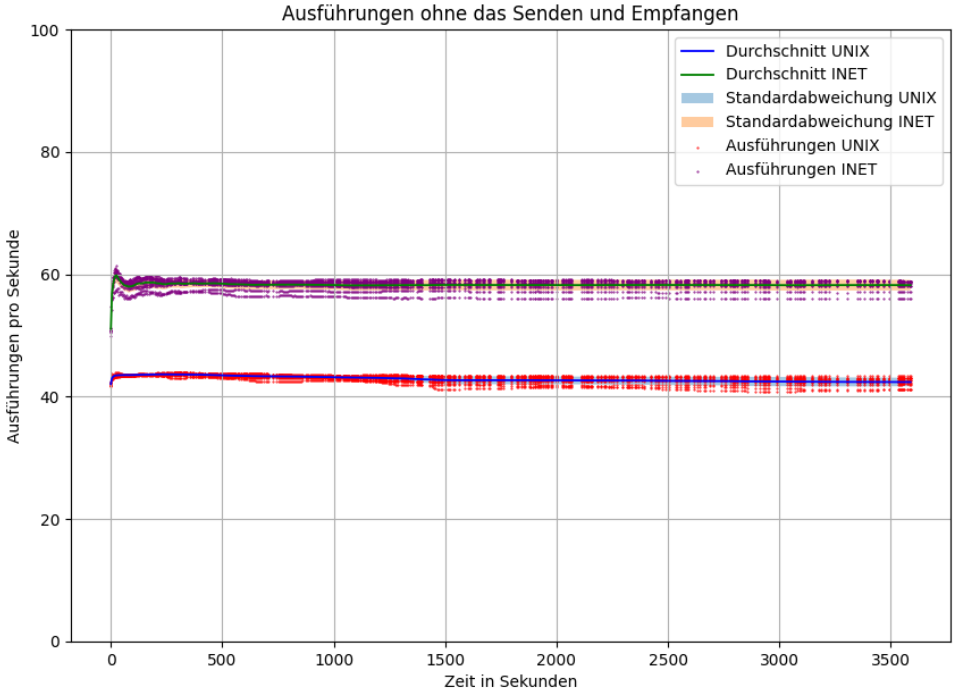

Abbildung A.9: Anzahl an Ausführungen pro Sekunde für TCP ohne den Sende-, Bearbeitungs- und Empfangsprozess und der *UNIX-Socket* ist blockierend.

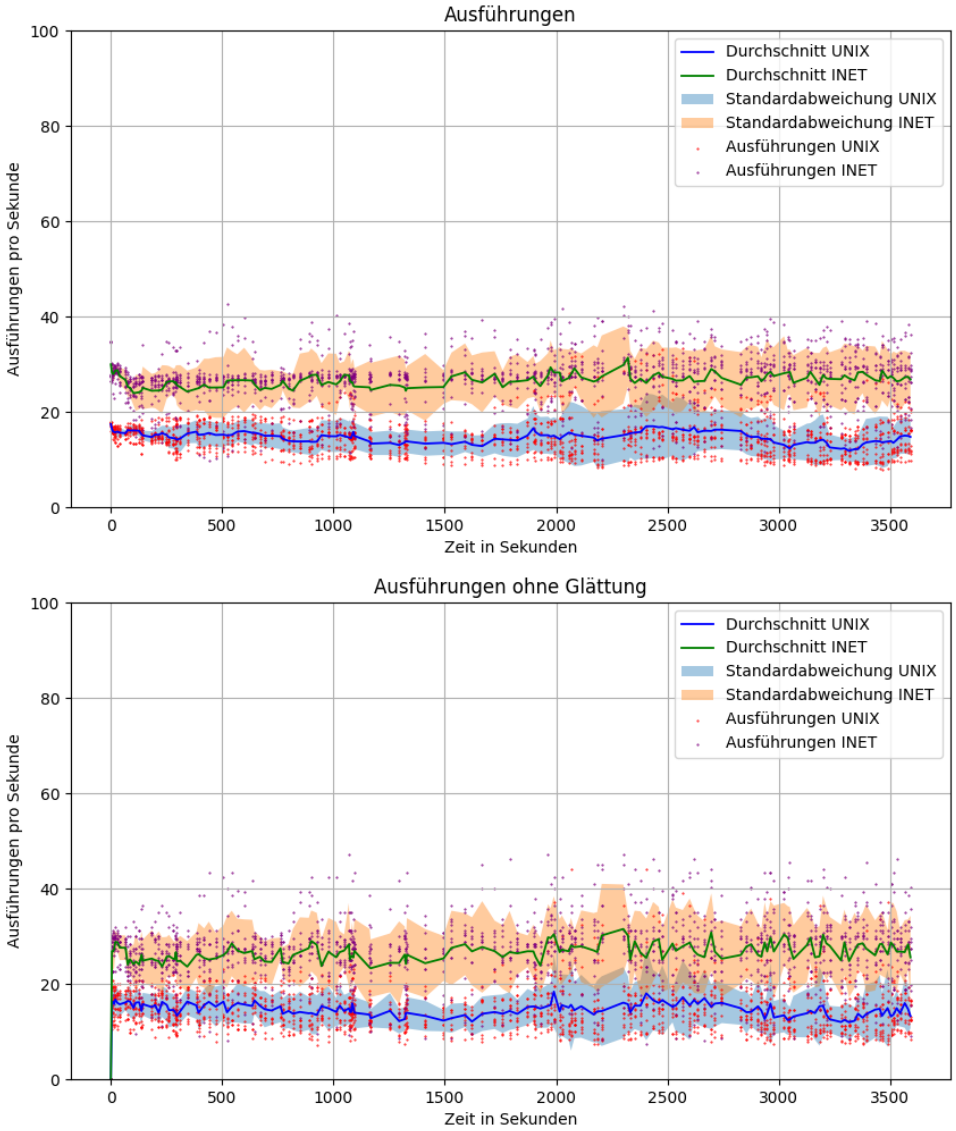

Abbildung A.10: Anzahl an Ausführungen pro Sekunde für UDP und der *UNIX-Socket* ist blockierend.

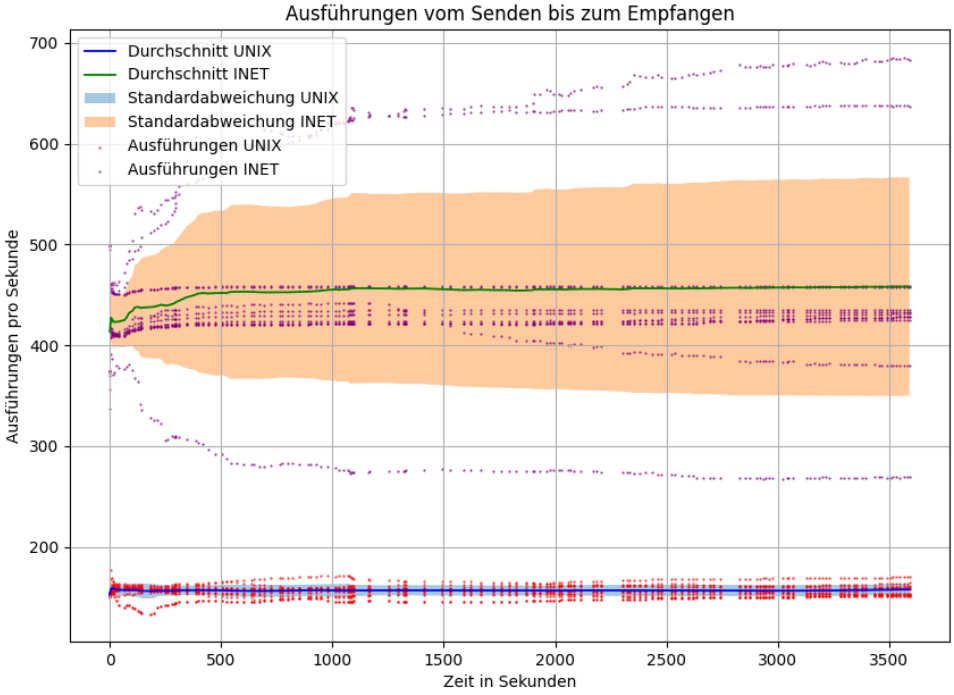

Abbildung A.11: Sende-, Bearbeitungs- und Empfangsprozess für UDP und der *UNIX-Socket* ist blockierend.

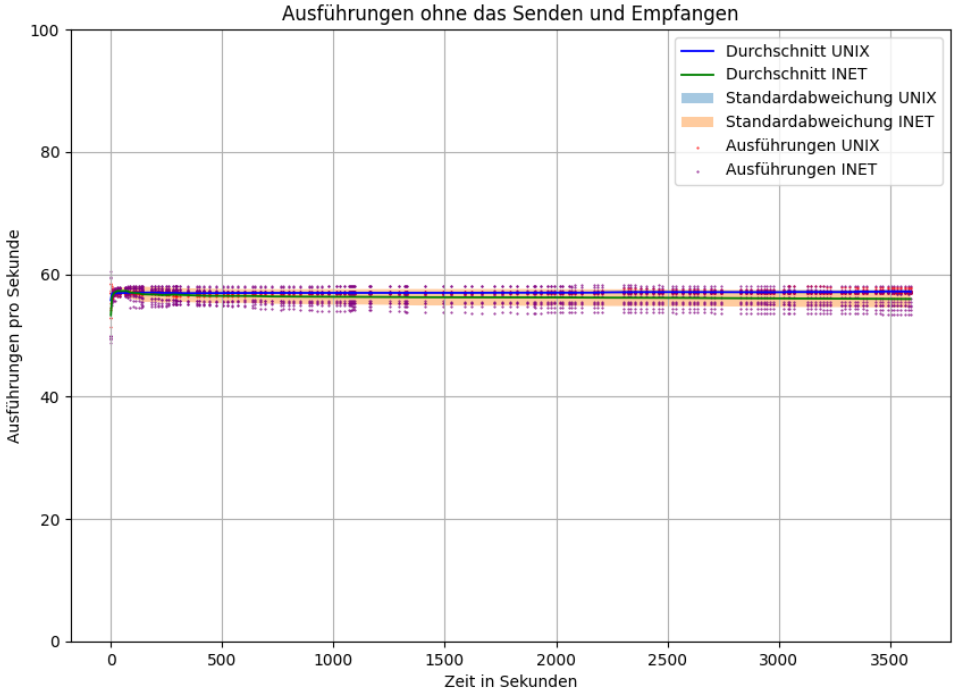

Abbildung A.12: Anzahl an Ausführungen pro Sekunde für UDP ohne den Sende-, Bearbeitungs- und Empfangsprozess und der *UNIX-Socket* ist blockierend.

A.3 Ergebnisse des User-Single-Modes mit der Verwendung von mehreren Prozessorkernen

In diesem Abschnitt zeigen wir die Ergebnisse zwischen TCP und UDP mit dem *UNIX-Socket* im *Single-User-Mode* mit mehreren zugewisenen Prozessorkernen.

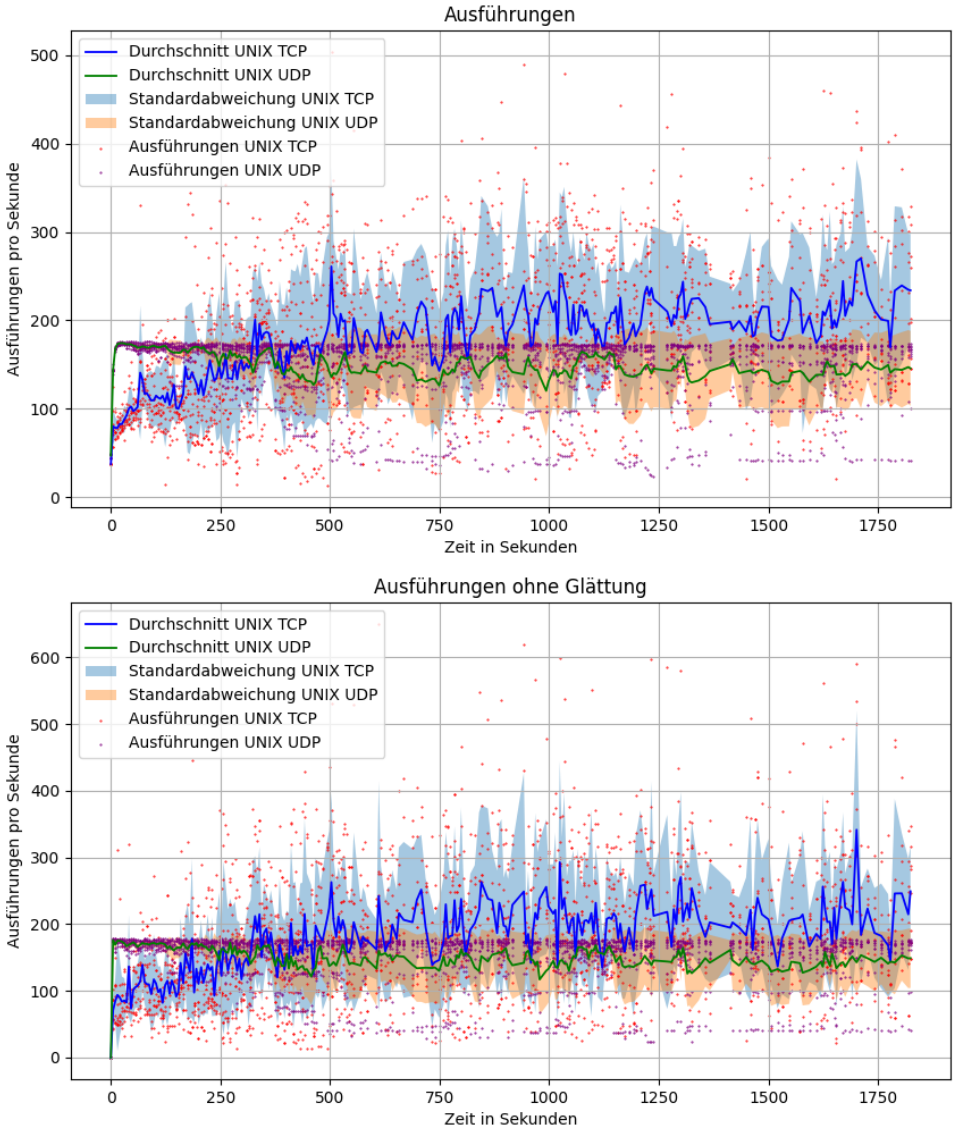

Abbildung A.13: Anzahl an Ausführungen pro Sekunde zwischen TCP und UDP mit dem *UNIX-Socket* im *Single-User-Mode* mit mehreren zugewisenen Prozessorkernen.

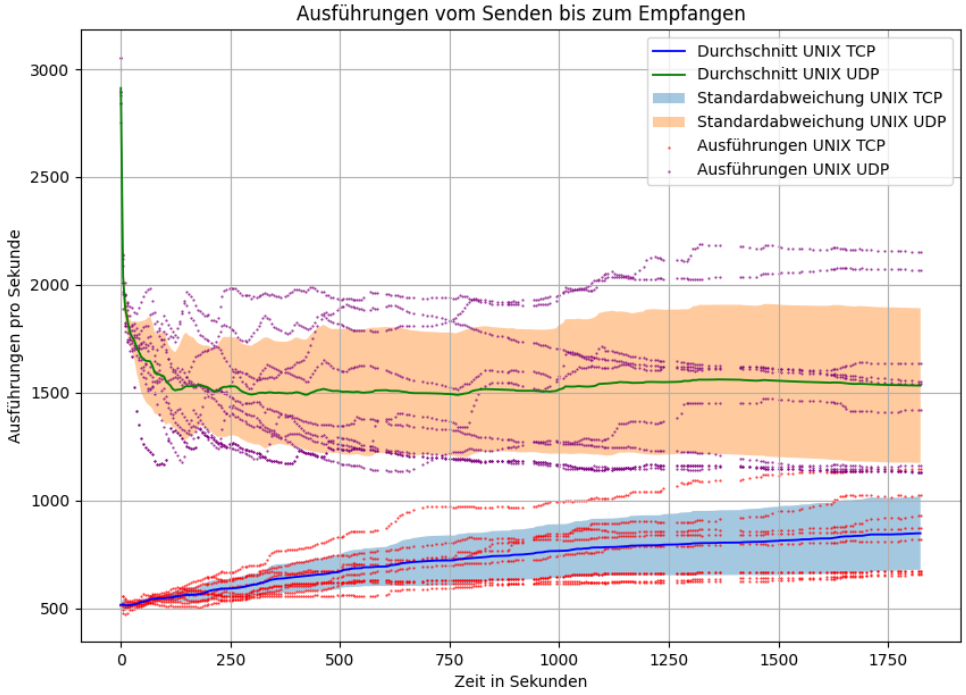

Abbildung A.14: Anzahl an Ausführungen pro Sekunde des Sende-, Bearbeitungs- und Empfangsprozesses zwischen TCP und UDP mit dem *UNIX-Socket* im *Single-User-Mode* mit mehreren zugewisenen Prozessorkernen.

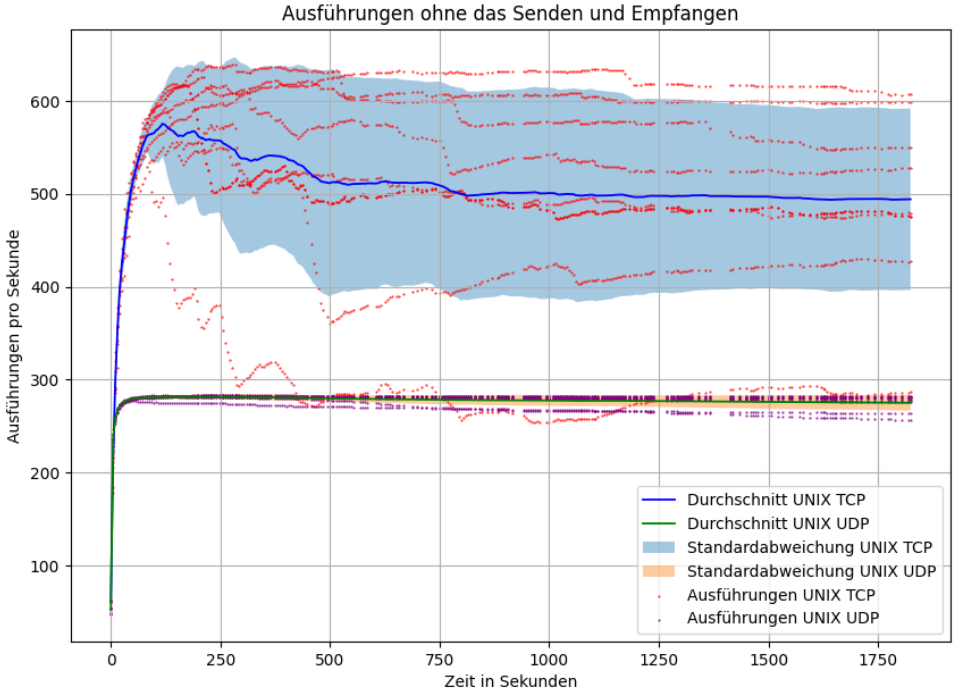

Abbildung A.15: Anzahl an Ausführungen pro Sekunde ohne den Sende-, Bearbeitungs- und Empfangsprozess zwischen TCP und UDP mit dem *UNIX-Socket* im *Single-User-Mode* mit mehreren zugewisenen Prozessorkernen.