Karlsruhe Institute of Technology

Masterarbeit

Überprüfung von Schutzzielen der Informationssicherheit durch Aufzeichnung und Wiedergabe virtueller Maschinen

Sommersemester 2022

von

Marco Liebel

Institut für Theoretische Informatik
Kompetenzzentrum für angewandte Sicherheitstechnologie
Fakultät für Informatik
Karlsruher Institut für Technologie

Prüfer: Prof. Dr. Jörn Müller-Quade
Betreuer: M.Sc. Felix Dörre

14.02.2022 - 15.08.2022

Copyright © ITI und Verfasser 2022

Institut für Theoretische Informatik
Fakultät für Informatik
Karlsruher Institut für Technologie
Am Fasanengarten 5
76131 Karlsruhe

Eidesstattliche Erklärung

Ich versichere, dass ich die vorstehende Arbeit selbständig ohne fremde Hilfe gefertigt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und alle Zitate kenntlich gemacht habe. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Karlsruhe, den 15. August 2022

....
Marco Liebel

Aus Gründen der besseren Lesbarkeit wird auf die gleichzeitige Verwendung der Sprachformen männlich, weiblich und divers (m/w/d) verzichtet. Sämtliche Personenbezeichnungen gelten gleichermaßen für alle Geschlechter.

Zusammenfassung

Auf Grund der weiterhin voranschreitenden Digitalisierung findet immer mehr Kommunikation über Computernetzwerke statt. Bei diesen Datenübertragungen ist es wichtig, dass sie sicher vonstattengehen. Insbesondere wenn sensible Daten übertragen werden, wie zum Beispiel Bankdaten oder Firmengeheimnisse. Deshalb müssen geeignete Verfahren verwendet werden, um die Datenübertragungen abzusichern. Ein Verfahren zur Absicherung von Datenübertragungen ist die Verwendung von Record-/Replay-Funktionalität virtueller Maschinen, um damit Angreifer zu erkennen. Dabei wird eine virtuelle Maschine aufgezeichnet und parallel dazu, mit geringer Zeitverzögerung, wieder abgespielt. Durch den Vergleich ausgehender Netzwerkpakete werden korrumpierte Systeme während der Ausführung dieses System erkannt. Das Ziel dieser Arbeit ist es, ein solches System auf Basis der Open Source Software QEMU zu realisieren. Hierfür mussten, neben Erweiterungen an QEMU, zusätzlich Erweiterungen am Linux-Kernel durchgeführt werden. Dabei handelt es sich ebenfalls um Open Source Software, die innerhalb der virtuellen Maschinen eingesetzt wird. In Kombination mit weiterer, eigens für diese Arbeit geschriebener Software, konnte ein Aufbau erstellt werden, der eine Überprüfung der Sicherheitsziele Integrity und Confidentiality ermöglicht. Dieser wurde dafür verwendet, um eine SSH-Verbindung zu einem Server aufzubauen, der zuerst aufgezeichnet und im Anschluss erfolgreich wieder abgespielt wurde. Der prototypische Aufbau zeigt, dass eine Umsetzung nicht nur möglich ist, sondern auch für die Verwendung bei nicht-trivialen Aufgaben, unter Einsatz kryptografischer Verfahren, geeignet ist.

Abstract

Due to the ongoing digitization, there is more and more communication through computer networks taking place. With these data transfers, it is important that they are secure. Especially when sensitive data is transmitted, such as bank details or company secrets. Therefore, suitable methods must be used to secure data transfers. A theoretical procedure for securing data transmissions is the use of record/replay functionality of virtual machines to detect attackers. A virtual machine is recorded and parallel to it, with a small time delay, played again. By comparing outgoing network packets, corrupted systems are identified. The aim of this work is to develop such a system using the open source software QEMU. In addition to extensions to QEMU, this also required extensions to the Linux kernel. This is also open source software and is used within the virtual machines themselves. In combination with other software specially written for this work, a system was created that allows checking the security goals Integrity and Confidentiality. This was used to establish an SSH connection to a server, which was first recorded and then successfully replayed. The prototypical system shows that an implementation is not only possible, but also suitable for non-trivial tasks using cryptographic methods.

Inhaltsverzeichnis

1 Einleitung **1**

2 Grundlagen **2**
2.1 Linux-Kernel 2
2.1.1 Konfiguration, Kompilation und Installation 2
2.1.2 Die Random und HW-Random Komponenten 5
2.1.3 TUN-, TAP- und Bridge-Devices 7
2.2 QEMU 8
2.2.1 Konfiguration, Kompilation und Debugging 9
2.2.2 Replay 10
2.2.3 Networking 13

3 Problemstellung **16**

4 Umsetzung **19**
4.1 Schritt 1: Live abspielen über einen Ereignisstrom 19
4.2 Schritt 2: Zufall der VMs festlegen 21
4.3 Schritt 3: Überprüfung ausgehender Pakete 25
4.4 Vollständiger Aufbau 30

5 Zusammenfassung und Ausblick **33**

Literatur **35**

Abkürzungen

Abb. Abbildung

Lst. Listing

Kap. Kapitel

CRNG Cryptographic-Random-Number-Generator

DHCP Dynamic Host Configuration Protocol

FIFO First in, First Out

IP Internet Protokoll

MITM Man-in-the-Middle

SSH Secure Shell

TCG Tiny Code Generator

TCG Ops TCG Operations

TCP Transmission Control Protocol

UDP User Datagram Protocol

VM virtuelle Maschinen

Abbildungsverzeichnis

1 Aufbau des Linux Git Repositorys 2
2 Ausschnitt möglicher Configuration-Targets 2
3 Hauptmenü des Konfigurationsmenüs 3
4 TUN-, TAP- und Bridge-Devices im Zusammenspiel 8
5 Schematische Darstellung der Arbeitsweise des TCG 10
6 QEMU-Kommandozeilen-Option zum Konfigurieren des Instruction Counting . . 11
7 Inhalt der Replay-Komponente 11
8 Ausschnitt aus einer Aufzeichnung 13
9 Network Backends in Verbindung mit Virtual Network Devices 14
10 Network Filters im Zusammenspiel mit Network Backends und Virtual Network Devices 15
11 System das die Ausgangssituation dieser Arbeit darstellt 16
12 Darstellung des Zielsystems 17
13 Umsetzung des Ereignis-Stroms 19
14 Ausschnitt aus einer Aufzeichnungs-Datei 20
15 Erweiterung des Aufbaus um Zufall auszuhandeln 23
16 Unsicherer Münzwurf mit indirekter Kommunikation 24
17 Sicherer Münzwurf mit indirekter Kommunikation 25
18 Aufbau des Socket-Backend-Protokolls 26
19 Gesamtsystem mit allen Komponenten 31

Listings

1 Interfaces zum Hinzufügen von Zufall 5
2 add_hwgenerator_randomness() 5
3 hwrng_fillfn() 6
4 Erstellen eines Bridge- und eines TAP-Devices 8
5 Icount im TimersState struct 11
6 Enums die Ereignisse definieren 11
7 Kommandozeile zum Erstellen von Network Backends und Virtual Network Devices 14
8 QEMU-Optionen zur Aktivierung der Replay-Funktionalität 19
9 Ausschnitt aus der Replay-Komponente von QEMU 20
10 Erweiterung zum Schreiben des Headers am Anfang einer Aufzeichnung 21
11 Erstellen eines Hardware-Zufallszahlengenerators mit QEMU 22
12 Netzwerk Konfiguration beider QEMU-Instanzen 25
13 Konfiguration des TAP-Devices für den Komparator 27
14 Standard Replay Filter 28
15 Erweiterter Replay Filter 28
16 Funktionen zur Handhabung der vom Komparator gesendeten Pakete 29
17 Injizieren der vom Komparator gesendeten Pakete 30

1 Einleitung

Auf Grund der immer steigenden Anzahl an Computern durch die Digitalisierung steigt ebenfalls die Anzahl der Computersysteme die mit Computernetzwerken verbunden sind, insbesondere dem Internet. Dies hat Einfluss auf die Datenübertragungen, die über solche Computernetzwerke stattfinden, da damit auch sensible Daten übertragen werden. Beispiele für solche Daten sind Bankdaten, welche beim Online-Banking übermittelt werden oder Firmengeheimnisse, auf die Mitarbeiter bei einer Verbindung mit deren Firmennetzwerk zugreifen können. Für diese Art von Daten ist es besonders wichtig, dass sie über sichere Kanäle übertragen werden, da im Falle eines Angriff große finanzielle Schäden entstehen können. Eine Möglichkeit Schutz zu gewährleisten ist die Überprüfung der Schutzziele Confidentiality, Integrity und Availability, die in der Informationssicherheit gebräuchlich sind. Confidentiality und Integrity können beispielsweise durch Verschlüsselungs- und Signaturverfahren erreicht werden. Hat ein Angreifer ein System jedoch bereits korrumpiert, zum Beispiel den Server für das Online-Banking einer Bank, besteht die Gefahr, das die verwendeten kryptografischen Verfahren nicht ausreichen, um die Sicherheitsziele zu gewährleisten. Ein mögliches Szenario ist, dass ein Angreifer den Zufall, der für kryptografische Verfahren verwendet wird, kontrolliert. Dadurch kann dieser so gewählt werden, dass Verschlüsselungen nicht mehr sicher sind und gebrochen werden können. Um dieser Gefahr entgegenzuwirken soll im Zuge dieser Arbeit ein System entwickelt werden, das unter Verwendung der Record-/Replay-Funktionalität von QEMU die Sicherheitsziele Confidentiality und Integrity gewährleistet. Bisherige Ansätze versuchen konkrete Angriffe zu erkennen und eine virtuelle Maschinen (VM) parallel zur Aufzeichnung oder im Anschluss daran zu analysieren.[17][18][28] Der hier vorgestellte Ansatz soll nicht die VMs selbst analysieren, sondern ausschließlich deren Netzwerkkommunikation. Dafür werden Änderungen an QEMU und am Linux-Kernel durchgeführt um ein Gesamtsystem zu erstellen, das aus mehreren VMs und einem Komparator besteht. Dieser Komparator erlaubt die Übertragung von Netzwerkpaketen nur dann, wenn die von den VMs gesendeten Pakete Bit für Bit gleich sind. Zusätzlich wird angenommen, dass mindestens eine VM und der Komparator nicht durch einen Angreifer korrumpiert sind. Somit sind nur Übertragungen unveränderter Netzwerkpakete möglich und die Schutzziele können gewährleistet werden.

Im zweiten Kapitel dieser Arbeit werden Grundlagen vorgestellt, die für die Umsetzung des Systems relevant sind, das im Zuge dieser Arbeit entwickelt wird. Dies sind einerseits Kenntnisse über den Linux-Kernel, die das Kompilieren des Kernels, Informationen über die interne Zufalls-Komponente und die Konfiguration von TAP-Devices beinhalten. Andererseits umfassen diese Kenntnisse einen Überblick über QEMU, das als Software zur Virtualisierung zum Einsatz kommt. Neben Informationen über die Kompilation wird hier der Aufbau der Replay-Komponente und das interne Networking von QEMU vorgestellt.

Kapitel drei befasst sich mit dem Problem, das durch diese Arbeit gelöst werden soll. Dazu wird ausgehend von einem grundlegenden Aufbau, der üblicherweise für VMs eingesetzt wird, ein konzeptioneller Aufbau entwickelt, der die Überprüfung der Sicherheitsziele Integrity und Confidentiality ermöglicht.

Das vierte Kapitel stellt schrittweise alle Änderungen vor, die für die Realisierung des konzeptionellen Aufbaus nötig sind und beschreibt diese ausführlich. Hierfür wird QEMU zunächst um Live-Replay-Funktionalität erweitert. Daraufhin folgen Änderungen am Linux-Kernel, um Zufall bereitstellen zu können, der für die Umsetzung dieser Arbeit geeignet ist. Der letzte Änderungsschritt erweitert die Netzwerk-Komponente von QEMU, um Pakete in aufzeichnende VMs injizieren zu können. Abschließend wird das Gesamtsystem mit allen zuvor eingebauten Änderungen vorgestellt und dabei auch das Szenario, mit dem dieser Aufbau getestet wurde.

Aller Änderungen an den Open Source Projekten, die im Laufe dieser Arbeit entstanden sind, sowie die eigens für diese Arbeit erstellten Werkzeuge sind der Arbeit als Zip-Archiv beigefügt.

```
arch    COPYING  Documentation  include  Kbuild   lib          Makefile  README   security  usr
block   CREDITS  drivers        init     Kconfig  LICENSES     mm        samples  sound     virt
certs   crypto   fs             ipc      kernel   MAINTAINERS  net       scripts  tools
```

Abbildung 1: Aufbau des Linux Git Repositorys

```
Configuration targets:
  config          - Update current config utilising a line-oriented program
  nconfig         - Update current config utilising a ncurses menu based program
  menuconfig      - Update current config utilising a menu based program
  xconfig         - Update current config utilising a Qt based front-end
  gconfig         - Update current config utilising a GTK+ based front-end
  oldconfig       - Update current config utilising a provided .config as base
```

Abbildung 2: Ausschnitt möglicher Configuration-Targets

2 Grundlagen

Im folgenden Kapitel wird Wissen vermittelt, das für die späteren Teile dieser Arbeit wichtig ist. Dabei handelt es sich im Wesentlichen um Informationen zum Umgang mit den beiden Software-Projekten Linux und QEMU. Dies beinhaltet sowohl Kenntnisse zur Nutzung beider Softwares als auch Kenntnisse, die deren Verständnis dienen und für das Arbeiten am jeweiligen Projekt erforderlich sind. Des Weiteren werden mehrere Annahmen bezüglich des Kenntnisstandes der Leser dieser Arbeit getroffen. An dieser Stelle wir vorausgesetzt, dass bekannt ist worum es sich bei den genannten Projekten handelt. Darin enthalten sind auch grundlegende Kenntnisse über Betriebssysteme und Virtualisierung. Zudem ist der Umgang mit der Linux-Kommandozeile und verschiedenen Werkzeugen wie make, git und gdb Voraussetzung, da Linux selbst als Plattform für die spätere Umsetzung verwendet wird.

2.1 Linux-Kernel

Damit am Linux-Kernel gearbeitet werden kann, muss zunächst eines von mehreren GitHub-Repositorys ausgewählt werden. Die unter [5] verfügbaren Repositorys unterscheiden sich darin, dass sie als zentrale Anlaufstelle für bestimmte Unterkomponenten oder Releases dienen. Beispielsweise das Mainline-Repository unter [11] oder das Stable-Repository unter [21]. Diese Unterscheidung ist relevant, da für die Umsetzung im späteren Verlauf dieser Arbeit die Version 5.15.30 des Stable Trees zum Einsatz kommt.

Unabhängig davon welcher Tree für die Arbeit am Linux-Kernel ausgewählt wird, ist die grobe Struktur immer gleich. Abbildung (Abb.) 1 zeigt beispielsweise das Hauptverzeichnis des Stable Trees. Neben verschiedenen Dateien, die unter anderem für die Konfiguration des Projekts nötig sind, enthält dieses Verzeichnis eine Vielzahl von Unterverzeichnissen. Der Großteil dieser Verzeichnisse gehört zu den Unterkomponenten des Linux-Kernels. Dazu zählen etwa *crypto*, *net* und *sound*. Die für diese Arbeit relevanten Teile sind die Character-Devices *random* und *hwrandom*. Diese befinden sich in der Driver-Komponente und werden im Kapitel 2.1.2 betrachtet.

2.1.1 Konfiguration, Kompilation und Installation

Der Prozess, um einen selbst gebauten Kernel zu erhalten, besteht aus drei Schritten. Diese Schritte sind die Konfiguration, die Kompilation oder das Bauen und die Installation des Kernels. Die Reihenfolge dieser Schritte ist nicht variabel und muss in der genannten Abfolge durchgeführt werden.

Einen ersten Überblick über die Optionen, die beim Build-Prozess zur Verfügung stehen, gibt der Befehl *make help*. Abb. 2 zeigt einen Ausschnitt dieser Optionen, die für die Konfiguration ausgewählt werden können. Der Fokus dieses Abschnitts liegt auf dem Configuration-Target

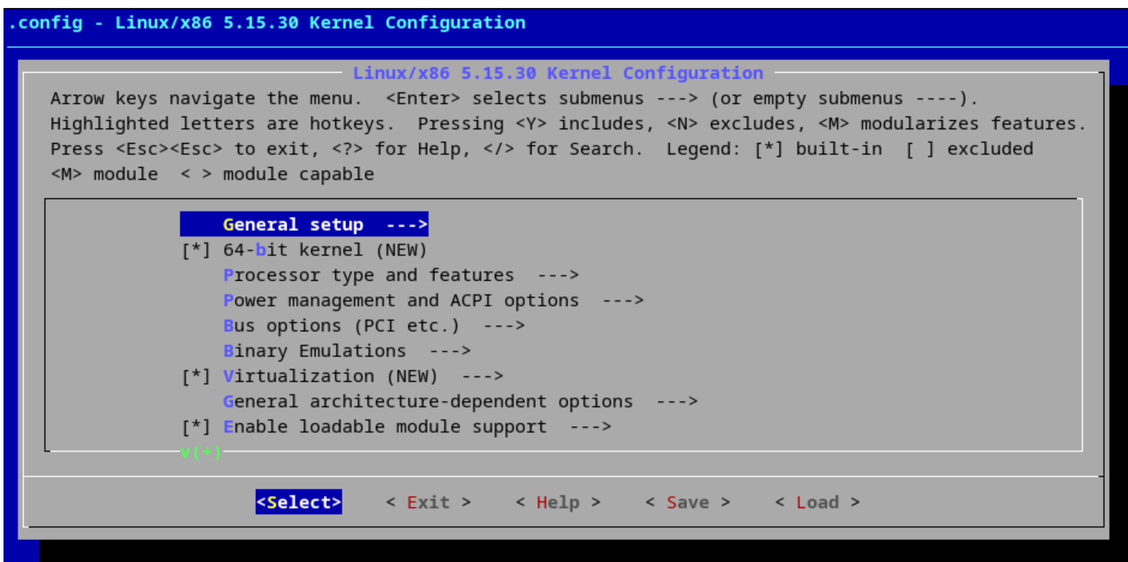

Abbildung 3: Hauptmenü des Konfigurationsmenüs

menuconfig, das in dieser Liste an dritter Stelle steht und mit dem Befehl *make menuconfig* aufgerufen wird. Im Wesentlichen unterscheiden sich die abgebildeten Optionen nur in der Art der Ausgabe. An der Beschreibung, die neben den Configuration-Targets hinter dem Bindestrich folgt, wird dies verdeutlicht. Die Option nconfig beispielsweise verwendet die ncurses-Bibliothek, um eine menübasierte Ausgabe im Terminal zu erzeugen, xconfig hingegen verwendet Qt für die Ausgabe. Die Ausgabe der in dieser Arbeit verwendeten Option ist in 3 abgebildet und zeigt das Hauptmenü. Zu erkennen ist, dass es sich ebenfalls um eine menübasierte Ausgabe im Terminal handelt.

Die tatsächliche Konfiguration des Kernels geschieht über die verschiedenen Einträge im Hauptmenü und dessen Untermenüs. Die Einträge können unter anderem vom Typ String, Boolean oder Integer sein, abhängig von der gewünschten Konfiguration. Nach Abschluss der Konfiguration werden die ausgewählten Konfigurations-Optionen in einer Datei mit dem Namen *.config* gespeichert. Bei der Kompilation der Software im nächsten Schritt wird diese Datei verwendet, um den Kernel entsprechend der gewählten Konfigurations-Optionen zu erstellen. Sollten keine besonderen Anforderungen an die Konfiguration des Kernels bestehen, gibt es die Möglichkeit eine Default-Konfiguration zu wählen. Dafür genügt ein Aufruf des Konfigurationsmenüs ohne existierende Konfigurationsdatei. Unabhängig davon, ob bereits eine Konfigurationsdatei existiert oder Änderungen vorgenommen werden müssen, sollte das Konfigurationsmenü vor der ersten Kompilation aufgerufen werden. Grund dafür ist, dass einige Konfigurationsoptionen automatisch durch das Build-System angepasst werden (z.B. der installierte Compiler).

Zum Starten der Kompilation ist nach Abschluss der Konfiguration lediglich ein Aufruf des Befehls *make* nötig. Je nach Art der Konfiguration kann dieser Schritt mehrere Stunden in Anspruch nehmen. Da es in dieser Arbeit keine besonderen Anforderungen an die Konfiguration des Kernels gibt, wird die Default-Konfiguration mit wenigen, unwesentlichen Anpassungen verwendet.

Der dritte und letzte Schritt, die Installation des Kernels, besteht wiederum aus zwei[1] getrennten Schritten. Zuerst werden mit dem Befehl *make modules_install* die Kernel-Module in die für sie vorgesehenen Verzeichnisse kopiert. Im Anschluss daran werden mit dem Befehl *make install* unter anderem das Kernel-Image, das Initramfs und die Bootloader-Konfiguration in das boot-Verzeichnis kopiert. Sobald dieser Schritt abgeschlossen ist, kann der selbst kompilierte Kernel nach einem Neustart über das Bootloader-Menü ausgewählt und verwendet werden. Worum es sich bei Kernel-Modulen und dem Initramfs handelt, wird im weiteren Verlauf dieses

[1] Tatsächlich besteht der zweite dieser Schritte ebenfalls aus mehreren Schritten, allerdings werden diese Schritte auf x86_64-Systemen in einem einzelnen Befehl gekapselt.

Kapitels vorgestellt.

Zum Abschluss dieses Abschnitts sollen noch zwei der Cleaning-Targets erwähnt werden: *clean* und *mrproper*. Genau wie bei den Configuration-Targets steht mit *make help* eine Kurzbeschreibung beider Targets zur Verfügung. Das Target *clean* entspricht dem in Projekten die make verwenden üblichen Target. Hiermit werden alle Dateien entfernt, die während einer Kompilation entstanden sind. Etwa die Kernel-Module oder das Kernel-Image. Das *mrproper*-Target dahingegen entfernt alle generierten Dateien. Dazu gehört auch die während der Konfiguration entstehende Konfigurationsdatei, was gegebenenfalls zu zusätzlichem Aufwand führen kann, falls diese unbeabsichtigt gelöscht wird, da erneut eine Konfiguration durchgeführt werden muss. Was nicht entfernt wird sind die während der Installation kopierten Datei. Ein bereits installierter Kernel ist also nach einem Aufruf von *mrproper* noch vorhanden.

Kernel-Module und Initramfs

Um das Initramfs zu verstehen ist zuerst eine Auseinandersetzung mit Kernel-Modulen hilfreich, da diese einen der Hauptgründe für die Existenz des Initramfs darstellen.

Im Grunde handelt es sich bei Kernel-Modulen um Software die den Kernel erweitert und zu dessen Laufzeit eingebunden oder entfernt werden kann. Der Unterschied zu normaler Software, wie zum Beispiel einem Webbrowser oder einem E-Mail-Programm, liegt darin, dass normale Software im User-Space angesiedelt ist, Kernel-Module hingegen im Kernel-Space. Dies hat Auswirkungen auf die Berechtigungen dieser Programme, da Software im User-Space sich im unprivilegierten Modus und im Kernel-Space im privilegierten Modus befindet[2]. Kernel-Module haben dadurch die gleichen Berechtigungen wie der Kernel selbst, wodurch die Erweiterung des Kernels selbst möglich ist. Für User-Space-Anwendungen hingegen ist die einzige Möglichkeit privilegierte Operationen durchzuführen bzw. mit dem Kernel in Verbindung zu treten über System Calls. Typische Kernel-Module sind beispielsweise Gerätetreiber oder Dateisysteme.

Gerätetreiber und Dateisysteme eignen sich zudem, um das Initramfs zu verstehen. Der Name selbst gibt schon einen ersten Hinweis darüber was das Initramfs ist: Ein Dateisystem das beim Systemstart in den Arbeitsspeicher geladen wird. Ein Grund für dieses Vorgehen ist, dass der Kernel ohne Initramfs, in bestimmten Situationen, nicht in der Lage ist Kernel-Module zu laden. Angenommen der primäre Datenträger eines Systems ist mit dem ext4-Dateisystem formatiert und die Implementierung dieses liegt als Kernel-Modul auf dem Datenträger. In diesem Fall kann der Kernel nicht ohne weiteres auf die Implementierung zugreifen, da der Datenträger mit der Implementierung gelesen werden muss, die sich auf dem Datenträger selbst befindet. Ein weiteres Beispiel ist der Gerätetreiber für den Datenträger. Liegt dieser als Modul vor, müsste dieser vom Datenträger gelesen werden, für den er selbst die Funktionalität implementiert, um auf die Hardware zuzugreifen.

Die allgemeine Funktion des Initramfs ist somit die Bereitstellung eines Dateisystems, um Kernel-Module oder sonstige Software während des Systemstarts ausführen zu können. Ein weiterer wichtiger Punkt ist jedoch, dass ein Initramfs nicht erforderlich ist. Es ist ebenso möglich kein Initramfs zu verwenden und benötigte Software im Kernel selbst zu integrieren. Der wesentliche Vorteil den die Verwendung eines Initramfs mit sich bringt ist Flexibilität, da Gerätetreiber beispielsweise als Kernel-Modul bereitgestellt werden können und nicht fest in den Kernel mit integriert werden müssen. Somit muss ein Kernel nur einmal kompiliert werden und kann für unterschiedliche Hardware eingesetzt werden.

Ausführliche Informationen zum Kernel-Build-System sind in der offiziellen Dokumentation unter [3] zu finden. Eine detaillierte Einführung in den Build-Prozess gibt es zudem unter [1]. Dort sind alle in Kapitel 2.1.1 vorgestellten Informationen ebenfalls enthalten.

[2] Mit un/privilegiertem Modus sind hier die durch Hardware implementierte Sicherheitsmodi gemeint, worauf in dieser Arbeit nicht weiter eingegangen wird. Bei x86 beispielsweise Ring 0-3.

2.1.2 Die Random und HW-Random Komponenten

Zur Durchführung kryptografischer Operationen werden häufig Zufallszahlen benötigt. Zum Beispiel für Schlüssel, Salts bei der Speicherung von Passwörtern oder Nonces bei Kommunikationsprotokollen. Zwei Möglichkeiten Zufallszahlen zu generieren bzw. zu erhalten sind das Erfassen von nicht-deterministischen Ereignissen und die Verwendung von Zufallszahlengeneratoren die in Hardware implementiert sind. Bei der ersten Möglichkeit kann beispielsweise die Zugriffszeit auf Datenträger oder die Verzögerungszeit bei Netzwerkkommunikation verwendet werden. Bei der zweiten Möglichkeit wird spezielle Hardware zur Generierung von Zufallszahlen verwendet. Diese kann Teil eines Computers selbst sein oder als externe Hardware an einen Computer angeschlossen werden. Zwei konkrete Beispiele sind Intel-Prozessoren ab der Ivy Bridge Generation, die einen Zufallszahlengenerator integrieren, auf den mit der Instruktion *RDRAND* zugegriffen werden kann und externe Generatoren die über eine USB-Schnittstelle angeschlossen werden.[10][9]

```
/* include/linux/random.h */
void add_device_randomness(const void *buf, unsigned int size);
void add_input_randomness(unsigned int type, unsigned int code,
    unsigned int value);
void add_interrupt_randomness(int irq, int irq_flags);
void add_disk_randomness(struct gendisk *disk);

/* include/linux/hw_random.h */
void add_hwgenerator_randomness(const char *buffer, size_t count,
    size_t entropy);
```

Listing 1: Interfaces zum Hinzufügen von Zufall

Der Linux-Kernel setzt die zuvor genannten Möglichkeiten in den beiden Komponenten *Random* und *HW-Random* um. Die *Random*-Komponente übernimmt die Verwaltung des Zufalls, was sowohl die Bereitstellung, als auch die Aggregation von Zufall beinhaltet. Eine Alternative, um Zufall vom Kernel zu erhalten, sind die Character-Special-Files */dev/random* und */dev/urandom*. Daneben gibt es weitere Möglichkeiten Zufall programmatisch zu erhalten, auf die hier nicht weiter eingegangen wird.

Wesentlich für die spätere Umsetzung ist die Aggregation von Zufall. Dies ermöglichen die in Listing (Lst.) 1 abgebildeten Funktionen. Die ersten vier Funktionen sind Teil der *Random*-Komponente selbst und sammeln Zufall anhand nicht-deterministischer Ereignisse, wie dies im vorherigen Abschnitt beschrieben wurde. Die fünfte Funktion ist in der *HW-Random*-Komponente deklariert, die Definition befindet sich jedoch ebenfalls in der *Random*-Komponente. Anders als die vier Ereignis-basierten Funktionen wird von *add_hwgenerator_randomness* Zufall von echten Hardware-Zufallsgeneratoren gelesen und an die *Random*-Komponente weitergeleitet. Dies beschreibt ebenfalls die Hauptaufgabe der *HW-Random*-Komponente. Aus diesem Grund besteht der überwiegende Teil dieser Komponente aus Treibern für Zufallsgeneratoren verschiedener Hersteller, wie zum Beispiel Intel, AMD oder Atmel.

```
/* driver/char/random.c */
void add_hwgenerator_randomness(const char *buffer, size_t count,
    size_t entropy) {
    struct entropy_store *poolp = &input_pool;
    size_t ret;

    if (unlikely(crng_init == 0)) {
        ret = crng_fast_load(buffer, count);
        count -= ret;
        buffer += ret;
        if (!count || crng_init == 0) return;
    }
```

```
12
13    /* Suspend writing if we're above the trickle threshold.
14     * We'll be woken up again once below random_write_wakeup_thresh,
15     * or when the calling thread is about to terminate.
16     */
17    wait_event_interruptible(random_write_wait, !system_wq ||
          kthread_should_stop() || ENTROPY_BITS(&input_pool) <=
          random_write_wakeup_bits);
18
19    mix_pool_bytes(poolp, buffer, count);
20    credit_entropy_bits(poolp, entropy);
21 }
```

Listing 2: add_hwgenerator_randomness()

Die Implementierung von *add_hwgenerator_randomness* ist in Lst. 2 aufgeführt. Zu Beginn der Funktion wird überprüft, ob der Cryptographic-Random-Number-Generator (CRNG) der *Random*-Komponente bereits initialisiert ist. Im Fehlerfall wird versucht eine schnelle Initialisierung durchzuführen und abgebrochen, falls dies nicht gelingt. Das eigentliche Hinzufügen von Zufall geschieht in den Zeilen 19 und 20. Des Weiteren ist hier zu erkennen, dass das Hinzufügen aus zwei getrennten Operationen besteht. Einerseits wird der an die Funktion übergebene Zufall mit *mix_pool_bytes* in den Entropy-Pool eingefügt. Andererseits wird der Entropy-Wert mit *credit_entropy_bits* aktualisiert. Der Aufruf von *wait_event_interruptible* in Zeile 17 sorgt dafür, dass nur Zufall durch einen Hardware-Zufallsgenerator in den Entropy-Pool eingefügt werden kann, wenn eine bestimmt Schwelle unterschritten wurde oder der aufrufende Thread beendet werden soll.

Zwei Begriffe die an dieser Stelle weiterer Erklärung bedürfen sind die Entropy und der Entropy-Pool. Der Entropy-Pool ist lediglich ein C-Struct das Informationen über den im Kernel vorhandenen Zufall enthält. Zwei der darin enthaltenen Werte sind die Entropy selbst und ein Puffer, in dem die durch *mix_pool_bytes* eingefügten Bits gespeichert sind. Bei dem Entropy-Wert handelt es sich um ein Integer, das eine Schätzung darstellt, wie viel Zufall tatsächlich in dem Puffer vorhanden ist.

```
1  /* driver/char/hw_random/core.c */
2  static int hwrng_fillfn(void *unused) {
3      long rc;
4      int i;
5
6      while (!kthread_should_stop()) {
7          struct hwrng *rng;
8
9          rng = get_current_rng();
10         if (IS_ERR(rng) || !rng) break;
11
12         mutex_lock(&reading_mutex);
13         rc = rng_get_data(rng, rng_fillbuf, rng_buffer_size(), 1);
14         mutex_unlock(&reading_mutex);
15
16         put_rng(rng);
17
18         if (rc <= 0) {
19             pr_warn("hwrng: no data available\n");
20             msleep_interruptible(10000);
21             continue;
22         }
23
```

```
24          /* Outside lock, sure, but y'know: randomness. */
25          add_hwgenerator_randomness((void *)rng_fillbuf, rc, rc *
                current_quality * 8 >> 10);
26      }
27
28      hwrng_fill = NULL;
29      return 0;
30  }
```

Listing 3: hwrng_fillfn()

Der eigentliche Aufruf von *add_hwgenerator_randomness* erfolgt in der *hwrng_fillfn* der *HW-Random*-Komponente und ist in Lst. 3 zu sehen. An der Schleife in Zeile 6 ist zu erkennen, dass es sich hierbei um eine Thread-Funktion handelt. Dies ist der Thread, der bei Überschreitung des Schwellwerts zur Befüllung des Entropy-Pools, durch *add_hwgenerator_randomness*, pausiert wird. Bevor in Zeile 25 der Zufall in den Entropy-Pool eingefügt wird, passieren die folgenden Dinge:

1. Ein Zeiger auf den aktuell ausgewählte Zufallszahlengenerator wird angefordert und im Fehlerfall abgebrochen (Zeile 9-10)
2. Der Mutex zum Schutz des Lesevorgangs wird gesperrt (Zeile 12)
3. Lesen des Zufalls (Zeile 13)
4. Der Mutex wird freigegeben (Zeile 14)
5. Der Zufallszahlengenerator wird "zurückgegeben" (Zeile 16)
6. Falls kein Zufall vorhanden war, wird pausiert (Zeile 18-22)

Alle Informationen zu den beiden in diesem Kapitel vorgestellten Komponenten sind in [8], [20], [7] und [19] zu finden.

2.1.3 TUN-, TAP- und Bridge-Devices

In diesem Abschnitt geht es um TUN-, TAP- und Bridge-Devices. Bei diesen drei Begriffen handelt es sich nicht um echte Hardware, sondern um Softwarekomponenten die vom Linux-Kernel bereitgestellt werden. Da alle drei Komponenten konzeptuell zusammenhängen, sind sie gemeinsam in Abb. 4 dargestellt. Der dort abgebildete Aufbau stellt eine Situation dar, wie sie bei der Verwendung der verschiedene Devices üblich ist. Die Hauptaufgabe in diesem Fall ist die Anknüpfung von virtuellen Maschinen an ein Netzwerk. Der blaue Bereich, der alle Komponenten umschließt, stellt in diesem Fall einen gewöhnliches Computersystem dar. Am linken Rand des Systems ist die Netzwerkkarte abgebildet, die ebenfalls eine gewöhnliche Netzwerkkarte darstellt. Diese beiden Bestandteile machen in diesem Beispiel die gesamte Hardware aus. Die restlichen Bestandteile, sind Teil der Software des Systems, was durch die äußere gestrichelte Linie hervorgehoben ist. Am linken Rand des Software-Bereichs befindet sich ein Bridge-Device, das einerseits mit der Netzwerkkarte und andererseits mit TUN-/TAP-Devices verbunden ist. Konzeptionell übernimmt das Bridge-Device die gleiche Aufgabe wie ein Netzwerk-Switch und verteilt eingehende Netzwerkpakete an verbundene Komponenten. Ein Bridge-Device kann somit als virtueller Switch betrachtet werden. Die verbundenen TUN-/TAP-Devices sind ebenfalls eine Art virtuelle Hardware, jedoch keine virtuellen Switches, sondern virtuelle Netzwerkkarten. Die Verbindungen auf der rechten Seite der TUN-/TAP-Devices sorgt für eine Verknüpfung zu der entsprechenden virtuellen Maschine. Zusammenfassend kann der Aufbau wie folgt beschrieben werden: Die virtuellen Maschinen sind jeweils mit einem TUN-/TAP-Devices bzw. einer virtuellen Netzwerkkarte verbunden, um über das Bridge-Device bzw. den virtuellen Switch eine Verbindung zum Netzwerk zu erhalten, die durch die physische Netzwerkkarte ermöglicht wird.

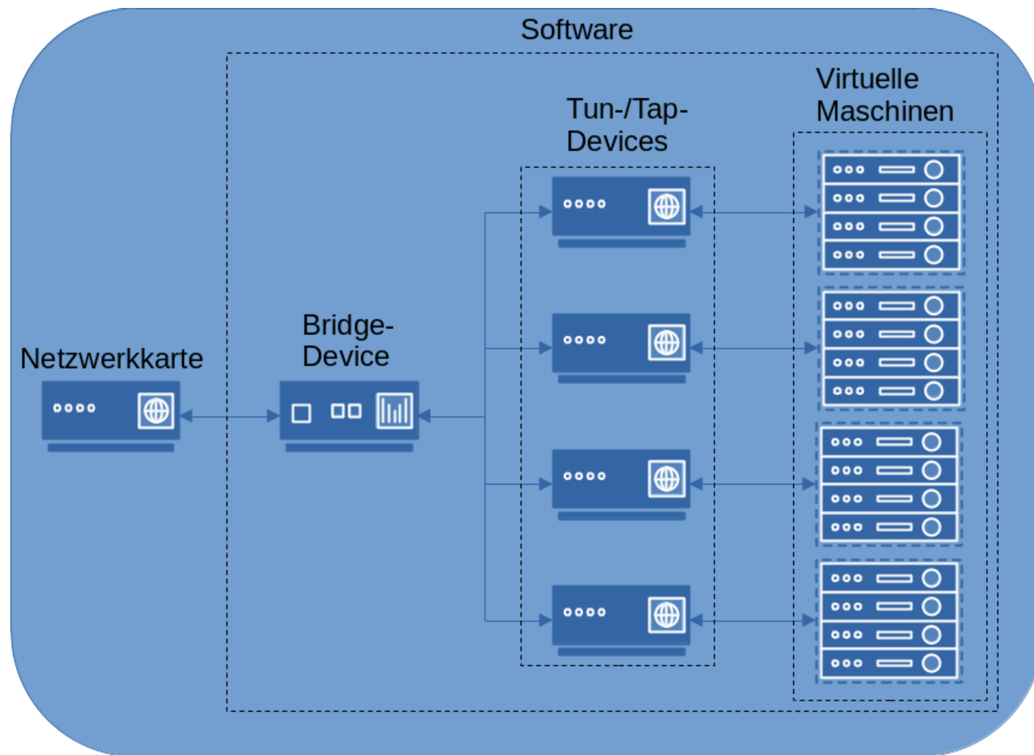

Abbildung 4: TUN-, TAP- und Bridge-Devices im Zusammenspiel

Damit TUN-, TAP- und Bridge-Devices verwendet werden können müssen diese zuerst erzeugt werden. Dies ist sowohl programmatisch mittels *ioctl* oder mit Hilfe von *ip* möglich. In Lst. 4 ist die Erstellung eines Vertreters jedes Typen unter Verwendung von *ip* zu sehen und begrenzt sich jeweils auf einen einzelnen Aufruf. Auf diese Art erstellte Devices sind nicht permanent und müssen beim Neustart des Systems erneut erzeugt werden. Zusätzlich wird hier deutlich, dass TUN- und TAP-Devices eng miteinander verwandt sind. Der wesentliche Unterschied zwischen den beiden Typen ist, dass TUN-Devices auf Schicht 3 und TAP-Devices auf Schicht 2 des Protokollstapels arbeiten. Das bedeutet, TUN-Devices nehmen Internet Protokoll (IP)-Pakete entgegen und TAP-Devices Ethernet-Pakete. Davon abgesehen verhalten sich beide Modi aus Sicht eines Nutzers gleich. Dies zeigt sich auch beim Zugriff, der über User-Space-Programme vonstatten geht. Hierzu muss eine spezielle Datei geöffnet werden, die im Anschluss mittels *ioctl* konfiguriert wird. Die Unterscheidung zwischen TUN-/TAP-Devices geschieht durch setzten eines entsprechende Flags bei der Konfiguration. Im Anschluss daran, können durch Schreiben bzw. Lesen der mit diesem Verfahren geöffneten Datei Netzwerkpakete gesendet bzw. empfangen werden. [12][27]

```
1 ip link add BRIDGE_NAME type bridge
2 ip tuntap add dev TAP_NAME mode tap
3 ip tuntap add dev TUN_NAME mode tun
```

Listing 4: Erstellen eines Bridge- und eines TAP-Devices

2.2 QEMU

Bevor die Arbeit an QEMU möglich ist, muss zuerst das Git-Repository dafür heruntergeladen werden. Hierfür kommt im QEMU-Projekt GitLab als Plattform für die Entwicklung zum Einsatz. Das Haupt-Repository kann unter [13] geklont werden und enthält mehrere Branches die mit den regulären Git-Befehlen ausgewählt werden können. Das Hauptverzeichnis besteht aus Verzeichnissen für Unterkomponenten und Hilfswerkzeuge, Dateien die für Kompilierung wichtig sind, Dokumentation und einigen Quelldateien. Da die Anzahl der Dateien und Verzeichnisse im

Hauptverzeichnis deutlich größer ist als die des Linux-Kernels, wird an dieser Stelle keine grafische Darstellung gezeigt. Die Unterkomponenten die für diese Arbeit hauptsächlich relevant sind beschränken sich auf *net* und *replay* und werden im Verlauf dieses Kapitels genauer vorgestellt.

2.2.1 Konfiguration, Kompilation und Debugging

Als Werkzeug zur Konfiguration wird GNU Autotools[6] im QEMU-Projekt verwendet. Dafür kommt der übliche *configure, make, make install* Prozess zum Einsatz. Das bedeutet, dass im ersten Schritt die Konfiguration durch das im Hauptverzeichnis vorhandene *configure*-Skript durchgeführt wird. Eine Aufgabe dieses Skripts ist die Überprüfung von Abhängigkeiten für die Kompilation, wie zum Beispiel Bibliotheken. Sollte eine Abhängigkeit nicht vorhanden sein, bricht die Konfiguration ab und es ist Aufgabe des Entwicklers fehlende Softwarepakete zu installieren. Ist die Konfiguration abgeschlossen kann mit einem Aufruf von *make* die Kompilation gestartet werden. Alle Artefakte die dabei entstehen werden standardmäßig in einem Unterverzeichnis des QEMU-Hauptverzeichnisses abgelegt, das den Namen *build* trägt. Um die Ausgabe der Kompilation zu löschen steht das bereits bekannte *make clean* Target zu Verfügung. Soll eine vollständig neue Konfiguration durchgeführt werden, kann das *build*-Verzeichnis gelöscht und der Prozess von neuem begonnen werden. Der letzte Schritt, die Installation, ist optional und muss nicht durchgeführt werden.

Bei der Konfiguration steht eine Vielzahl verschiedener Optionen zur Verfügung, die benutzerdefinierte Anpassungen ermöglichen. Auf einige dieser Optionen soll nun eingegangen werden. Um die Kompilationszeit zu verringern, bietet *–target-list=LIST* die Möglichkeit, nur ausgewählt Targets zu erstellen. Im Falle dieser Arbeit besteht die Liste nur aus dem Target *x86_64-softmmu*, da nur die x86_x64-Architektur relevant ist. Der Begriff Softmmu ist QEMU-spezifisch und bezeichnet Binarys die für System-Emulation vorgesehen sind. Das bedeutet, einen echten Computer zu emulieren und beispielsweise Betriebssysteme darauf zu installieren. Alternativ dazu gibt es User-Targets wie zum Beispiel *x86_64-linux-user* mit deren Hilfe die Ausführung von Binarys möglich ist, die für andere Architekturen kompiliert wurden. User-Targets spielen für diese Arbeit allerdings keine Rolle und wurden nur der Vollständigkeit wegen genannt.

Drei weitere Optionen sind *–disable-werror*, *–enable-debug* und *–enable-debug-info*. Die Option *–disable-werror* deaktiviert die Funktionalität, dass Kompilerwarnungen als Fehler interpretiert werden, was standardmäßig bei QEMU der Fall ist. Problematisch kann die Interpretation als Fehler unter anderem dann sein, wenn sich die Version des Kompilers ändert und dabei eine Warnung anders eingestuft wird. Die beiden verbleibenden Optionen *–enable-debug* und *–enable-debug-info* aktivieren zum einen verschiedene Kompileroptionen, die für Debugging-Zwecke hilfreich sind und zum anderen werden einem kompilierten Target Debug-Informationen hinzugefügt.

Nachdem die Konfiguration und die Kompilation abgeschlossen sind, kann QEMU verwendet werden. Detaillierte Informationen dazu enthalten die *man pages* von QEMU, die mit dem Befehl *man qemu* aufgerufen werden können. Hierfür ist es jedoch notwendig, dass QEMU entweder über die Paketverwaltung des Betriebssystems installiert oder der zuvor erwähnte Installations-Schritt durchgeführt wird. Eine weitere Informationsquelle ist die Hilfe-Option von QEMU selbst. Durch Anfügen von *-h* oder *-help* an einen Aufruf von beispielsweise *qemu-system-x86_64* kann diese abgerufen werden.

Zwei Arten QEMU zum Debugging zu verwenden sind, QEMU selbst mit GDB auszuführen oder das innerhalb von QEMU laufende Programm mit GDB auszuführen. Die erste Variante ist hilfreich bei der Entwicklung an QEMU und erfordert lediglich einen Aufruf von GDB mit den entsprechenden Optionen. Hierfür ist es empfehlenswert die zuvor erwähnten Debug-Informationen bei der Konfiguration zu aktivieren, um den Debug-Prozess zu erleichtern. Für die zweite Variante kann ein QEMU ohne Debug-Informationen verwendet werden, jedoch ist es hilfreich das mit QEMU auszuführende Programm mit Debug-Informationen zu versehen. Für den Linux-Kernel ist dies mittels des Konfigurationsmenüs möglich. Hierfür muss unter dem Menüpunkt *Kernel Hacking* die Option *Kernel debugging* aktiviert werden. Damit der Debugging Prozess durchgeführt werden kann, muss QEMU mit einer der Optionen *-s* oder *-gdb DEV*

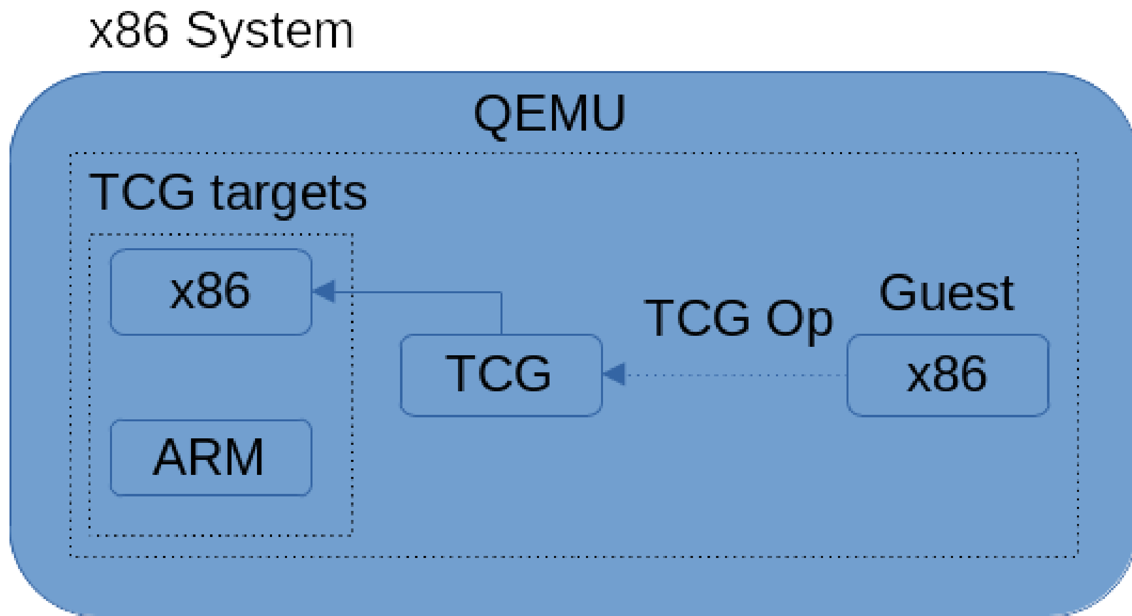

Abbildung 5: Schematische Darstellung der Arbeitsweise des TCG

gestartet werden. In beiden Fällen wird ein GDB-Server gestartet, zu dem eine Verbindung aufgebaut werden kann, um das in QEMU laufende Programm zu debuggen. Der Unterschied zwischen den Optionen ist, dass *-s* eine Kurzform für *-gdb typ::1234* darstellt. Mit *-gdb* hingegen muss das Backend zum Verbindungsaufbau explizit festgelegt werden. Mögliche Optionen für das Backend können in den QEMU-man-pages eingesehen werden. Sobald die Verbindung besteht, kann das emulierte Programm mit GDB analysiert werden. Informationen zur Verwendung von GDB sind in der offiziellen Dokumentation unter [4] zu finden.

2.2.2 Replay

Die Replay-Funktionalität von QEMU ist die wesentliche Eigenschaft, die diese Arbeit ermöglicht. Der Grund dafür ist, dass dadurch die Möglichkeit besteht eine VM aufzuzeichnen und die somit gewonnene Aufzeichnung beliebig oft wiederzugeben. Eine VM aufzeichnen bedeutet dabei diese zu starten, beliebig viele Operationen durchzuführen und abschließend herunterzufahren. Der ursprüngliche Grund für die Implementierung dieser Funktionalität ist, Softwareentwickler bei der Entwicklung zu unterstützen, da die Wiedergabe einer Aufzeichnung zu Analyse- oder Debugging-Zwecken genutzt werden kann.[16]

Der Bestandteil von QEMU, der das Aufzeichnen ermöglicht, heißt Tiny Code Generator (TCG). Ursprünglich als Backend für einen C Compiler vorgesehen, wurde dieser angepasst und ist heutzutage dafür verantwortlich Instruktionen von einer Computer-Architektur in eine andere zu überführen. Somit ist beispielsweise die Ausführung einer für x86 kompilierten Binärdatei auf einem ARM-System möglich. Die genaue Bezeichnung für eine solche Software lautet: Dynamischer Übersetzter. In 5 ist die Vorgehensweise des TCG schematisch dargestellt. Auf der rechten Seite befindet sich das Gast-System, also das System das emuliert wird. Hier werden Instruktionen für ein x86-System in TCG Operations (TCG Ops) umgewandelt. Dabei handelt es sich um einen virtuellen Befehlssatz, der konzeptuell ähnlich zu Zwischendarstellungen bei Compilern ist. Im nächsten Schritt nimmt der TCG die TCG Ops entgegen und führt Optimierungen darauf aus, um die Performanz zu verbessern. Im letzten Schritt werden die TCG Ops in den Befehlssatz des TCG Targets überführt. In diesem Fall verwendet das Zielsystem eine x86-Architektur, weshalb nur der obere Block im Diagramm relevant ist. Die somit generierten

```
-icount  [shift=N|auto][,align=on|off][,sleep=on|off][,rr=record|replay,rrfile=filename[,rrsnap-
shot=snapshot]]
```

Abbildung 6: QEMU-Kommandozeilen-Option zum Konfigurieren des Instruction Counting

```
meson.build     replay-char.c       replay-input.c      replay-net.c       replay-time.c
replay-audio.c  replay-debugging.c  replay-internal.c   replay-random.c    stubs-system.c
replay.c        replay-events.c     replay-internal.h   replay-snapshot.c
```

Abbildung 7: Inhalt der Replay-Komponente

x86-Befehle können im Anschluss von QEMU ausgeführt werden. Zudem ist zu erkennen, dass es mehrere mögliche Zielsysteme gibt.[22][26][23][24]

Die Möglichkeit mit dem TCG dynamisch Befehlssätze ineinander zu überführen ist alleine nicht ausreichend, um eine VM aufzuzeichnen. Um dies zu ermöglichen ist ein weiteres QEMU-Feature notwendig, das *Instruction Counting* genannt wird. Dadurch können Instruktionen, die von einer VM ausgeführt werden, mitgezählt werden. Dieser Wert wird im *TimerState struct* gespeichert, das in Lst. 5 aufgeführt ist und bei der Wiedergabe zur Berechnung der Zeit genutzt. Hierbei gibt es zudem die Möglichkeit festzulegen wie schnell die Zeit während des Abspielens vergehen soll. Die Festlegung dieses Wertes geschieht über die Kommandozeilen-Option von QEMU, die in 6 dargestellt ist. Über den Wert *shift* kann entweder eine Zahl n zwischen 1 und 8 festgelegt werden, um alle 2^n Nanosekunden eine Instruktion auszuführen oder *auto*, um die virtuelle Zeit der Echtzeit anzugleichen.[25]

```c
/* softmmum/timers-state.h */
...
typedef struct TimersState {
    ...
    /* Only written by TCG thread */
    int64_t qemu_icount;
    ...
} TimersState;
...
```

Listing 5: Icount im TimersState struct

Zusätzlich zur Bestimmung der Zeit, wird der Icount von der Replay-Komponente verwendet, die den überwiegende Teil der Replay-Funktionalität beinhaltet. In Abb. 7 sind alle Dateien zu sehen, die Teil der Komponente sind und sich im Unterverzeichnis *replay* des QEMU-Hauptverzeichnis befinden. Bevor auf die einzelnen Bestandteile eingegangen werden kann ist jedoch eine konzeptionelle Betrachtung der Replay-Funktionalität hilfreich. Eine Kerneigenschaft die diese aufweist ist Determinismus. Das bedeutet, dass die Aufzeichnungen, die mit QEMU durchgeführt werden, deterministisch wiedergegeben werden können und somit immer das gleiche Ergebnisse liefern. Dies ermöglicht, die Wiedergabe beliebig oft und auf verschiedenen System auszuführen. Eine Herausforderung die dabei auftritt, ist der Umgang mit nicht-deterministischen Ereignissen. Tritt zum Beispiel ein Interrupt während einer Aufzeichnung auf, muss dieses Ereignis ebenfalls bei der Wiedergabe auftreten. QEMU legt hierfür einen *Event Log* an, der während einer Aufzeichnung mit nicht-deterministischen Events gefüllt wird. Mit Hilfe dieses Event Logs werden dann bei der Wiedergabe alle nicht-deterministischen Ereignisse wieder eingespielt. Die Wiedergabe ist ebenfalls der Punkt an dem der Icount Verwendung findet. Dieser wird dafür genutzt, alle nicht-deterministischen Ereignisse zum richtigen Zeitpunkt auszuführen. Dafür wird neben Ereignis-spezifischen Informationen die Anzahl an Instruktionen im Event Log festgehalten, die seit dem vorherigen Ereignis ausgeführt wurden.[15]

```c
/* replay/replay-internal.h */

/* Any changes to order/number of events will need to bump
```

```c
   REPLAY_VERSION */
enum ReplayEvents {
    /* for instruction event */
    EVENT_INSTRUCTION,
    /* for software interrupt */
    EVENT_INTERRUPT,
    /* for emulated exceptions */
    EVENT_EXCEPTION,
    /* for async events */
    EVENT_ASYNC,
    /* for shutdown requests, range allows recovery of ShutdownCause
       */
    EVENT_SHUTDOWN,
    EVENT_SHUTDOWN_LAST = EVENT_SHUTDOWN + SHUTDOWN_CAUSE__MAX,
    /* for character device write event */
    EVENT_CHAR_WRITE,
    /* for character device read all event */
    EVENT_CHAR_READ_ALL,
    EVENT_CHAR_READ_ALL_ERROR,
    /* for audio out event */
    EVENT_AUDIO_OUT,
    /* for audio in event */
    EVENT_AUDIO_IN,
    /* for random number generator */
    EVENT_RANDOM,
    /* for clock read/writes */
    /* some of greater codes are reserved for clocks */
    EVENT_CLOCK,
    EVENT_CLOCK_LAST = EVENT_CLOCK + REPLAY_CLOCK_COUNT - 1,
    /* for checkpoint event */
    /* some of greater codes are reserved for checkpoints */
    EVENT_CHECKPOINT,
    EVENT_CHECKPOINT_LAST = EVENT_CHECKPOINT + CHECKPOINT_COUNT - 1,
    /* end of log event */
    EVENT_END, EVENT_COUNT
};

/* Asynchronous events IDs */
enum ReplayAsyncEventKind {
    REPLAY_ASYNC_EVENT_BH, REPLAY_ASYNC_EVENT_BH_ONESHOT,
    REPLAY_ASYNC_EVENT_INPUT, REPLAY_ASYNC_EVENT_INPUT_SYNC,
    REPLAY_ASYNC_EVENT_CHAR_READ, REPLAY_ASYNC_EVENT_BLOCK,
    REPLAY_ASYNC_EVENT_NET, REPLAY_ASYNC_COUNT
};
```

Listing 6: Enums die Ereignisse definieren

Mit diesem Wissen lassen sich nun die zuvor erwähnten Dateien in Abb. 7 erklären und unterscheiden. Der Großteil der vorhandenen Dateien ist für die Verarbeitung spezifischer Ereignisse vorgesehen. So zum Beispiel die Datei *replay-audio.c*, die für Audio-Ereignisse zuständig ist oder *replay-net.c* für Netzwerk-Ereignisse. Im Gegensatz zu Dateien die spezifische Ereignisse verarbeiten, übernimmt die Datei *replay-events.c*, allgemeine, Ereignis-bezogene Aufgaben. Einerseits ist das das Auslesen des nächsten Events und das anschließende weiterleiten an eine der spezifischen Dateien, anhand der Art des Ereignisses. Andererseits befinden sich hier allgemeine Funktionen wie das eben erwähnte Auslesen von Ereignissen, das Speichern von Er-

```
67: EVENT_CP_RESET(31)
68: EVENT_CLOCK_HOST(21): 0x16f3eddf85022058
77: EVENT_CLOCK_HOST(21): 0x16f3eddf85022058
86: EVENT_CLOCK_HOST(21): 0x16f3eddf85022440
95: EVENT_CLOCK_HOST(21): 0x16f3eddf89b1c5b8
104: EVENT_CP_CLOCK_HOST(28)
105: EVENT_CLOCK_HOST(21): 0x16f3eddf89b315a8
114: EVENT_CLOCK_HOST(21): 0x16f3eddf89b31990
123: EVENT_CP_CLOCK_HOST(28)
124: EVENT_CLOCK_HOST(21): 0x16f3eddf89b33100
133: EVENT_CLOCK_HOST(21): 0x16f3eddf89b334e8
142: EVENT_CP_CLOCK_HOST(28)
143: EVENT_ASYNC(3)
145: REPLAY_ASYNC_EVENT_NET(6): size: 0x78, data: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 51, 51, 0, 0, 0, 22, 122,
 93, 122, 18, 193, 145, 134, 221, 96, 0, 0, 0, 0, 56, 0, 1, 254, 128, 0, 0, 0, 0, 0, 0, 120, 93, 122, 255,
 254, 18, 193, 145, 255, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 22, 58, 0, 5, 2, 0, 0, 1, 0, 143, 0, 24
8, 80, 0, 0, 0, 2, 4, 0, 0, 0, 255, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 251, 4, 0, 0, 0, 255, 2, 0,
0, 0, 0, 0, 0, 0, 0, 0, 1, 255, 18, 193, 145]
```

Abbildung 8: Ausschnitt aus einer Aufzeichnung

eignissen, sowie die Aktivierung und Deaktivierung des Ereignis-Systems. In *replay.c* sind die Funktionen enthalten die allgemeine Aufgaben erfüllen, jedoch keine Ereignis-bezogenen. Beispiele hierfür sind die Initialisierung, das Starten und das Beenden des Replay-Systems. Da einer der Anwendungsfälle der Replay-Funktionalität das Debugging ist, gibt es in *replay-debugging.c* Funktionen die dies unterstützen. Eine Funktionalität die hier implementiert wird ist das Reverse Debugging, was eine Rückwärtsbewegung beim Debugging ermöglicht, also das Programm in umgekehrter Reihenfolge zu durchlaufen. Die letzten wesentlichen Dateien sind *replay-internel.h* und *replay-internal.c*. Darin sind ebenfalls allgemeine Funktionen angesiedelt, allerdings nur solche die innerhalb der Replay-Komponente verwendet werden, wie die Funktionen zum Schreiben und Lesen der Bytes der Event-Log-Datei. Außerdem sind dort die Event-Arten definiert, die in Lst. 6 gelistet sind. Jedes Ereignis entspricht dabei einem Eintrag im *ReplayEvents* oder *ReplayAsyncEventKind* enum. Ausnahme bilden *EVENT_SHUTDOWN*, *EVENT_CLOCK* und *EVENT_CHECKPOINT*, die jeweils mehrere Ereignisse durch einen Start- und einen Endpunkt definieren, sowie *EVENT_COUNT* und *REPLAY_ASYNC_COUNT* die die maximale Größe der enums festlegen. Beim Durchführen einer Aufzeichnung werden diese Werte mit Ereignis-spezifischen Zusatzinformationen in eine Datei geschrieben. Bei *EVENT_INSTRUCTION* etwa der Icount. Eine Besonderheit ist zudem das Ereignis *EVENT_ASYNC*, auf das immer eines der asynchronen Ereignisse im unteren enum folgt, also immer nur zweistufig vorkommt. [15][16]

Um eine bessere Vorstellung vom Aufzeichnungs-Format zu erlangen, zeigt Abb. 8 einen Ausschnitt aus einer Event-Log-Datei. Der Aufbau der Ausgaben einzelner Events ist immer gleich. Links steht die Nummer des Ereignis in der Datei, gefolgt vom Namen und dem Wert aus dem enum und danach Ereignis-spezifische Daten. Bei Ereignis 145 beispielsweise handelt es sich um ein Netzwerk-Ereignis, also einem eingehenden oder ausgehenden Netzwerkpaket, von dem die Daten und deren Größe ausgegeben werden. Zur Erzeugung einer solchen Ausgabe, für beliebige Aufzeichnungen, enthält das *scripts*-Verzeichnis des QEMU-Repository die Datei *replay_dump.py*. Zudem sei gesagt, dass die Ausgabe nicht vollständig ist, da einig Werte beim Lesen durch das Skript ignoriert werden. Soll eine vollständige Ausgabe erzeugt werden, sind Anpassungen am Skript nötig.

2.2.3 Networking

In diesem Abschnitt werden drei Konzepte vorgestellt, die Teil der Netzwerk-Funktionalität von QEMU sind und denen größere Bedeutung in dieser Arbeit zukommt: *Network Backends*, *Virtual Network Devices* und *Network Filters*. Alle drei Konzepte sind eng miteinander verbunden. Bei Network Backends und Virtual Network Devices handelt es sich jedoch um grundlegendere Bestandteile, da beide notwendig sind, um eine Netzwerkverbindung aufzubauen. Network Filters hingegen müssen nicht eingesetzt werden, da sie optionale Funktionalitäten bereitstellen.

Abb. 9 zeigt einen konzeptionellen Aufbau, wie er in der Realität oft eingesetzt wird. Zu

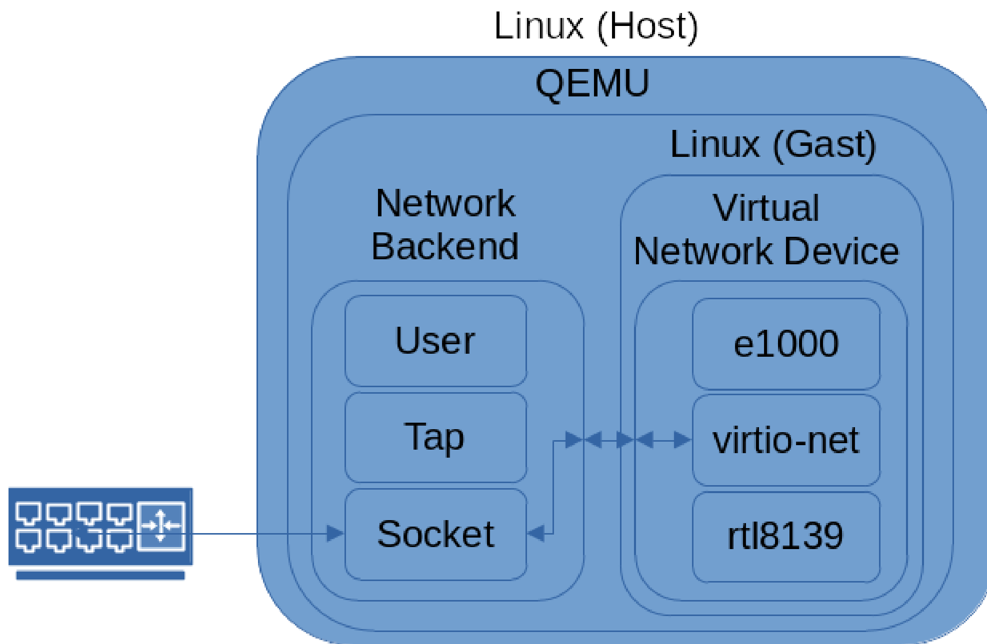

Abbildung 9: Network Backends in Verbindung mit Virtual Network Devices

sehen ist ein einzelnes physisches System, das Host-System. Auf diesem System wird ein Linux verwendet, das eine einzelne QEMU-Instanz ausführt. Innerhalb der QEMU-Instanz ist einerseits das Network Backend angesiedelt und andererseits das Gast-System, also das System das von QEMU emuliert wird. Das Gast-System wiederum, das ebenfalls ein Linux ausführt, enthält ein Virtual Network Device. Mit dieser Konstellation ist das Gast-System in der Lage Netzwerk-Pakete zu versenden. Ausgehend von dem Virtual Network Device, das Pakete vom Linux-Gast entgegennimmt, werden diese über das Network Backend zum Beispiel an einen Router übermittelt, um eine Verbindung zum Internet aufzubauen. Würde einer der beiden Bestandteile fehlen, wäre kein Verbindungsaufbau möglich. Der genaue Pfad vom Network Backend zum Router ist an dieser Stelle vereinfacht dargestellt und enthält in der Praxis weitere Schritte, die für das Verständnis der QEMU-Bestandteile jedoch nicht relevant sind und deshalb nicht abgebildet werden.

Innerhalb der beiden Bestandteile verdeutlichen die unterschiedlichen Kästen, dass die konkreten Implementierungen austauschbar sind. Im dargestellten Beispiel handelt es sich bei *virtio-net* um eine virtuelle Netzwerkkarte, die kein physisches Gegenstück besitzt. Im Gegensatz dazu sind *e1000* und *rtl8139* virtuelle Implementierungen physischer Netzwerkkarten. Neben den hier gezeigten Vertretern gibt es noch eine Vielzahl weiterer virtueller Netzwerkkarten, die über die Hilfefunktion von QEMU einsehbar sind. Auf Seiten des Network Backends wird das Socket-Backend verwendet, das es ermöglicht, über ein Socket eine Verbindung aufzubauen und darüber Netzwerkpakete zu senden oder zu empfangen. Alternativ zum Socket-Backend gibt es unter anderem das TAP-Backend, bei dem Netzwerkpakete über ein Linux-TAP-Device gesendet werden, und das User-Backend, das das Standard-Backend von QEMU ist und dem in dieser Arbeit keine Bedeutung zukommt.

In Lst. 7 ist eine mögliche Variante zu sehen, um beide Bestandteile zu erstellen. Beim Aufrufe von QEMU wird zum Erstellen von Network Backends die Option *-netdev* und zum Erstellen von Virtual Network Devices die Option *-device* verwendet. Das Network Backend ist vom Typ *TAP* und das Virtual Network Device vom Typ *virtio-net*. Die restlichen Optionen und ein Vielzahl weiterer Optionen kann über die QEMU-man-page eingesehen werden. [14]

```
1 qemu-system-x86_64 \
2     -netdev tap,id=net1,ifname=thesis_tap,script=no,downscript=no \
```

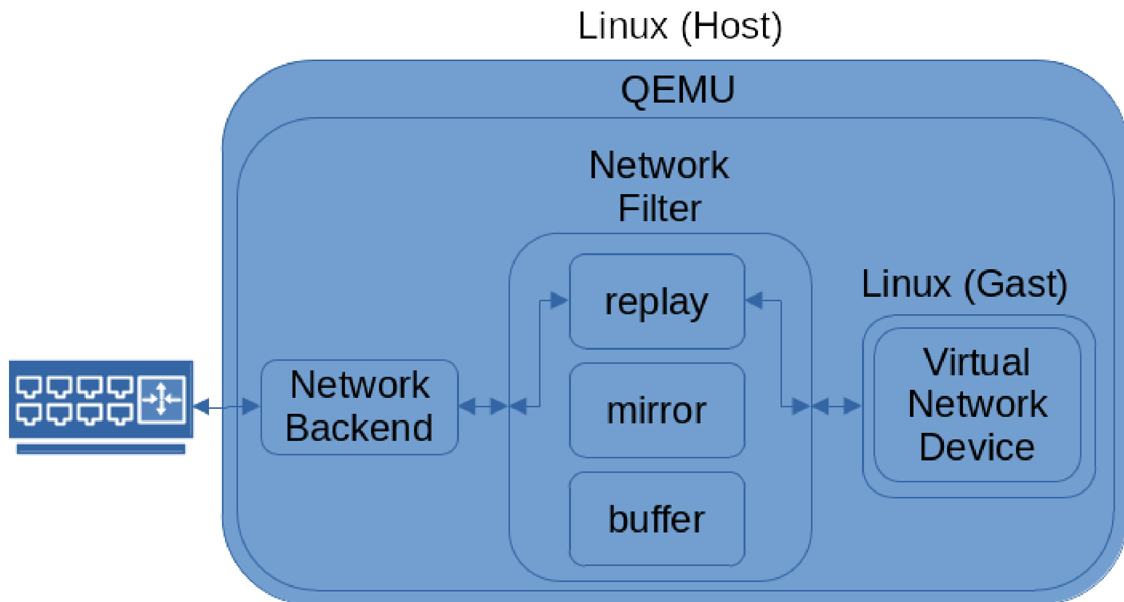

Abbildung 10: Network Filters im Zusammenspiel mit Network Backends und Virtual Network Devices

```
3       -device virtio-net,netdev=net1,mac=42:42:42:42:42:42 \
```

Listing 7: Kommandozeile zum Erstellen von Network Backends und Virtual Network Devices

Die Network-Filter als letzte der drei Komponenten ist in Abb. 10 dargestellt. Dieses Schaubild ist eine Erweiterung der Abb. 9 und platziert die Network Filters zwischen den Network Backends und den Virtual Network Devices. Damit können sowohl eingehende, als auch ausgehende Netzwerkpakete von verschiedenen Filter-Funktionen untersucht werden. In diesem Fall wird der Replay-Filter verwendet, der im späteren Verlauf dieser Arbeit relevant ist. Ein Unterschied zu den beiden anderen Bestandteilen ist, dass nicht nur einer der Filter verwendet werden kann. Ist der Einsatz mehrerer Filter gewünscht, können weitere Filter registriert werden. Alle registrierten Filter werden bei der Ausführung nacheinander durchlaufen.

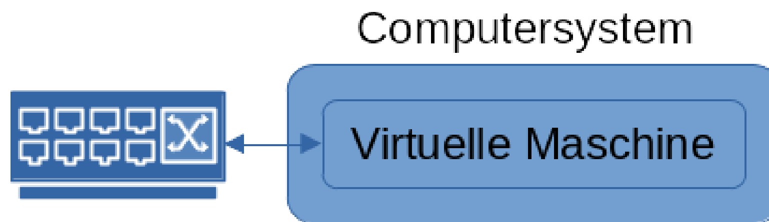

Abbildung 11: System das die Ausgangssituation dieser Arbeit darstellt

3 Problemstellung

Das Ausgangsszenario, das die Grundlage für die Umsetzung dieser Arbeit bildet, ist die Allgemeine Situation, dass ein System, auf dem eine VM ausgeführt wird, angreifbar ist, auf Grund einer Schwachstelle. Welche konkrete Schwachstelle dabei ausgenutzt wird, ist von geringer Bedeutung. Denkbar sind sowohl Schwachstellen in der Software, als auch in der Hardware eines Systems. Wichtig ist nur, dass der Angreifer den Ablauf des Systems beeinflussen kann und dadurch das Systems korrumpiert.

Teil des Ausgangsszenarios ist zudem die Verbindung mit einem Netzwerk. Dies muss nicht notwendigerweise eine Verbindung zum Internet beinhalten, trifft in dieser Arbeit jedoch zu. Dieser grundlegende und simple Aufbau ist in Abb. 11 noch einmal grafisch dargestellt.

Bei einem Angriff auf dieses System besteht die Gefahr, dass Informationen von innerhalb des Systems über das Netzwerk nach außen gelangen, obwohl diese nicht beabsichtigt ist. Beispielsweise kann ein Angreifer kryptografische Schlüssel manipulieren, um durch die daraus resultierende unsichere Netzwerkverbindung geheime Informationen zu erlangen. Das Ziel ist es nun solche Situationen zu verhindern, indem das System so erweitert wird, dass Angreifer dieses nicht unbemerkt manipulieren kann. Konkret bedeutet das, es wird untersucht, ob die Konstruktion eines System möglich ist, um zwei der in der Informationssicherheit gebräuchlichen CIA-Eigenschaften zu erreichen: Confidentiality und Integrity.

Dafür wird der Ausgangszustand des Systems, wie er in Abb. 11 zu sehen ist, unter Verwendung der Replay-Funktionalität von QEMU erweitert. Einen grundlegenden Überblick dieses Zielzustandes gibt Abb. 12. Der markanteste Unterschied ist die Anzahl der Systeme, die von einem einzelnen System auf drei Systeme ansteigt. Die beiden unteren Systeme haben den gleichen Aufbau wie das System in Abb. 11. Beide führen eine VM aus, die in dieser Darstellung jedoch nicht abgebildet ist. Die konkreten Ausführungen beider Systeme unterscheiden sich jedoch geringfügig. Das linke System nutzt die Replay-Funktionalität von QEMU, um die Ausführung der VM aufzuzeichnen und dann über den Ereignis-Strom an das rechte System zu übertragen. Dieses wiederum nimmt die aufgezeichneten Ereignisse entgegen, um dann die Aufzeichnung abzuspielen. Anstatt die Ausführung abzuspeichern und im Nachhinein abzuspielen, was der Standard-Anwendungsfall der Replay-Funktionalität ist, wird das Abspielen parallel mit etwas Zeitverzögerung durchgeführt.

Damit die oben genannten Sicherheitsziele erreicht werden können, ist jedoch noch das dritte System erforderlich. Der Komparator hat einerseits die Aufgabe eingehende Netzwerkpakete an die beiden unteren Systeme weiterzuleiten und andererseits ausgehende Netzwerkpakete auf Gleichheit zu überprüfen. Diese Überprüfung ist der wesentliche Bestandteil für das Erreichen der Sicherheitsziele, denn der Komparator übermittelt nur Netzwerkpakete, die sowohl vom aufzeichnenden als auch vom abspielenden System gesendet werden. Das bedeutet, dass alle Pakete bitweise gleich sein müssen. Implizit muss dadurch zudem die Reihenfolge der Pakete gleich sein, da der Komparator selbst keine Information über die Ausführung hat, die auf den beiden Systeme abläuft.

Für die Gewährleistung der Sicherheitsziele müssen zudem mehrere Annahmen getroffen werden. Diese Annahmen betreffen einerseits welche Systeme tatsächlich durch Angreifer korrumpiert sind und andererseits die Fähigkeiten eines Angreifers. Zusätzlich dazu ist das Szenario in dem der Aufbau verwendet wird von Bedeutung. In erster Linie ist der Aufbau dafür gedacht

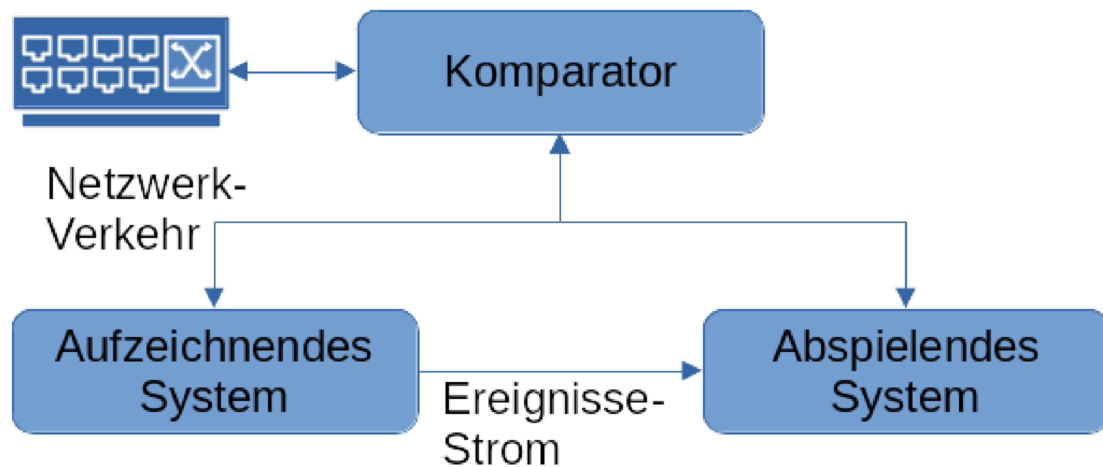

Abbildung 12: Darstellung des Zielsystems

Server-Anwendungen abzusichern, die über das Netzwerk kommunizieren. Ein Beispiel dafür ist etwa ein Server, auf dem ein Webserver läuft und der über eine Secure Shell (SSH)-Verbindung erreicht werden kann. In solchen Szenarien kann die Netzwerkverbindung abgesichert werden, um keine Informationen unbeabsichtigt nach außen zu kommunizieren. Kein Ziel ist, sicherzustellen, dass auf dem aufzeichnenden und dem abspielenden System genau die gleichen Berechnungen durchgeführt werden. Wenn ein Angreifer nur lokal auf dem korrumpierten System unerwünschte Berechnungen ausführt, aber das Verhalten nach außen über die Netzwerkverbindung sich nicht ändert, dann wird dies nicht erkannt. Ein triviales Beispiel ist etwa das Aktivieren einer LED.

Welche Systeme des Aufbaus tatsächlich durch Angreifer korrumpiert sind führt zu einer Annahme, die ebenfalls die Grundidee und die Motivation für diese Arbeit darstellt. Diese lautet, dass wenn zwei der drei Systeme nicht durch Angreifer korrumpiert sind, können die Sicherheitsziele gewährleistet werden. Verdeutlichen lässt sich diese Aussage anhand einiger Beispiele. Angenommen die aufzeichnende VM ist durch einen Angreifer korrumpiert und dieser Angreifer nutzt während der Ausführung eine Schwachstelle aus, um eine Änderung vorzunehmen, die das Verhalten nach außen verändert. Sobald der Vergleich der Netzwerkpakete durch den Komparator abgeschlossen ist, erkennt dieser den Betrugsversuch und stoppt die weitere Übertragung. Sind die Rollen der VMs vertauscht, also die Abspielende korrumpiert, ergibt sich das gleiche Verhalten. Ist der Komparator korrumpiert sein, besteht die Möglichkeit, dass dieser beliebige Kommunikation nach außen betreibt, jedoch können keine geheimen Informationen übertragen werden, da der Komparator keine Kenntnisse über den Zustand der VMs besitzt. Dies gilt jedoch nur, wenn kein Man-in-the-Middle (MITM)-Angriff durch den Komparator möglich ist. Ist ein solcher Angriff möglich, zum Beispiel, weil dem SSH-Client der öffentliche Schlüssel des Servers nicht bekannt ist und dieser zu Beginn einer Verbindung ausgetauscht wird, dann können die Sicherheitsziele nicht gewährleistet werden.

Gewährleistet werden können die Sicherheitsziele ebenfalls nicht, wenn mehr als eines der Systeme durch einen Angreifer korrumpiert ist. Für den Fall, dass beide VMs korrumpiert sind, können diese beliebige Änderungen vornehmen, solange die Netzwerkpakete beider Systeme gleich sind. Der Komparator hat keine Möglichkeit dies zu erkennen. Wenn nicht die beiden VMs, sondern der Komparator und eine VM korrumpiert sind, ist dies ebenfalls unsicher. Da der Komparator die Entscheidungskraft darüber hat welche Pakete versendet werden, können beliebige Pakete gesendet werden, unabhängig davon, ob Aufzeichnung und Wiedergabe gleich sind.

Confidentiality und Integrity ergeben sich nun beide daraus, dass veränderte Netzwerkpakete erkannt werden können. Tritt eine Veränderung auf einem System auf, unterscheiden sich die Pakete, die von den beiden Systemen gesendet werden. Sie sind also nicht integer, weil bei einem der Pakete eine Veränderung stattfand. Confidentiality besteht deshalb ebenfalls nicht, da keine Aussage darüber getroffen werden kann, ob durch die Veränderung geheime Informationen nach

außen geraten oder nicht.

Bezogen auf den Angreifer werden zwei Annahmen getroffen. Zum einen darf ein Angreifer keine Eingabe-Ereignisse erzeugen und zum anderen darf ein Angreifer nicht alleine den Zufall bestimmen, der innerhalb der VMs eingesetzt wird. Die Annahme über die Eingabe-Ereignisse bezieht sich auf die Ereignisse, die durch die Replay-Funktionalität von QEMU zur Verfügung stehen. Ist ein Angreifer in der Lage auf dem aufzeichnenden System beliebige Eingabe-Ereignisse zu erzeugen, kann dadurch eine beliebige Ausführungen erzeugt werden. Damit besteht die Möglichkeit das abspielende System zu kontrollieren, was zu einer Situation führt, die durch die Annahme über das Szenario bereits ausgeschlossen wurde. Die Systeme sind dadurch nur indirekt über beispielsweise Interrupt-Timings manipulierbar.

Der Zufall, der über die Linux-Betriebssysteme innerhalb der VMs bezogen wird, darf ebenfalls nicht durch einen Angreifer manipulierbar sein. Insbesondere dürfen die VMs jeweils nicht alleinig den Zufall der anderen bestimmen. Wenn etwa die aufzeichnende VM den Zufall wählen kann und dieser ebenfalls von der abspielenden VM verwendet wird, besteht die Möglichkeit, dass dieser Zufall eine Form besitzt, die bei sicherheitskritischen Berechnungen problematisch ist. Da jedoch beide Systeme den gleichen, kompromittierten Zufall verwenden, können geheime Informationen nach außen gelangen, weil der Komparator dies nicht erkennt.

Ein letzte Eigenschaft, die der Aufbau besitzt ist, das auf Grund der Verwendung von QEMU unterschiedliche Hardware für die Systeme verwendet werden kann. Dies bringt nicht direkt Vorteile für die Sicherheit mit sich, erschwert aber möglicherweise einen Angriff. Wenn zum Beispiel das aufzeichnende System ein x86-basiertes System und das abspielende System ein ARM-basiertes System ist, dann können nicht die gleichen Hardware-Schwachstellen genutzt werden.

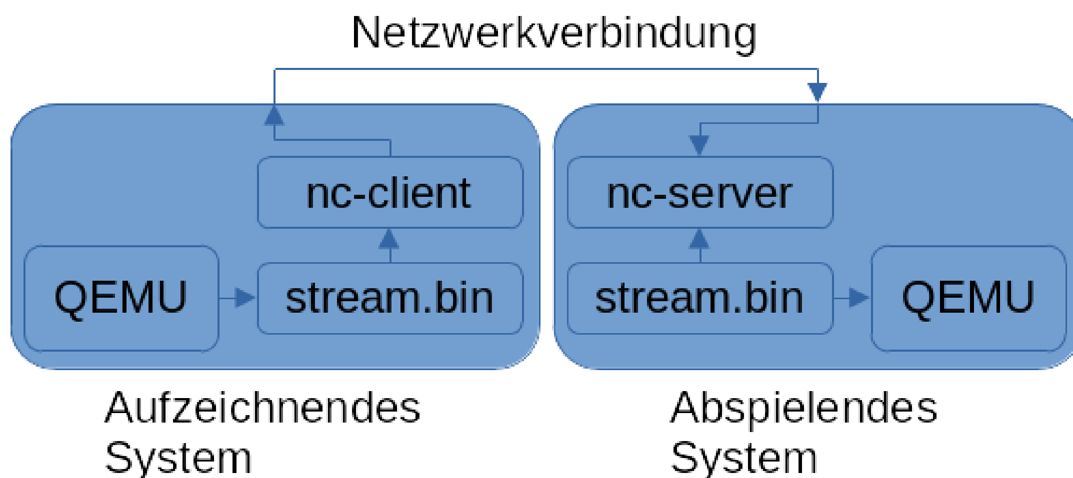

Abbildung 13: Umsetzung des Ereignis-Stroms

4 Umsetzung

Nach der Auseinandersetzung mit der Problemstellung und dem grundlegende, konzeptionellen Aufbau des Systems aus Kapitel (Kap.) 3, folgt nun die detaillierte Betrachtung des Systems, das im Laufe dieser Arbeit entstanden ist. Dies geschieht in einer schrittweisen, iterativen Vorstellung der Ergebnisse. Das bedeutet, dass das Ausgangssystem aus Kap. 3, in jedem Unterkapitel um bestimmte Funktionalitäten erweitert wird und am Ende das vollständige System daraus resultiert.

4.1 Schritt 1: Live abspielen über einen Ereignisstrom

Basierend auf Abb. 11 und 12 besteht der erste Schritt daraus das System dahingehend zu erweitern, dass ein quasi-paralleles Aufzeichnen und Abspielen über einen Ereignis-Strom möglich ist. Quasi-parallel deshalb, weil durch die Übertragung der nicht-deterministischen Ereignisse über den Ereignis-Strom eine Latenz hinzukommt. Um diese Erweiterung umzusetzen sind zwei Anpassungen nötig: Eine Verbindung zwischen zwei unterschiedlichen QEMU-Instanzen schaffen und über diese Verbindung die Übertragung von Ereignissen ermöglichen.

```
1 qemu-system-x86_64
2     ...
3     -icount shift=2,rr=record,rrfile=stream.bin
4     -drive file=deb.qcow2,if=none,snapshot=on,id=img-direct
5     -drive driver=blkreplay,if=none,image=img-direct,id=img-blkreplay
6     -device ide-hd,drive=img-blkreplay
7     ...
```

Listing 8: QEMU-Optionen zur Aktivierung der Replay-Funktionalität

Für die erste Anpassung ist eine Betrachtung der QEMU-Kommandozeilen-Option zum Aktivieren der Replay-Funktionalität der erste Ansatzpunkt. Diese wurde bereits in Abb. 6 vorgestellt. Für bessere Übersichtlichkeit ist diese erneut in Lst. 8 aufgeführt, mit einigen weiteren Optionen. Dies ist ein Ausschnitt eines Skripts, das für das Ausführen der QEMU-Instanzen zuständig ist und zeigt neben drei Optionen, die für den virtuellen Datenträger verantwortlich sind, auch die *icount*-Option. Entscheidend für die Übertragung von Ereignissen über einen Ereignis-Strom ist die *rrfile*-Option. Damit wird QEMU die Datei mitgeteilt, die die Aufzeichnung enthalten soll. Unter Verwendung von *netcat* und *FIFO special files* kann die rrfile-Option dazu genutzt werden das in Abb. 13 gezeigte System zu realisieren.

Zu erkennen sind wieder zwei Systeme, das aufzeichnende System links und das abspielende System rechts. Das aufzeichnende System führt wie zuvor auch eine QEMU-Instanz aus, die

```
00000000: 00e0 200a 0000 0000 0000 0000 1516 f3ed  .. .............
00000010: df7f dc68 681e 1516 f3ed df84 9247 8815  ...hh........G..
00000020: 16f3 eddf 8492 95a8 1516 f3ed df84 9299  ................
00000030: 9015 16f3 eddf 8492 9990 1516 f3ed df84  ................
00000040: 929d 781f 1516 f3ed df85 0220 5815 16f3  ..x........ X...
00000050: eddf 8502 2058 1516 f3ed df85 0224 4015  .... X.......$@.
00000060: 16f3 eddf 89b1 c5b8 1c15 16f3 eddf 89b3  ................
00000070: 15a8 1516 f3ed df89 b319 901c 1516 f3ed  ................
```

Abbildung 14: Ausschnitt aus einer Aufzeichnungs-Datei

nicht-deterministische Ereignisse in eine Datei schreibt. Der Unterschied zur herkömmlichen Vorgehensweise ist, dass die Datei, hier *stream.bin* genannt, keine reguläre Datei ist, sondern ein *FIFO special file*.

Konzeptionell verhält diese Art von Datei sich wie ein gewöhnlicher FIFO-Speicher. Das bedeutet, dass Daten die in die Datei geschrieben werden in genau der gleichen Reihenfolge auch ausgelesen werden. Linux-intern verhalten FIFOs sich wie Pipes, weshalb sie auch als *Named Pipes* bezeichnet werden. Der Unterschied besteht lediglich darin, dass Zugriff über das Dateisystem möglich ist. Zudem gibt es die Besonderheit, dass FIFOs sowohl lesend als auch schreibend geöffnet werden müssen, bevor eine Übertragung möglich ist. Weiter Informationen gibt es in den man pages *fifo(7), pipe(2) und mkfifo(1)*.

Die aufzeichnende QEMU-Instanz schreibt also alle auftretenden Ereignisse in die Datei *stream.bin*. Am Leseende des FIFOs wiederum befindet sich eine *netcat*-Instanz im Client-Modus. *netcat* ist ein Werkzeug, das es ermöglicht TCP- und UDP-Verbindungen aufzubauen und beliebige Daten zu übertragen. In diesem Fall werden die Ereignisse entgegengenommen und an eine netcat-Server auf dem zweiten System gesendet. Am abspielenden System angekommen, geschieht das Gleiche wie auf dem aufzeichnenden System, nur in umgekehrter Reihenfolge. Der netcat-Server schreibt die Ereignisse in einen FIFO, aus dem eine QEMU-Instanzen liest und die Ereignisse verwendet, um die Wiedergabe durchzuführen.

Damit ist die erste Anpassung, eine Verbindung zwischen zwei QEMU-Instanzen herzustellen, abgeschlossen. Ohne weitere Änderungen ist jedoch keine Übertragung möglich. Der Grund dafür ist das Format mit dem die Ereignis-Dateien geschrieben werden. In Kap. 2.2.2 wurde bereits vorgestellt in welcher Form Ereignisse in die Datei geschrieben werden. Zusätzlich dazu beginnt jede Datei jedoch mit einem 12 Byte großen Header, der unter anderem die Version des Datei-Formats enthält. Dargestellt ist dies in Abb. 14. Der rot markierte Bereich kennzeichnet den Header einer Aufzeichnungs-Datei, die in Hexadezimaler-Darstellung angezeigt ist. Die ersten 4 Byte enthalten die Version und sind gefolgt von 8 Byte Padding, die immer 0 sind.

Eine Übertragung ist deshalb nicht möglich, weil der Header erst am Ende einer Aufzeichnung geschrieben wird. Bei der Wiedergabe hingegen wird der Header zu Beginn gelesen, um sicherzustellen, dass das Datei-Format der Aufzeichnung mit der in der QEMU-Instanz implementierten Version übereinstimmt. Die zwei Code-Bestandteile der Replay-Komponente, die das Lesen und Schreiben des Headers übernehmen, sind in Lst. 9 aufgeführt. Zu sehen sind hier Ausschnitte der Funktionen *replay_enable* und *replay_finish*, die beim Start bzw. beim Beenden einer Aufzeichnung aufgerufen werden. Außerdem ist zu erkennen, dass die Datei-Version erst am Ende eine Aufzeichnung in den Zeilen 18 und 19 von *replay_finish* an den Anfang der Datei geschrieben wird. Beim Start hingegen wird in Zeile 6 lediglich der Header übersprungen.

```
1 /* replay/replay.c */
2 static void replay_enable(const char *fname, int mode) {
3     ...
4     /* skip file header for RECORD and check it for PLAY */
5     if (replay_mode == REPLAY_MODE_RECORD) {
```

```
        fseek(replay_file, HEADER_SIZE, SEEK_SET);
    } else if (replay_mode == REPLAY_MODE_PLAY) {
        ...
    }
    ...
}

void replay_finish(void) {
    ...
    if (replay_mode == REPLAY_MODE_RECORD) {
        ...
        /* write header */
        fseek(replay_file, 0, SEEK_SET);
        replay_put_dword(REPLAY_VERSION);
    }
    ...
}
```

Listing 9: Ausschnitt aus der Replay-Komponente von QEMU

Aus diesem Wissen ergibt sich, dass für die Übertragung von Ereignissen die Version im Header zu Beginn einer Aufzeichnung geschrieben werden muss. Hierfür müssen die Zeilen 18 und 19 der *replay_finish*-Funktion vor dem Aufruf von *fseek* in Zeile 6 der *replay_enable*-Funktion platziert werden. Das Ergebnis dieser Änderung ist in Lst. 10 zu sehen.

```
/* replay/replay.c */
static void replay_enable(const char *fname, int mode) {
    ...
    /* skip file header for RECORD and check it for PLAY */
    if (replay_mode == REPLAY_MODE_RECORD) {
        fseek(replay_file, 0, SEEK_SET);
        replay_put_dword(REPLAY_VERSION);
        fseek(replay_file, HEADER_SIZE, SEEK_SET);
    } else if (replay_mode == REPLAY_MODE_PLAY) {
        ...
    }
    ...
}
```

Listing 10: Erweiterung zum Schreiben des Headers am Anfang einer Aufzeichnung

Eine hilfreiche Eigenschaft, die die in diesem Kapitel vorgenommenen Änderungen ebenfalls gewährleisten, ist die Möglichkeit beide QEMU-Instanzen auf dem gleichen physischen oder zwei unterschiedlichen physischen System auszuführen. Dadurch, dass *netcat* zum Einsatz kommt, kann sowohl der Localhost, als auch ein beliebiges System im Netzwerk als Kommunikationspartner verwendet werden. Da der Zielzustand jedoch zwei physische Systeme beinhalten soll, wurde in Abb. 13 und im Großteil dieses Kapitels auf die Localhost-Variante nicht eingegangen.

4.2 Schritt 2: Zufall der VMs festlegen

Der zweite Schritt der Umsetzung nimmt Abb. 13 als Grundlage und erweitert das System mit dem Ziel, auf beiden Maschinen den gleichen Zufall bereitzustellen. Konzeptionell handeln die beiden VMs zu Beginn einer Aufzeichnung Zufall aus und verwenden ausschließlich diesen während der Ausführung. Der Grund für dieses Vorgehen ist, dass keiner der beiden Teilnehmer alleinig die Möglichkeit haben darf Zufall zu bestimmen. Insbesondere die in Kap. 2.1.2 in Lst. 1 vorgestellten Funktionen der *Random*-Komponente sind hierbei problematisch.

Angenommen die *add_interrupt_randomness*-Funktion wird zur Aggregation von Zufall verwendet. Dann hat ein Angreifer direkten Einfluss auf den Zufall des Linux-Kernels, wenn er Interrupt-Timings kontrolliert. Der Grund dafür ist, dass *add_interrupt_randomness* Interrupt-Timings als Zufall zum Füllen des Entropy-Pools heranzieht. Die drei restlichen Funktionen weisen ähnliches Verhalten auf. Da die Angreifer-Definition in Kap. 3 Fähigkeiten einschließt, die bei Verwendung der vier Funktionen in Lst. 1 eine Manipulation des Entropy-Pools erlauben, sind diese vier Funktionen nicht für die Umsetzung des Systems geeignet.

Mit *add_hwgenerator_randomness* sind solche Manipulationen nicht möglich. Anders als bei den vier manipulierbaren Funktionen, wird bei dieser Funktion der Zufall direkt durch einen Hardware-Zufallsgenerator erzeugt. Konkret bedeutet das, ein Bitstrom wird vom Generator an den Entropy-Pool weitergereicht und nicht aus Timings oder anderen manipulierbaren Größen abgeleitet.

Die erste Maßnahme zur Umsetzung von Schritt Zwei besteht darin, alle Funktionen zur Aggregation von Zufall zu deaktivieren, abgesehen von der Funktion *add_hwgenerator_randomness*. Dafür müssen lediglich die Implementierungen der zu deaktivierenden Funktionen auskommentiert werden. Da es sich bei dieser Änderung um eine triviale Maßnahme handelt, wird an dieser Stelle keine Abbildung gezeigt.

Die nächste Maßnahme besteht darin, Zufall in Form eines Bitstroms an die QEMU-Instanzen zu übergeben. Dafür bietet QEMU die Möglichkeit eine Datei über die Kommandozeile zu spezifizieren, die als Quelle für einen Hardware-Zufallszahlengenerator dient. Dieser Zufallszahlengenerator steht einem emulierten Linux in Form der Datei */dev/hwrng* zur Verfügung. Um den somit gewonnenen Zufall zu verwendet, ist keine Konfiguration nötig, da der Kernel diesen automatisch verwendet. Die Funktion, die diese Aufgabe übernimmt, ist die *hwrng_fillfn* die in Kap. 2.1.2 in Lst. 3 vorgestellt wurde. Die QEMU-Optionen zur Erstellung eines Hardware-Zufallszahlengenerators sind in Abb. 11 zu sehen. Ähnlich wie bei der Netzwerkfunktionalität gibt es auch hier eine Zweiteilung. Mit der *-object rng-random*-Option wird das Backend für den Zufallszahlengenerator erstellt. Dieses hat die Aufgabe den Bitstrom einzulesen. Wie zuvor erwähnt, geschieht dies durch Verknüpfung mit einer Datei, die in diesem Fall *hwrng.bin* heißt. Der zweite Teil, das virtual Nertwork Device, wird mit der Option *-device virtio-rng-pci* erzeugt. Verbunden werden die beiden Bestandteile über die *id* des Backends, das von der Zusatzoption *rng* des virtuellen Geräts referenziert wird. Di Zusatzoptionen *max-bytes* und *period* dienen lediglich dazu die Lesegeschwindigkeit von QEMU bzw. des darin emulierten Linux zu beschränken. In diesem Fall dürfen jede Sekunde maximal 1024 Bytes aus dem Bitstrom gelesen werden.

```
1 qemu-system-x86_64 \
2     -object rng-random,filename=hwrng.bin,id=rng0 \
3     -device virtio-rng-pci,rng=rng0,max-bytes=1024,period=1000 \
```

Listing 11: Erstellen eines Hardware-Zufallszahlengenerators mit QEMU

Ein Vorteil der beschriebenen Vorgehensweise, eine Datei als Quelle für Zufall zu verwenden, ist, dass bei der Entwicklung des Gesamtsystems immer die gleiche Eingabe verwendet werden kann. Dafür ist es lediglich nötig eine ausreichend große Menge an zufälligen Bits in einer Datei zu speichern, die bei unterschiedlichen Durchläufen als Zufallsquelle mit den QEMU-Instanzen verbunden wird. Zum erreichen der Sicherheitsziele ist dies jedoch nicht ausreichend. Dazu muss bei jeder Ausführung neuer Zufall bereitstehen und keine Partei darf, unabhängig von anderen Parteien, dessen Zusammensetzung bestimmen können. Der Weg zum Erreichen dieses Ziels ist eine Erweiterung des Aufbaus in Abb. 13, der eine Aushandlung des Zufalls zwischen den QEMU-Instanzen ermöglicht. Diese Erweiterung wurde in Abb. 15 hinzugefügt. Neben den Bestandteilen die bereits für die Implementierung des Ereignisstroms vorhanden sind, kommen auf beiden Systeme jeweils die Datei *hwrng.bin* und das *rng-feeder*-Skript hinzu. Wie auch *stream.bin* ist *hwrng.bin* ebenfalls eine FIFO-Datei. Gelesen wird diese jeweils von der QEMU-Instanz, die auf den Systemen läuft. Für die Befüllung mittels Bitstrom ist das *rng-feeder*-Skript verantwortlich. Zuvor handeln die beiden *rng-reeder*-Skripte den Zufall jedoch erst aus.

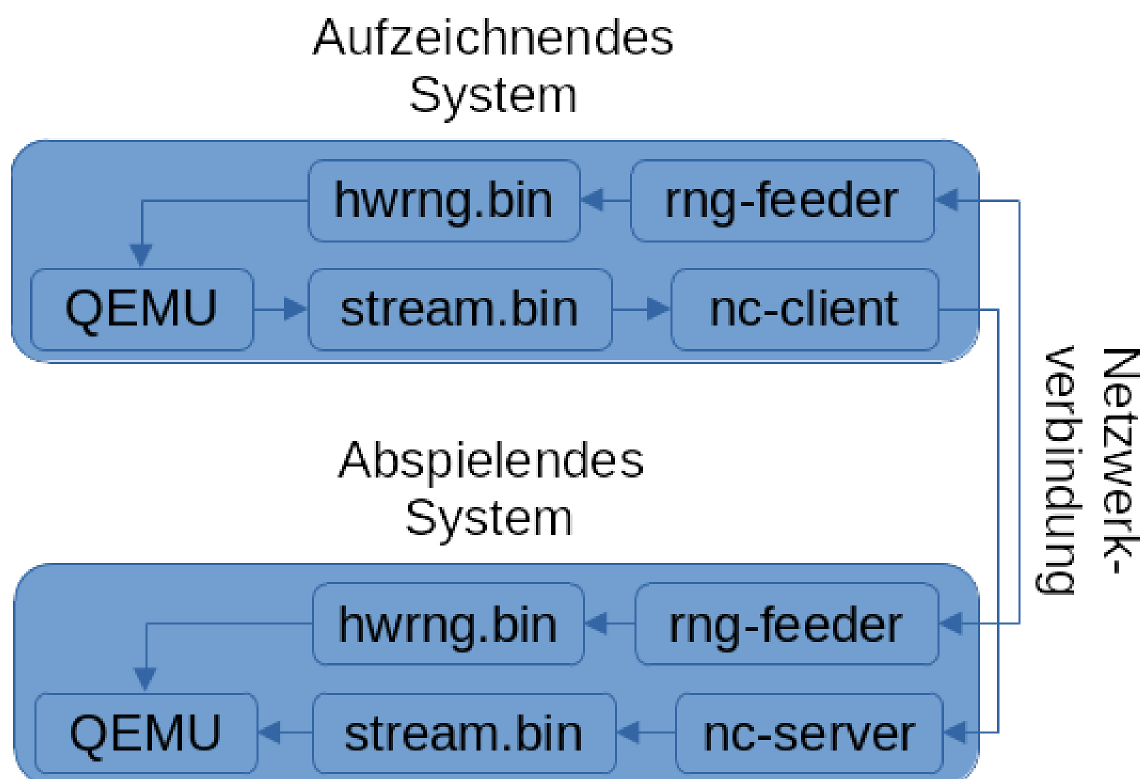

Abbildung 15: Erweiterung des Aufbaus um Zufall auszuhandeln

Das Verfahren zur Aushandlung des Zufalls nimmt das Coin-Flipping-Protokoll von Manuel Blum als Ausgangspunkt. Ziel dieses Verfahrens ist der Austausch von Zufall zwischen zwei Kommunikationspartnern die sich gegenseitig nicht Vertrauen. Eine weitere Besonderheit ist, dass beide Parteien nur indirekt kommunizieren. Ein konkretes Beispiel, um das Problem und die Relevanz des Verfahrens zu verdeutlichen, ist der Münzwurf über ein Telefon, wie Blum selbst ihn in [2] vorstellt. In der darin beschriebenen Situation wollen zwei Parteien, Alice und Bob, eine Münze werfen, um eine Entscheidung zu treffen. Die Besonderheit dabei ist, dass sie sich nicht am gleichen Standort befinden und deshalb über Telefon kommunizieren. Der Münzwurf kann deshalb nur von einer Partei beobachtet werden. Dabei tritt das Problem auf, dass eine Partei den Ausgang des Münzwurfs manipulieren kann, falls das Ergebnis nicht deren Wunsch entspricht.

Sieht die Kommunikation zwischen Alice und Bob so aus, dass Alice zuerst ein Ergebnis vorhersagt und Bob daraufhin die Münze wirft, dann kann Bob ein falsches Ergebnis kommunizieren, ohne dass Alice dies bemerkt. Alice müsste in diesem Fall Bob vertrauen, dass er nicht lügt. Eine weitere Möglichkeit ist, dass Alice ihre Vermutung erst nach dem Münzwurf mitteilt, nachdem Bob das Ergebnis des Münzwurfs kommuniziert hat. Auch in diesem Fall ist eine Manipulation möglich, indem Alice das von Bob kommunizierte Ergebnis als ihres ausgibt, selbst wenn sie eine andere Vermutung hatte, um den Münzwurf zu gewinnen. In beiden Fällen kann das Zufallsereignis durch eine Partei umgangen werden, um einen Vorteil zu erlangen. Diese beiden Szenarien sind nochmals grafisch in Abb. 16 dargestellt. Im ersten Fall, wenn Alice zuerst ihre Vermutung abgibt und Bob im Anschluss die Münze wirft, gewinnt Bob immer, wenn er auf die Vermutung von Alice mit genau dem anderen Wert antwortet, was durch das Ausrufezeichen kenntlich gemacht wird. Vermutet Alice beispielsweise *Kopf* als Ergebnis, übermittelt Bob *Zahl* als Ergebnis und gewinnt. Im zweiten Fall, wenn Erst Bob die Münze wirft und Alice dann die Vermutung abgibt, gewinnt Alice immer, indem sie das von Bob übermittelte Ergebnis als ihres ausgibt.

Der Weg, Manipulation zu vermeiden und einen Münzwurf über indirekte Kommunikation zu ermöglichen, besteht darin, kryptografische Hash-Funktionen zu verwenden, wie dies in Abb.

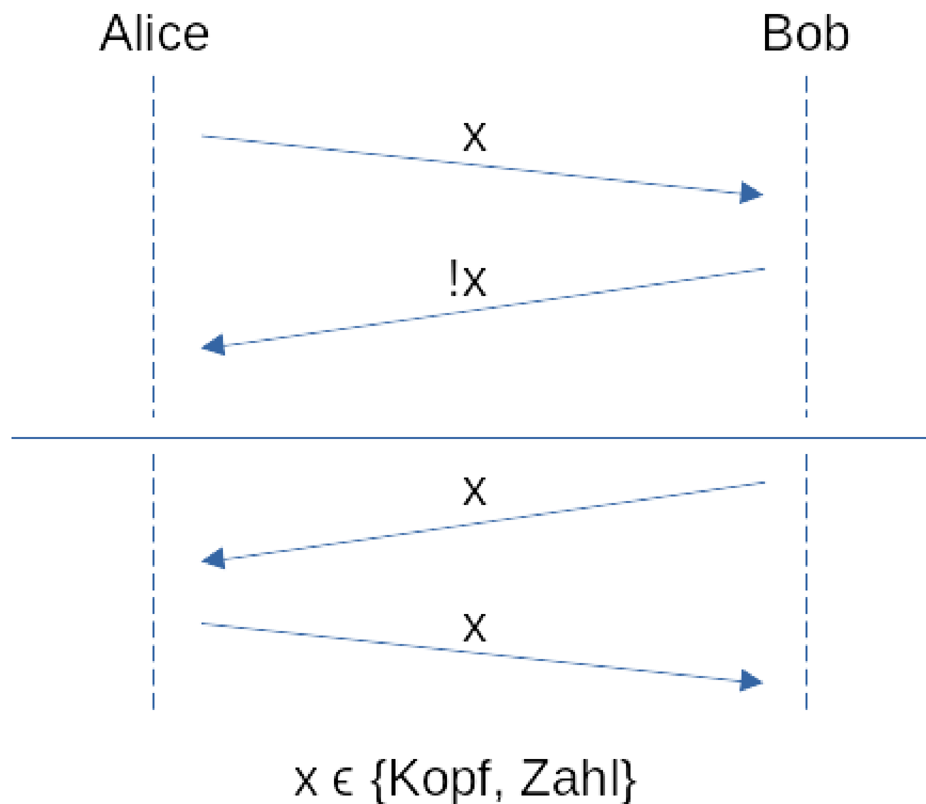

Abbildung 16: Unsicherer Münzwurf mit indirekter Kommunikation

17 zu sehen ist. Das Verfahren verändert sich dahingehend, dass ein Partei zu Beginn eine verbindliche Auswahl trifft. Diese verbindliche Auswahl ist das *Commitment*. Angenommen Alice übernimmt das Commitment, dann wählt sie zu Beginn eine beliebige Zahl aus. Mit dieser Zahl wird daraufhin, durch Verwendung einer Hash-Funktion, auf die sich Alice und Bob geeinigt haben, ein Hash erstellt. Der somit erzeugte Hash-Wert ist das Commitment und wird an Bob übermittelt. Dieser wiederum antwortet mit einer Vermutung, ob die Zahl gerade oder ungerade ist. Nachdem Bob die Vermutung preisgegeben hat, teilt Alice ihm die echte Zahl mit. Dadurch kann Bob die Hash-Funktion erneut ausführen und überprüfen, ob Alice die Wahrheit sagt. Stimmen der von Alice übermittelte Hash und der von Bob berechnete Hash überein, hat Alice nicht gelogen und keiner von beiden konnte das Zufallsereignis zu seinen Gunsten beeinflussen.

Die *rng-feeder* in Abb. 15 machen sich dieses Verfahren zu Nutze, um die Bitströme zu generieren, die in die *hwrng.bin*-Dateien geschrieben werden. Dafür wird der in Abb. 17 gezeigte Ablauf leicht angepasst. Anstatt einer einzelnen beliebigen Zahl, die von nur einer Partei gewählt wird, wählen beide Parteien ein beliebige Zahl, die über */dev/urandom* bezogen wird. Zudem Verknüpfen beide Parteien, im Anschluss an die Übertragung, diese Zufallszahlen durch eine XOR-Operation. Dadurch wird ein Seed für einen Zufallsgenerator erzeugt, der es ermöglicht auf beiden Systemen den gleichen Bitstrom zu generieren.

Der konkrete Ablauf sieht damit wie folgt aus: Alice und Bob wählen jeweils eine zufällige Zahl, Alice erzeugt einen Hash dieser Zahl und schickt diesen an Bob. Dieser Antwortet mit seiner eigenen Zufallszahl und erhält von Alice ebenfalls ihre Zufallszahl. Sobald Alice und Bob beide Zufallszahlen erhalten haben, überprüft Bob, ob Alice die richtige Zahl übermittelt hat, indem er den Hash erneut berechnet und mit dem Hash von Alice vergleicht. Hat Alice nicht gelogen, verwenden beide die XOR-Verknüpfung beider Zufallszahlen als Seed für einen Zufallszahlengenerator. Durch dieses Vorgehen wird sichergestellt, dass der Seed tatsächlich zufällig ist. Es besteht für beide Parteien keine Möglichkeit die eigene Zufallszahl in Abhängigkeit der Zufallszahl der anderen Partei zu wählen. Zudem steuern beide Parteien Zufall für die Berechnung des Seeds bei. Wegen der XOR-Verknüpfung ist der Seed selbst dann zufällig, wenn

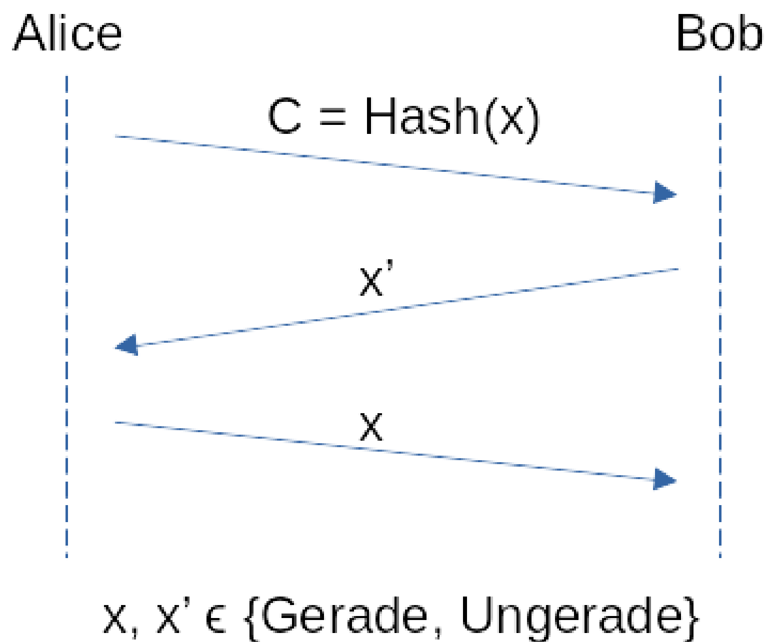

Abbildung 17: Sicherer Münzwurf mit indirekter Kommunikation

nur eine der beiden Parteien eine tatsächlich zufällig Zahl wählt. Diese Anpassungen sind nötig, da im eigentlichen Verfahren von Blum nur die Parität der Zahl von Bedeutung ist, jedoch kein Austausch einer Zufallszahl vorgesehen ist.

4.3 Schritt 3: Überprüfung ausgehender Pakete

Im dritten Schritt wird ebenfalls wieder der bereits erweiterte Aufbau aus dem vorherigen Schritt als Grundlage verwendet, um darauf aufbauend weitere Ergänzungen durchzuführen. Das Ziel an dieser Stelle ist es, den Aufbau so zu erweitern, dass die VMs mit dem Komparator-System verknüpft werden können. Der grundlegende Aufbau wurde bereits in Kap. 3 in Abb. 12 vorgestellt. Dafür müssen zum einen die QEMU-Instanzen auf eine spezielle Art konfiguriert werden und zum anderen QEMU selbst dahingehend erweitert werden, dass Netzwerkpakete auch bei der Wiedergabe über die Netzwerkverbindung empfangen werden können, anstatt Netzwerkpakete aus dem Ereignisstrom zu lesen. Außerdem muss der Komparator implementiert und das System, auf dem dieser ausgeführt wird, konfiguriert werden. All diese Aufgaben sind Teil dieses Kapitels und werden nach und nach vorgestellt.

Begonnen wird mit der Konfiguration der Network-Backends und der virtuellen Netzwerkkarten beider QEMU-Instanzen. Die dafür nötigen Kommandozeilen-Optionen sind in Lst. 12 abgebildet. An die virtuelle Netzwerkkarte besteht die Anforderungen, dass *virtio-net* nicht verwendet werden kann, da keine Hooks für die Replay-Komponente vorhanden sind. Deshalb muss eine virtuelle Variante einer physischen Netzwerkkarte verwendet werden. In diesem Fall ist dies *rtl8139*. Als Network Backend kommt das Socket-Backend zum Einsatz. Dies wird verwendet, da die QEMU-Instanzen im Zielzustand alle Netzwerkpakete direkt an das Komparator-System schicken sollen. Keine Paket soll direkt im Netzwerk verschickt werden können. Durch die Verwendung eins Socket-Backends kann bei der Erstellung der QEMU-Instanzen das Ziel über die IP-Adresse festgelegt werden. Im Falle des Listings ist dies der Localhost auf Port 4030.

```
1 qemu-system-x86_64 \
2     -netdev socket,id=net1,connect=localhost:4030 \
3     -device rtl8139,netdev=net1,mac=42:42:42:42:42:42 \
```

Listing 12: Netzwerk Konfiguration beider QEMU-Instanzen

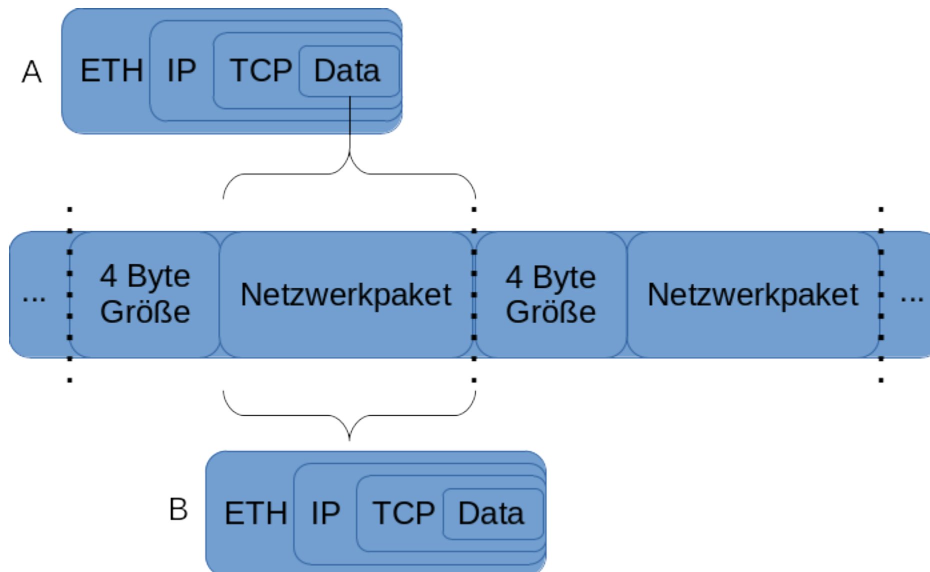

Abbildung 18: Aufbau des Socket-Backend-Protokolls

Der eigentliche Verwendungszweck des Socket-Backends besteht darin die Gastnetzwerke unterschiedlicher QEMU-Instanzen miteinander zu verbinden. Da hierfür lediglich eine Socket-Verbindung aufgebaut wird, ist ebenfalls die Anbindung des Komparators möglich. Für die Übertragung der Netzwerkpakete, die über diesen Socket versendet werden, wird ein simples Protokoll verwendet, das aus dem zu übertragenden Netzwerkpaket und dessen Länge besteht. Die Länge umfasst 4 Byte und wird vor dem Netzwerkpaket selbst gesendet. Abb. 18 verdeutlicht dies noch einmal. Zum einen ist hier zu erkennen, dass das Socket-Backend, in diesem Fall, eine TCP-Verbindung aufbaut. QEMU bietet jedoch die Möglichkeit User Datagram Protocol (UDP) zu verwenden. Zudem wird durch Paket A und Paket B verdeutlicht, dass es sich bei den Daten von Paket A ebenfalls um Netzwerkpakete handelt. Paket A ist ein Netzwerkpaket, das durch das Socket-Backend von QEMU versendet wird, dient also der Kommunikation zwischen den VMs und dem Komparator. Paket B hingegen ist ein Netzwerkpaket, dass von einem Linux Gast gesendet wird. Damit wird eine reguläre TCP-Verbindung mit dem Komparator aufgebaut, über die alle ausgehenden Netzwerkpakete der Gast-Systeme versendet werden.

Nach der oben beschriebenen Konfiguration der QEMU-Instanzen werden alle Pakete an die konfigurierte IP-Adresse weitergeleitet. Auf Seiten des Komparators ist der Empfang dieser Pakete durch öffnen eines Sockets möglich. Nach Abschluss des Verbindungsaufbaus besteht die einzige Aufgabe des Komparators darin, ausgehende und eingehende Netzwerkpakete von den bzw. an die VMs weiterzuleiten und zusätzlich ausgehende Pakete zu vergleichen. Deckt der Vergleich auf, dass zwei Pakete, die gleich sein sollten, dies nicht sind, dann wird die Ausführung gestoppt. Da die Aufgaben des Komparators überwiegend triviale Implementierungen besitzen, werden die entsprechenden Abschnitte des Quelltextes hier nicht im einzelnen vorgestellt.

Damit der Komparator Netzwerkpakete versenden kann, ist zusätzlich zur Implementierung die Konfiguration eines TAP-Device erforderlich. Im oben beschriebenen Zustand nimmt der Komparator zwar die ausgehenden Netzwerkpakete entgegen und kann eingehende an die VMs weiterleiten, jedoch besteht noch keine Verbindung zum Netzwerk. Ohne ein TAP-Device konfiguriert zu haben kann daher keine Netzwerkverbindung durch die VMs, über den Komparator, aufgebaut werden. Um ein solches zu erstellen sind die nötigen Befehle in Lst. 13 aufgeführt. Das TAP-Device selbst wird in Zeile 15 mit dem Namen *thesis_tap* erstellt. Zeile 16 dient der Aktivierung des TAP-Devices. Alle weiteren Zeilen werden dafür benötigt eine virtuelle Infrastruktur aufzubauen. Das Bridge-Device, das in Zeile 12 erstellt und in Zeile 13 aktiviert wird, entspricht einem virtuellen Netzwerk-Switch, der das TAP-Device und die physische Netzwerkkarte miteinander verknüpft. Für diese Verknüpfung sind die Zeilen 18 und 19 zuständig. Die verbleibenden Zeilen 10 und 21, sind dafür verantwortlich erst die IP-Adresse, die beim Start

des System durch den Dynamic Host Configuration Protocol (DHCP)-Client angefordert wurde, freizugeben und anschließend eine neue IP-Adresse anzufordern, um eine Netzwerkverbindung zu ermöglichen.

```
1  #!/bin/bash
2
3  if [[ -z ${1} ]]; then
4      echo "Device missing"
5      exit 1
6  else
7      declare -r DEVICE=${1}
8  fi
9
10 ip addr flush ${DEVICE}
11
12 ip link add thesis_br type bridge
13 ip link set thesis_br up
14
15 ip tuntap add dev thesis_tap mode tap
16 ip link set thesis_tap up
17
18 ip link set ${DEVICE} master thesis_br
19 ip link set thesis_tap master thesis_br
20
21 dhclient -v thesis_br
```

Listing 13: Konfiguration des TAP-Devices für den Komparator

Mit den bisherigen Erweiterungen des Aufbaus, ist die Kommunikations-Infrastruktur zwischen dem Komparator und den QEMU-Instanzen vollständig. Bisher kann jedoch nur die aufzeichnende QEMU-Instanz diese Infrastruktur nutzen. Deshalb besteht der letzte Schritt darin, QEMU so zu erweitern, dass die abspielende QEMU-Instanz die Infrastruktur ebenfalls verwenden kann. Hierfür ist erneut eine Betrachtung der Replay-Funktionalität von QEMU erforderlich.

Aus Sicht der aufzeichnenden QEMU-Instanz gibt es keine Änderungen, da diese wie vor der Einführung des Komparators Netzwerkpakete empfängt und sendet. Für die abspielende QEMU-Instanz gilt dies nicht. Bei der Wiedergabe einer Aufzeichnung werden grundsätzlich keine Netzwerkpakete über die Netzwerk-Komponente von QEMU empfangen oder versandt. Das liegt daran, dass eingehende Pakete bei der Aufzeichnung in den Ereignisstrom geschrieben und bei der Wiedergabe dort ausgelesen und wieder eingespielt werden. Pakete die möglicherweise tatsächlich während der Wiedergabe am System ankommen werden durch den Replay-Netzwerkfilter verworfen. Das gleiche gilt für ausgehende Pakete, auch diese werden durch den Replay-Netzwerkfilter verworfen.

Aus diesen Betrachtungen lassen sich die Maßnahmen ableiten, die nötig sind, um das abspielende System korrekt mit dem Komparator zu verknüpfen. Einerseits dürfen eingehende Pakete nicht verworfen werden. Anstatt verworfen zu werden, müssen diese entgegengenommen werden und die Pakete ersetzen, die über den Ereignisstrom eintreffen. Andererseits müssen ausgehende Pakete tatsächlich versandt werden, anstatt sie zu verwerfen. Dadurch gelangen diese zum Komparator und ermöglichen den Vergleich der Netzwerkpakete beider QEMU-Instanzen.

Der Weg diese Erweiterungen zu implementieren führt über die Netzwerk-Filter, die QEMU bereitstellt. Konkret betrifft dies den Netzwerk-Filter *filter-replay*, der der Standard-Filter ist, um die Replay-Funktionalität zu ermöglichen. Der Quelltext der die Filter-Funktionalität des Replay-Filters implementiert ist in Lst. 14 aufgeführt und unterscheidet zwischen drei Situation: QEMU wurde im Replay-Modus, im Record-Modus oder ohne Replay-Funktionalität gestartet. Im Record-Modus werden eingehende Pakete durch die Funktion *replay_net_packet_event* in den Event-Log gespeichert und ausgehende Pakete unverändert übertragen. Der Replay-Modus hingegen verwirft alle Pakete und für den Fall, dass die Replay-Funktionalität nicht aktiviert

wurde, werden alle Pakete unverändert weitergeleitet. Die Entscheidung, ob weitergeleitet oder verworfen werden soll, wird über den Rückgabewert des Filters getroffen. Ist dieser 0, wird weitergeleitet. Ist dieser nicht 0, wird verworfen. Der Grund für das Verwerfen von eingehenden Paketen im Record-Modus ist, dass die Pakete bei der Speicherung durch *replay_net_packet_event* automatisch weitergeleitet werden, um diese mit Zusatzinformationen zu versehen.

```
1  /* net/filter-replay.c */
2  static ssize_t filter_replay_receive_iov(NetFilterState *nf,
       NetClientState *sndr, unsigned flags,const struct iovec *iov, int
       iovcnt, NetPacketSent *sent_cb) {
3      NetFilterReplayState *nfrs = FILTER_REPLAY(nf);
4      switch (replay_mode) {
5      case REPLAY_MODE_RECORD:
6          if (nf->netdev == sndr) {
7              replay_net_packet_event(nfrs->rns, flags, iov, iovcnt);
8              return iov_size(iov, iovcnt);
9          }
10         return 0;
11     case REPLAY_MODE_PLAY:
12         /* Drop all packets in replay mode.
13            Packets from the log will be injected by the replay module
               . */
14         return iov_size(iov, iovcnt);
15     default:
16         /* Pass all the packets. */
17         return 0;
18     }
19 }
```

Listing 14: Standard Replay Filter

Der Ausgangspunkt für die Umsetzung der verbleibenden Maßnahmen ist der Replay-Zweig der switch-Anweisung. Um ausgehende Pakete zu versenden muss lediglich der Rückgabewert für solche Pakete auf 0 gesetzt werden, wie dies in Zeile 9 von Lst. 15 der Fall ist. Damit ist eine der beiden Maßnahmen abgeschlossen. Die zweite Maßnahme, Pakete vom Komparator zu empfangen und einzuspielen, ist eine mehrstufige Änderung, die bei der Funktion *thesis_replay_add_packet* in Zeile 6 beginnt. Ähnlich wie im Record-Zweig aus Abb. 14 sind nur eingehende Pakete von der if-Bedingung betroffen, die ebenfalls am Ende des Blocks verworfen werden. Der Unterschied besteht darin, dass die Pakete nicht dem Replay-Log hinzugefügt werden, sondern von der Funktion *thesis_replay_add_packet* in einer Queue abgelegt werden.

```
1  /* net/filter-replay.c */
2  {
3      ...
4      case REPLAY_MODE_PLAY:
5          if (nf->netdev == sndr) {
6              thesis_replay_add_packet(iov, iovcnt);
7              return iov_size(iov, iovcnt);
8          }
9          return 0;
10     ...
11 }
```

Listing 15: Erweiterter Replay Filter

Sowohl die *thesis_replay_add_packet*-Funktion, als auch die Queue, wurden speziell für diese Arbeit der Replay-Komponente hinzugefügt. Zusätzlich zu diesen beiden Bestandteilen wurde

die Funktion *thesis_replay_get_packet* hinzugefügt. Die Prototypen von *thesis_replay_add_packet* und *thesis_replay_get_packet* sind beide in der Header-Datei *include/sysemu/replay.h* zu finden. Die Definitionen der Funktionen und der Queue befinden sich in *replay/replay.c* und sind in Lst. 16 dargestellt. Die Zeilen 2 bis 9 haben die Aufgabe die Queue zu definieren. Beim Datentypen der Queue handelt es sich um eine Tail-Queue, die fester Bestandteil von QEMU ist. Um eine solche Queue zu erstellen muss ein struct erstellt werden, das als Datentyp für die Elemente der Queue dient. Dieser ist in den Zeilen 2 bis 7 aufgeführt. Dort wird das *thesis_queue_entry* struct erstellt, das einen void-Pointer für die Daten der Netzwerkpakete und deren Größe enthält. Das *QTAILQ_ENTRY*-Macro wird von der Implementierung der Queue benötigt, um Elemente miteinander zu verknüpfen. In Zeile 9 wird dann lediglich die Queue mit dem Namen *thesis_queue* erstellt und initialisiert.

Verwendung findet die Queue in beiden der darunter abgebildeten Funktionen. Mit der Funktion *thesis_replay_add_packet* können der Queue Netzwerkpakete hinzugefügt und mit der Funktion *thesis_replay_get_packet* wieder entnommen werden. Die Funktionsweise entspricht dabei einem First in, First Out (FIFO)-Speicher, da beim Hinzufügen am Ende der Queue eingefügt und beim Entfernen das erste Element entnommen wird.

```
1  /* replay/replay.c */
2  struct thesis_queue_entry {
3      void *data;
4      size_t size;
5
6      QTAILQ_ENTRY(thesis_queue_entry) entries;
7  };
8
9  static QTAILQ_HEAD(, thesis_queue_entry) thesis_queue =
       QTAILQ_HEAD_INITIALIZER(thesis_queue);
10
11 void thesis_replay_add_packet(const struct iovec *iov, int iovcnt) {
12     struct thesis_queue_entry *entry = g_new(struct
           thesis_queue_entry, 1);
13     entry->data = g_malloc(iov_size(iov, iovcnt)),
14     entry->size = iov_size(iov, iovcnt),
15
16     iov_to_buf(iov, iovcnt, 0, entry->data, entry->size);
17
18     QTAILQ_INSERT_TAIL(&thesis_queue, entry, entries);
19 }
20
21 void thesis_replay_get_packet(uint8_t **buf, size_t *size) {
22     struct thesis_queue_entry *entry = QTAILQ_FIRST(&thesis_queue);
23     QTAILQ_REMOVE(&thesis_queue, entry, entries);
24
25     *buf = entry->data;
26     *size = entry->size;
27
28     g_free(entry);
29 }
```

Listing 16: Funktionen zur Handhabung der vom Komparator gesendeten Pakete

Eingesetzt wird *thesis_replay_get_packet* in der Funktion *replay_event_net_load*, die Teil der Netzwerkfunktionalität der Replay-Komponente ist und in Lst. 17 gezeigt wird. Die eigentliche Aufgabe dieser Funktion besteht darin Ereignisse aus dem Event-Log zu lesen, was durch die Zeilen 3 bis 7 und 17 realisiert wird. Für die Erweiterung des Aufbaus wurde die if-Anweisung in den Zeilen 9 bis 15 hinzugefügt. Die Wirkung dieser Änderungen ist, dass die Daten, die bei

der Wiedergabe aus dem Ereignisstrom gelesen werden, durch die Daten ersetzt werden, die der Komparator übermittelt. Dafür ist es zum einen nötig, dass die aufzeichnende QEMU-Instanz weiterhin Pakete in den Ereignisstrom schreibt. Beim Aufzeichnen wird also die unveränderte QEMU-Funktionalität verwendet, obwohl die Daten nicht verwendet werden. Zum anderen ist die Reihenfolge der eingehenden Netzwerkpakete wichtig. Da der Komparator alle Pakete in der gleichen Reihenfolge an die VMs überträgt, wird diese dadurch implizit eingehalten. Aus diesem Grund ist die Ersetzung der Daten in Lst. 17 möglich, ohne weitere Überprüfungen durchzuführen.

```
1  /* replay/replay-net.c */
2  void *replay_event_net_load(void) {
3      NetEvent *event = g_new(NetEvent, 1);
4
5      event->id = replay_get_byte();
6      event->flags = replay_get_dword();
7      replay_get_array_alloc(&event->data, &event->size);
8
9      if(replay_mode == REPLAY_MODE_PLAY) {
10         size_t size = event->size;
11         thesis_replay_get_packet(&event->data, &event->size);
12         if (size != event->size) {
13             fprintf(stderr, "Something went wrong with packet order")
               ;
14         }
15     }
16
17     return event;
18 }
```

Listing 17: Injizieren der vom Komparator gesendeten Pakete

4.4 Vollständiger Aufbau

In diesem Kapitel wird die erweiterte Form des Zielsystems aus Abb. 12 vorgestellt und auf das Szenario eingegangen, das zum Tests des Aufbaus eingesetzt wurde. Das erweiterte Zielsystem ist in Abb. 12 dargestellt und enthält neben den bereits in Abb. 12 gezeigten Komponenten, alle Komponenten die im Verlauf dieses Kapitels hinzu kamen.

Am unteren Ende des Schaubildes ist das aufzeichnende System angesiedelt. Dort wird die QEMU-Instanz ausgeführt, die die Aufzeichnung des emulierten Gastsystems anfertigt. Dabei werden alle nicht-deterministischen Ereignisse, die auftreten, in *Event.fifo* geschrieben und über das *NC-Record*-Skript an das abspielende System gesendet. Zusätzlich wird über den *RNG-Feeder* Zufall mit dem abspielenden System ausgehandelt und der QEMU-Instanz über *RNG.fifo* zur Verfügung gestellt. Jegliche Kommunikation die innerhalb des Gastsystems stattfindet wird direkt von der QEMU-Instanz an den Komparator gesendet.

Die mittlere Komponente ist das abspielende System und ähnelt vom grundlegende Aufbau stark dem aufzeichnenden System. Wie auch auf dem aufzeichnenden System, läuft hier eine QEMU-Instanz, die anstatt aufzuzeichnen für die Wiedergabe der Aufzeichnung verantwortlich ist. Hierfür werden die Ereignisse, die während der Aufzeichnung auftreten, vom *NC-Replay*-Skript entgegengenommen und über *Event.fifo* an die QEMU-Instanz weitergereicht. Auf der Seite des *RNG-Feeder*-Skripts gibt es keine wesentlichen Unterschiede. Der einzige Unterschied ist, dass einer der beiden RNG-Feeder als Server und der andere als Client agiert. Sobald die Aushandlung des Zufalls zwischen den beiden RNG-Feedern abgeschlossen ist, verwendet das abspielende System ebenfalls *RNG.fifo*, um den Bitstrom an die QEMU-Instanz weiterzuleiten. Auch bei der Kommunikation mit dem Komparator gibt es keine wesentlichen Unterschiede.

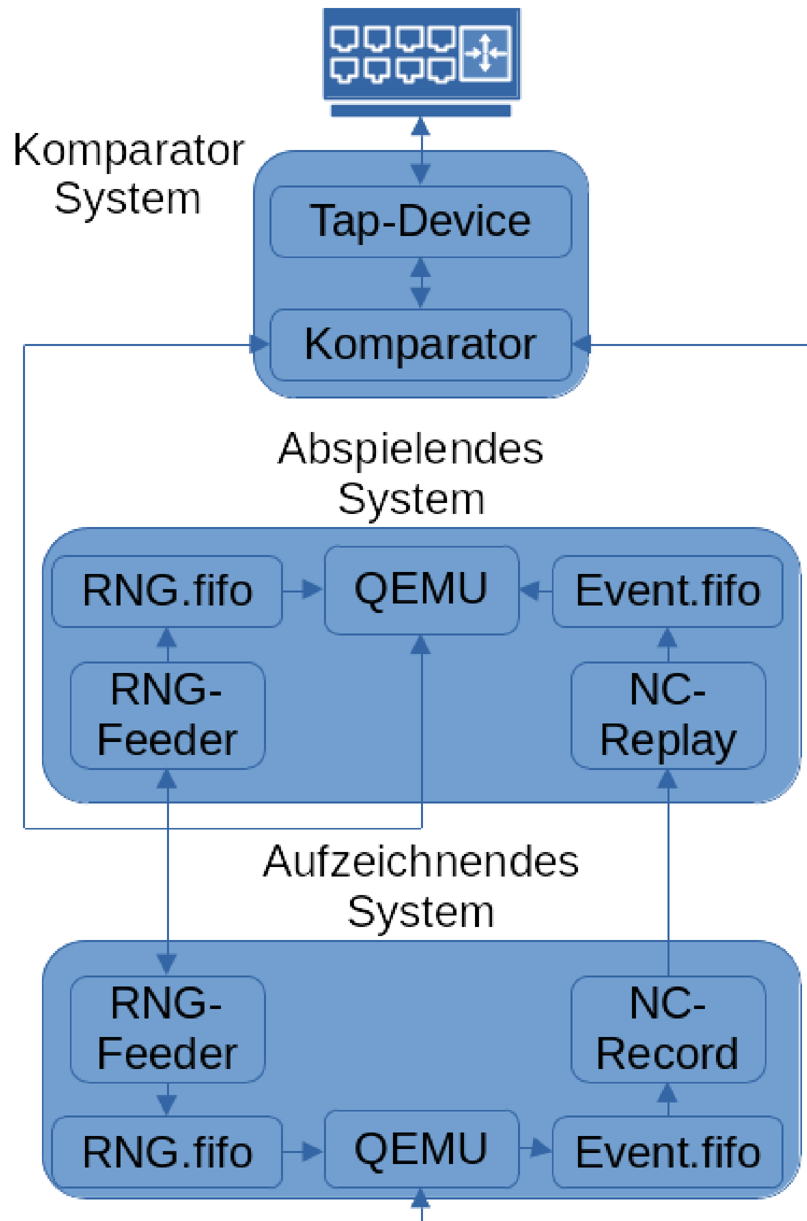

Abbildung 19: Gesamtsystem mit allen Komponenten

Hier werden die Pakete des Gastsystems ebenfalls durch die QEMU-Instanz an den Komparator übermittelt.

Das Komparator-System als kleinste Komponente übernimmt im Wesentlichen die Weiterleitung der Pakete, die von den beiden QEMU-Instanzen gesendet werden. Ebenso werden eingehende Pakete vom Komparator an die QEMU-Instanzen weitergeleitet. Die Komponente, die die Weiterleitung und damit einen Verbindungsaufbau der QEMU-Instanzen zum Netzwerk ermöglicht, ist das *TAP-Device*. Dieses agiert indirekt als Netzwerkkarte für die QEMU-Instanzen. Die Verbindung zwischen den QEMU-Instanzen und dem Komparator, über das Socket-Backend, dient somit als Tunnel und verbindet die QEMU-Instanzen mit der Netzwerkkarte. Die letzte verbleibende Aufgabe des Komparators ist der Vergleich ausgehender Pakete. Sobald über den Tunnel Pakete eintreffen, werden alle Pakete der QEMU-Instanzen auf deren Gleichheit und Reihenfolge überprüft. Im Falle einer Abweichung, wird die Ausführung abgebrochen.

Zur Überprüfung des Gesamtsystems wurde ein Szenario gewählt bei dem ein SSH-Server zum Einsatz kommt. Dieser wird von den VMs aufgezeichnet bzw. abgespielt und empfängt über die Verbindung zum Netzwerk, die ganz oben in Abb. 19 durch das Router-Symbol gekennzeich-

net ist, Anfragen von einem SSH-Client. Sobald das aufzeichnende System den Boot-Prozess abgeschlossen hat und der SSH-Server bereit ist, wird von einem vierten System eine Verbindung aufgebaut. Über diese Verbindung wird eine Datei erstellt und im Anschluss die Verbindung durch den SSH-Client wieder beendet. Daraufhin überprüft der Server, ob die Datei wirklich existiert und fährt das System herunter. Auf Seite des abspielenden Systems werden all diese Interaktionen Wiedergegeben, bis dieses ebenfalls herunter fährt.

Eine erfolgreiche Ausführung dieses Szenarios bedeutet, dass einerseits die beiden VMs einen vollständigen Durchlauf beenden. Hierfür muss jeweils das Herunterfahren abgeschlossen werden. Andererseits muss durch den Komparator sichergestellt werden, dass alle ausgehenden Pakete beider Maschinen gleich sind. Eine fehlgeschlagene Ausführung ist immer daran zu erkennen, dass nicht alle Systeme einen vollständigen Durchlauf beenden. Daraus folgt beispielsweise, dass die VMs in einer Endlosschleife feststecken, oder der Komparator eine Warnung ausgibt. Grund für die Endlosschleifen ist, dass QEMU nicht weiter ausgeführt werden kann, wenn sich die Zustände des aufzeichnenden und abspielenden Systems unterscheiden. Unterscheiden sich beispielsweise eingehende oder ausgehende Netzwerkpakete in Länge und Inhalt, kommt es zu einer Ungleichheit zwischen dem Zustand einer VM und auszuführenden Instruktionen. Dies muss nicht sofort erkannt werden, führt auf Dauer jedoch dazu, dass die Abläufe der VMs sich unterscheiden und QEMU in einer Endlosschleife endet.

Im Zuge dieser Arbeit wurde das oben beschrieben Zielszenario erfolgreich ausgeführt. Das bedeutet, es wurden alle Systeme wie beschrieben in Betrieb genommen und ausgeführt, sodass alle beteiligten Systeme vollständig und ohne Warnungen beendet wurden.

5 Zusammenfassung und Ausblick

Diese Arbeit zeigt die Umsetzung eines Verfahrens zur Überprüfung der Sicherheitsziele Integrity und Confidientality unter Verwendung von Record-/Replay-Funktionalität virtueller Maschinen. Dabei wird die Open Source Software QEMU als Grundlage verwendet, da diese deterministische Wiedergabe aufgezeichneter VMs ermöglicht. Zusammen mit Linux, das als Host- und als Gast-System für die VMs dient, wurde ein System realisiert, das ausgehende Netzwerkpakete von aufgezeichneten und abgespielten QEMU-Instanzen überprüft. Durch diese Überprüfung können die Sicherheitsziele gewährleistet werden, solange mindestens eine VM und das Komparator-System nicht korrumpiert sind.

Dafür wurde QEMU zunächst um Live-Replay-Funktionalität erweitert. Das bedeutet, dass die Record-/Replay-Funktionalität genutzt wird, um Aufzeichnungen wiederzugeben, ohne diese vollständig abgeschlossen zu haben. Dabei werden die durch QEMU aufgezeichneten nicht-deterministischen Ereignisse nicht nur in eine Datei geschrieben, sondern direkt an eine zweite QEMU-Instanz übertragen, die diese unmittelbar für die Wiedergabe nutzt.

Des Weiteren wurde die Zufallsgewinnung der Gastsysteme angepasst, um die Möglichkeit zu verhindern, dass Angreifer Zufall festlegen können. Hierfür waren Anpassungen am Linux-Kernel notwendig, die verhindern, dass ungeeigneter Zufall zum internen Entropy-Pool hinzugefügt werden kann. Zudem wurde ein Verfahren implementiert um in den VMs den gleichen Zufall zu Verfügung zu stellen. Dabei findet zunächst ein Austausch statt, der dazu dient ein Seed für einen Zufallsgenerator zu gewinnen. Der durch den Zufallsgenerator generierte Bitstrom wird im Anschluss verwendet, um die Gast-Systeme mit geeignetem Zufall zu versorgen.

Um den Gastsystemen einen Verbindungsaufbau zu Computernetzwerken zu ermöglichen, wurde QEMU dahingehend erweitert, dass ein Tunnel zu einem dritten System, dem Komparator, aufgebaut wird. Alle ausgehenden Pakete der QEMU-Instanzen werden über diesen Tunnel direkt an den Komparator gesendet. Dort werden sie zunächst verglichen und bei einem erfolgreichen Vergleich an das Netzwerk weitergeleitet. Zusätzlich wurde die Funktionalität hinzugefügt Netzwerkpakete in die VMs zu injizieren. Treffen am Komparator Pakete ein, werden diese über den Tunnel an die VMs weitergeleitet und dann von QEMU, entweder direkt verwendet oder in den Ereignisstrom injiziert. Für die Verbindung zum Netzwerk verwendet der Komparator ein TAP-Device, das als virtuelle Netzwerkkarte für die VMs dient.

Zu Überprüfung des Gesamtsystems wurde ein Experiment durchgeführt, bei dem eine Verbindung zu einem SSH-Server aufgebaut wird. Innerhalb der VMs wird dieser emuliert und ermöglicht den Verbindungsaufbau durch einen SSH-Client. Bei der Durchführung des Experiments wurden alle Bestandteile erfolgreich ausgeführt, sodass ein positiver Ausgang verzeichnet wurde.

Eine Eigenschaft die das System noch nicht realisiert, jedoch für einen Einsatz in Produktion notwendig ist, ist das Verwerfen von Eingabe-Ereignissen. Dies bezieht sich beispielsweise auf Eingaben mit Maus oder Tastatur. Im vorgestellten Aufbau sind diese noch möglich, sodass ein Angreifer die Möglichkeit hat beliebige Berechnungen durchzuführen. Für einen Einsatz in Produktion müssen Eingabegeräte durch QEMU oder Linux deaktiviert werden.

Da es sich bei dieser Arbeit um eine prototypische Umsetzung handelt, sind die durchgeführten Erweiterungen nicht in das offizielle QEMU Repository integriert. Um dies zu bewerkstelligen ist eine Überarbeitung der Implementierung erforderlich, um diese zu verallgemeinern. Bisher sind die Erweiterungen stark mit dem in dieser Arbeit vorgestellten Aufbau verknüpft.

Des Weiteren ist eine Verbesserung des Gesamtsystem denkbar, bei dem korrumpierte Systeme ersetzt oder abgekoppelt werden, anstatt die Ausführung lediglich zu beenden. In der aktuellen Umsetzung unterbricht der Komparator die Ausführung dadurch, dass bei einem Fehlgeschlagenen Vergleich keine Pakete mehr gesendet werden und gibt eine Warnung aus. Dadurch ist das Gesamtsystem nicht mehr nutzbar, bis es neu gestartet wird. Ebenso müssen korrumpierte Systeme händisch aus dem Gesamtsystem entfernt werden. Um diese manuellen Aufgaben zu vermeiden, besteht die Möglichkeit die Software so zu erweitern, dass einerseits physische Systeme automatisch vom Gesamtsystem abgekoppelt werden, falls durch den Komparator ein korrumpiertes System erkannt wird. Andererseits ist eine automatische Ersetzung korrumpierter

Systeme wünschenswert.

Eine weitere Einschränkung für das Gesamtsystem entsteht durch die Verwendung der Record-/Replay-Funktionalität. Da diese auf dem Instruction Counting des TCG aufbaut, kommt der Single-Threaded TCG zum Einsatz. Es existiert zwar ein Multi-Threaded TCG, jedoch ist Instruction Counting nicht mit diesem kompatibel. Zur Steigerung der Performanz des Gesamtsystems wäre deshalb die Implementierung des Instruction Counting für den Multi-Threaded TCG eine mögliches Thema für zukünftige Arbeiten.

Literatur

[1] Kaiwan N Billimoria. *Linux Kernel Programming*. Packt Publishing, 2021.

[2] *Coin Flipping by Telephone*. URL: https://www.cs.cmu.edu/~mblum/research/pdf/coin/ (besucht am 19. 07. 2022).

[3] *Dokumentation zum Kernel-Buildsystem*. URL: https://www.kernel.org/doc/html/latest/kbuild/index.html (besucht am 09. 06. 2022).

[4] *GDB Manual*. URL: https://sourceware.org/gdb/current/onlinedocs/gdb/ (besucht am 18. 06. 2022).

[5] *Git-Repository des Linux-Kernels*. URL: https://git.kernel.org/ (besucht am 05. 06. 2022).

[6] *GNU Autotools Dokumentation*. URL: https://www.gnu.org/software/automake/manual/html_node/index.html (besucht am 17. 06. 2022).

[7] *Header-Datei der HW-Random-Komponente in Version 5.15.30*. URL: https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/include/linux/hw_random.h?h=v5.15.30&id=0464ab17184b8fdec6676fabe76059b90e54e74f (besucht am 13. 06. 2022).

[8] *Header-Datei der Random-Komponente in Version 5.15.30*. URL: https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/include/linux/random.h?h=v5.15.30&id=0464ab17184b8fdec6676fabe76059b90e54e74f (besucht am 13. 06. 2022).

[9] *Intel Digital Random Number Generator*. URL: https://www.intel.com/content/dam/develop/external/us/en/documents/drng-software-implementation-guide-2-1-185467.pdf (besucht am 14. 06. 2022).

[10] J. Katz und Y. Lindell. *Introduction to Modern Cryptography*. CRC Press, 2021.

[11] *Mainline-Tree des Linux Kernels*. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/ (besucht am 05. 06. 2022).

[12] *Offizielle Dokumentation zu Tun-/Tap-Devices*. URL: https://www.kernel.org/doc/html/latest/networking/tuntap.html (besucht am 15. 06. 2022).

[13] *QEMU Haupt-Repository*. URL: https://gitlab.com/qemu-project/qemu (besucht am 17. 06. 2022).

[14] *QEMU Networking Dokumentation*. URL: https://wiki.qemu.org/Documentation/Networking (besucht am 17. 06. 2022).

[15] *QEMU Replay Dokumentation*. URL: https://www.qemu.org/docs/master/devel/replay.html (besucht am 28. 06. 2022).

[16] *QEMU Replay Feature Wiki*. URL: https://wiki.qemu.org/Features/record-replay (besucht am 17. 06. 2022).

[17] *ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay*. URL: https://www.usenix.org/legacy/events/osdi02/tech/full_papers/dunlap/dunlap.pdf (besucht am 11. 08. 2022).

[18] Yasser Shalabi u. a. „Record-Replay Architecture as a General Security Framework“. In: *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2018, S. 180–193. DOI: 10.1109/HPCA.2018.00025.

[19] *Source-Datei der HW-Random-Komponente in Version 5.15.30*. URL: https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/drivers/char/hw_random/core.c?h=v5.15.30&id=0464ab17184b8fdec6676fabe76059b90e54e74f (besucht am 13. 06. 2022).

[20] *Source-Datei der Random-Komponente in Version 5.15.30*. URL: https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/drivers/char/random.c?h=v5.15.30&id=0464ab17184b8fdec6676fabe76059b90e54e74f (besucht am 13. 06. 2022).

[21] *Stable-Tree des Linux Kernels*. URL: https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git (besucht am 03. 07. 2022).

[22] *TCG Dokumentation*. URL: https://www.qemu.org/docs/master/devel/tcg.html (besucht am 24. 06. 2022).

[23] *TCG Feature Wiki*. URL: https://wiki.qemu.org/Features/TCG (besucht am 24. 06. 2022).

[24] *TCG Frotend Ops Wiki*. URL: https://wiki.qemu.org/Documentation/TCG/frontend-ops (besucht am 24. 06. 2022).

[25] *TCG Instruction Counting Dokumentation*. URL: https://www.qemu.org/docs/master/devel/tcg-icount.html (besucht am 24. 06. 2022).

[26] *TCG Wiki*. URL: https://wiki.qemu.org/Documentation/TCG (besucht am 24. 06. 2022).

[27] Nuutti Varis. „Anatomoy of a Linux bridge“. 2012.

[28] Wei Wang u. a. „pRnR: A Parallel Record-Replay Framework for Virtual Machines“. In: *2020 IEEE 38th International Conference on Computer Design (ICCD)*. 2020, S. 610–618. DOI: 10.1109/ICCD50377.2020.00106.