technische universität
dortmund

Masterarbeit

Sicherheitsaspekte virtualisierter cyber-physischer Systeme

Majuran Rajakanthan
September 2017

Gutachter:
Prof. Dr.-Ing. Olaf Spinczyk
Dipl.-Inf. Boguslaw Jablkowski

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl 12
Arbeitsgruppe Eingebettete Systemsoftware
http://ess.cs.tu-dortmund.de

Inhaltsverzeichnis

1 Einleitung und Motivation 1
1.1 Zielsetzung der Arbeit . . . 2
1.2 Aufbau der Arbeit . . . 3

2 Grundlagen 5
2.1 Sicherheitsrisiken in Cyber-Physischen Systemen . . . 5
2.1.1 Typische Angriffsklassen . . . 6
2.2 Virtualisierung . . . 9
2.2.1 Virtualisierungsarchitekturen . . . 9
2.2.2 Xen Project Hypervisor . . . 10
2.3 Virtual Machine Introspection . . . 12
2.3.1 Semantic Gap . . . 13
2.3.2 LibVMI . . . 14
2.4 Software Defined Networking . . . 16
2.4.1 SDN Controller . . . 16
2.4.2 OpenFlow . . . 17

3 Stand der Forschung 19

4 Entwurf und Implementierung einer sicheren virtualisierten Infrastruktur 21
4.1 Virtualisierung der Ausführungsplattform . . . 22
4.1.1 Domain 0 Disaggregation . . . 22
4.1.2 Virtual Machine Introspection . . . 26
4.2 Virtualisierung der Kommunikation . . . 30
4.2.1 Software Defined Networking . . . 30
4.2.2 Erkennung und Abwehr von DoS-Attacken . . . 31
4.3 Sicherheitsaspekte der Virtualisierung . . . 36

5 Evaluation 43
5.1 Definition der Szenarien . . . 43
5.1.1 VMI Szenario . . . 44
5.1.2 SDN Szenario . . . 45

5.2 Evaluation der funktionalen Eigenschaften 46
5.2.1 VMI 46
5.2.2 SDN 48
5.3 Evaluation der nichtfunktionalen Eigenschaften 50
5.3.1 VMI 50
5.3.2 SDN 52

6 Fazit **57**

Abkürzungsverzeichnis **59**

Abbildungsverzeichnis **61**

Tabellenverzeichnis **63**

Quellcodeverzeichnis **65**

Literaturverzeichnis **67**

Erklärung **73**

1 Einleitung und Motivation

Cyber-Physische Systeme (*CPS*) sind komplexe Systeme, die mittlerweile in verschiedenen Anwendungsbereichen, wie Automotive Systeme, Produktionsanlagen oder Energienetze zum Einsatz kommen [Lee08]. *CPS* bestehen aus einem Verbund von Rechnereinheiten und physikalischen Prozessen. Ähnlich wie traditionelle Rechnersysteme sind auch solche *CPS* verwundbar. Auch wenn die Zahl der Angriffe in der Vergangenheit recht überschaubar gewesen ist, steigen die Sicherheitsrisiken in *CPS* [CAS08]. Heutzutage werden eingebettete Systeme und Mikrocontroller zum Überwachen von *CPS* genutzt. Diese sind zudem über Kommunikationsnetze miteinander verbunden. Daraus entstehen zahlreiche Sicherheitsrisiken, welche die Systeme immer angreifbarer machen. Im Gegensatz zu traditionellen Rechnersystemen können Angriffe auf *CPS* weitreichende Konsequenzen zur Folge haben, da physikalische Prozesse gefährdet sind. Beispielsweise wurde im Jahr 2000 das Abwasserkontrollsystem in Queensland, Australien, angegriffen [SM07]. Durch eine Störung der Kommunikation zwischen dem Kontrollzentrum und den Pumpen, wurden Flächen mit Abwasser geflutet. Da keine Sicherheitsmechanismen aktiv waren, wurde der Angriff erst erkannt, nachdem über eine Million Liter Abwasser in nahegelegenen Parks geflutet wurde. [CAS08]

Virtualisierung hat sich in den letzten Jahren stark etabliert. Sowohl in Unternehmen als auch im Privatbereich findet man zunehmend Virtualisierungslösungen. Auf Servern beispielsweise wird Virtualisierung zwecks der Konsolidierung von physischen Systemen eingesetzt. Durch die Ausführung von mehreren virtuellen Systemen auf einer physischen Hardware, kann diese somit besser ausgelastet werden, was zudem zu Kostenersparnissen führt. Durch die Isolation, die die Virtualisierung mit sich bringt, erhöht sich automatisch auch die Sicherheit der Systeme. Die virtuellen Maschinen können unabhängig voneinander arbeiten und bei einer Integritätsverletzung, wird der Schaden auf die anderen Systeme durch die Isolation verringert.

Es stellt sich daher die Frage, ob durch eine Nutzung von virtuellen Maschinen, zum Überwachen von CPS, die Sicherheit erhöht werden kann. Neben den Vorteilen wie Verfügbarkeit und Fehlertoleranz, die durch die Virtualisierung gegeben ist, müssen allerdings auch die Sicherheitsrisiken durch das Kommunikationsnetz betrachtet werden. Mithilfe einer durchgängig virtualisierten Infrastruktur für die Ausführungsplattform und das Kommunikationsnetz, können die Vorteile der Virtualisierung auch im Kommunikationsnetz genutzt werden, wodurch die Sicherheit in beiden Bereichen erhöht wird. So eine

Infrastruktur, zum Überwachen und Verwalten von speicherprogrammierbaren Steuerungen (*Programmable Logic Controller (PLC)*) in CPS, kann wie in 1.1 abgebildet aussehen.

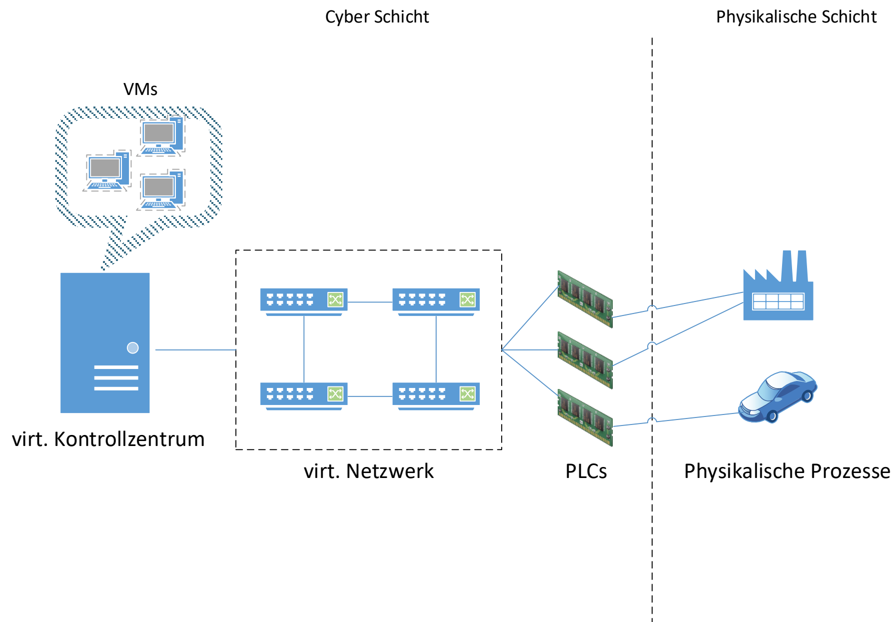


Abbildung 1.1: Virtualisierte cyber-physische Systeme.

1.1 Zielsetzung der Arbeit

Das Ziel dieser Masterarbeit ist die Untersuchung von Sicherheitsaspekten einer vollständig virtualisierten Infrastruktur für die Ausführungsplattform und das Kommunikationsnetz, um zu überprüfen ob durch die Virtualisierungstechnik die Voraussetzungen, wie Fehlertoleranz und Sicherheit, für einen zuverlässigen Betrieb einer Infrastruktur für CPS gegeben sind. Diese Arbeit beschäftigt sich dabei hauptsächlich mit den Sicherheitsaspekten. Dafür wird eine durchgängig virtualisierte Infrastruktur für die Ausführungsplattform und das Kommunikationsnetz entworfen und implementiert. Dabei sollen die existierenden Sicherheitsrisiken in cyber-physischen Systemen analysiert und geeignete Sicherheitsmaßnahmen, die auf der Virtualisierungstechnologie basieren, konzipert werden. Dazu werden Techniken und Konzepte wie *Virtual Machine Introspection* und *Software Defined Networking* verwendet. Durch eine Kombination bereits vorhandener Sicherheitslösungen soll die Sicherheit der virtualisierten Infrastruktur, zum Schutz der Arbeitsabläufe in CPS, erhöht werden. Die Sicherheitsaspekte durch Nutzung der Virtualisierungstechnik und dadurch entstehende Performanceeinbußen werden ebenfalls in dieser Arbeit betrachtet.

1.2 Aufbau der Arbeit

Im kommenden Kapitel *Grundlagen* wird zunächst ein Überblick der allgemeinen Sicherheitsrisiken in cyber-physischen Systemen gegeben. Anschließend werden die Grundbegriffe der Virtualisierung erläutert und die verschiedenen Virtualisierungsarchitekturen vorgestellt. Die darauffolgenden Abschnitte behandeln Technologien und Konzepte, welche die Sicherheit von virtualisierten Umgebungen und des Kommunikationsnetzes erhöhen können. Kapitel drei fasst den aktuellen Forschungsstand zusammen. In Kapitel vier wird eine virtualisierte Infrastruktur mit Sicherheitsmaßnahmen entworfen und anschließend implementiert. Dort wird auch auf die Sicherheitsaspekte durch Nutzung der Virtualisierung und den verschiedenen Technologien eingegangen. Im fünften Kapitel werden Szenarien für die auszuführenden Angriffe entwickelt und das Ziel dieser Szenarien definiert. Mit Hilfe dieser Szenarien werden anschließend die implementierten Sicherheitsmaßnahmen der virtualisierten Infrastruktur funktional und nichtfunktional evaluiert. In Kapitel sechs folgt eine abschließende Bewertung zu den Sicherheitsaspekten virtualisierter cyber-physischer Systeme.

2 Grundlagen

In diesem Kapitel wird zunächst auf die Sicherheitsrisiken in Cyber-Physischen Systemen eingegangen. Dabei werden die typischen Angriffsklassen in solchen Systemen aufgezählt. Daraufhin werden Grundbegriffe der Virtualisierung erklärt und es wird eine Übersicht der verschiedenen Virtualisierungsarchitekturen aufgezeigt. Die letzten zwei Abschnitte behandeln die in dieser Arbeit verwendeten Technologien, um die anfangs genannten möglichen Angriffe abzuwehren.

2.1 Sicherheitsrisiken in Cyber-Physischen Systemen

Cyber-Physische Systeme bestehen gewöhnlich aus mehreren Komponenten, wie Sensoren, Aktuatoren (Antriebselemente), Überwachungs- und Kontrolleinheiten, die über ein Kommunikationsnetz miteinander verbunden sind [CAS08]. Dadurch können sich verschiedene Angriffsmöglichkeiten auf ein CPS ergeben. Wenn beispielsweise die Kommunikation gestört wird, sodass Messdaten das Ziel nicht erreichen, kann der Systemablauf gefährdet werden. Durch eine Manipulation von Sensordaten, wodurch falsche Daten an das Ziel gesendet werden, kann das System ebenfalls kompromittiert werden.

Ein Anwendungsbeispiel (entnommen aus [ATKF13]) für Cyber-Physische Systeme sind Wasserverteilungssysteme, wie in Abbildung 2.1 dargestellt. Das Beispielszenario zeigt drei

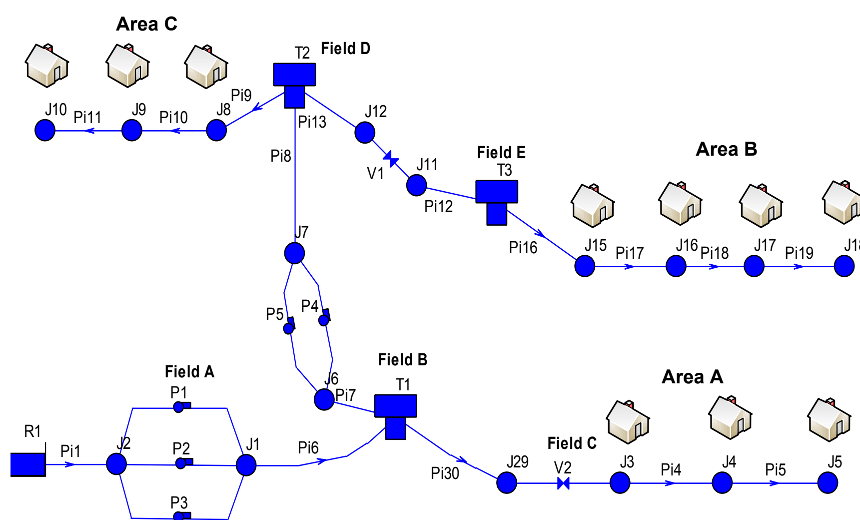


Abbildung 2.1: Beispielszenario eines Wasserverteilungssystems. [ATKF13]

Gebiete *A*, *B* und *C*, die durch drei Tanks *T1*, *T2* und *T3* mit Wasser versorgt werden. Die Pumpen *P1* - *P5* beliefern die beiden Tanks *T1* und *T2* mit Wasser, während *T3* das Wasser von *T2*, aufgrund der erhöhten Lage, erhält. [ATKF13] Abbildung 2.2 zeigt ein CPS für die Überwachung und Verwaltung dieses Wasserverteilungssystems. Das CPS be-

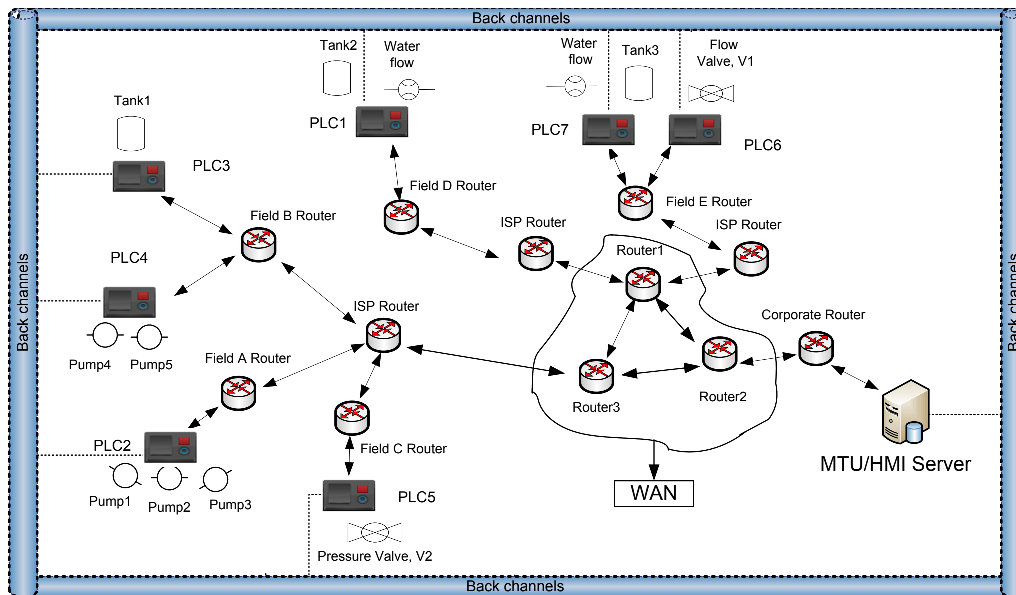


Abbildung 2.2: Cyber-Physisches System für das Wasserverteilungssystem. [ATKF13]

steht aus speicherprogrammierbaren Steuerungen (*Programmable Logic Controller (PLC)*) für die Sensoren und Aktuatoren, einer Überwachungs- und Verwaltungsinfrastruktur (*Master Terminal Unit (MTU) Server*) für die *PLCs* und einer Netzwerkinfrastruktur für die Kommunikation untereinander. Durch das Senden von Nachrichten an die *PLCs*, werden die Aktuatoren der Pumpen aktiviert. Der Wasserstand in *T1* wird durch *PLC3* gelesen, um entsprechende Nachrichten an *PLC2* für das Aus- und Einschalten der Pumpen senden zu können. Analog dazu wird der Wasserstand in *T2* durch *PLC1* gelesen und entsprechende Nachrichten an *PLC4* gesendet. Der Wasserfluss zwischen *T2* und *T3* wird automatisch durch ein Zuflussventil (*V1*) reguliert, damit sich in beiden Tanks genügend Wasser zur Versorgung der Bereiche befindet. Dafür wird eine Nachricht von der *MTU* an *PLC6* gesendet.

Im Folgenden wird eine Übersicht der typischen Angriffsklassen gegeben und die Konsequenz dieser Angriffe auf das erwähnte CPS betrachtet.

2.1.1 Typische Angriffsklassen

Die drei wesentlichen Anforderungen in der IT-Sicherheit, um Daten zu schützen, sind Vertraulichkeit, Integrität und Verfügbarkeit [TPSJ12] [Bis02]. Vertraulichkeit bedeutet, dass Daten nur von autorisierten Nutzern gelesen werden können. Integrität umfasst die Korrektheit der Daten. Mit Verfügbarkeit wird der rechtzeitige Zugriff auf Daten bezeichnet.

Angriffe auf eine dieser Klassen, können weitreichende Konsequenzen im Cyber-Physischen System zur Folge haben. Durch *Man-In-The-Middle (MITM)* Angriffen können Daten von nicht autorisierten Nutzern gelesen werden, indem die Kommunikation zwischen zwei Systemen abgefangen wird. Dadurch wird die Vertraulichkeit des Systems gefährdet. Allerdings wird der Systemablauf nicht beeinflusst [TPSJ12]. Mit Angriffen auf die Integrität oder Verfügbarkeit hingegen, wird der Systemablauf und dadurch physikalische Prozesse in einem CPS gefährdet.

Angriff auf Integrität

Diese Art von Angriff erfordert meistens Vorkenntisse über das anzugreifende System und sind möglicherweise schwer zu entdecken [TPSJ12]. Durch *Malware* und *Rootkits* kann unbemerkter Zugriff auf die Überwachungs- und Verwaltungsinfrastruktur erlangt werden und dadurch Daten manipuliert werden, wodurch die Integrität der Infrastruktur verletzt wird. Als *Rootkits* werden Programme bezeichnet, die sich selbst und laufende andere Prozesse, wie schädliche *Malware*, vor dem System verbergen können. Darüber hinaus sind sie in der Lage dem Nutzer jederzeit erhöhte Rechte auf dem System zu gewähren. Unter Linux sind diese *Rootkits* als *LKM (Loadable Kernel Module) Rootkits* bekannt, da sie als Kernel Modul geladen werden und Teile des Kernels modifizieren. Daher laufen sie auch im *Kernelspace* mit den höchsten Rechten. Dies erlaubt ihnen *System Calls* zu manipulieren, wodurch sie schwer zu entdecken sind. In dem oben erwähnten Wasserverteilungssystem kann durch unerlaubten Zugriff auf die *MTU*, beispielsweise manipulierte Nachrichten an *PLCs* gesendet werden, die für die Steuerung der Pumpen verantwortlich sind. Dadurch wird erreicht, dass die Tanks nicht mehr mit genügend Wasser zur Versorgung der Gebiete beliefert werden. Der Wasserstand in den Tanks erreicht erst nach einer gewissen Zeit, nachdem der Angriff durchgeführt wurde, einen niedrigen Wert. Dies führt dazu, dass solche Angriffe schwerer zu entdecken sind aufgrund der zeitverzögerten Wirkung. [ATKF13]

Angriff auf Verfügbarkeit

Die Gefährdung der Verfügbarkeit eines Systems, wird durch eine Überlastung der Kommunikationsinfrastruktur erreicht, sodass die Erreichbarkeit nicht mehr gewährleistet ist. Dies geschieht durch sogenannte *Denial-of-Service (DoS)* Angriffe. Eine Klasse von *DoS*-Angriffen sind *Flood* Angriffe, wie *Ping Flood* oder *SYN Flood*. Dabei wird durch das Fluten vieler Pakete, das Ziel überlastet. *Ping Flood* Angriffe nutzen dafür *ICMP (Internet Control Message Protocol)*, um sehr viele Ping-Anfragen an das Ziel zu senden, welche ebenfalls Antworten zurück senden. Dadurch wird das Ziel überlastet und ist nicht mehr erreichbar. *SYN Flood* Angriffe nutzen *TCP (Transmission Control Protocol)*. Das Ziel hierbei ist, durch viele *TCP* Verbindungsaufbauten das System zu überlasten und die Verfügbarkeit zu gefährden. Abbildung 2.3 zeigt einen gewöhnlichen TCP-Verbindungsaufbau. Der

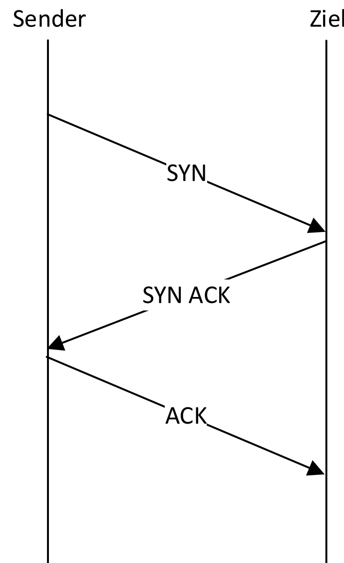


Abbildung 2.3: TCP Handshake.

Sender sendet dem Ziel ein Paket mit dem Synchronisations-Flag (SYN), um eine Verbindung aufzubauen. Dieser sendet dem Sender ein Paket mit dem Synchronisations-Flag und Bestätigungs-Flag (ACK), um den Verbindungsaufbau zu bestätigen. Wenn nun der Sender dem Ziel ebenfalls ein ACK sendet, ist der Verbindungsaufbau erfolgt. Bei einem *SYN Flood* Angriff werden parallel mehrere Verbindungen vom Angreifer eröffnet. Dieser sendet allerdings, wie in 2.4 abgebildet, sein letztes ACK nicht, damit die Verbindung erfolgreich hergestellt werden kann. Dadurch entstehen beim Ziel mehrere halboffene Verbindungen,

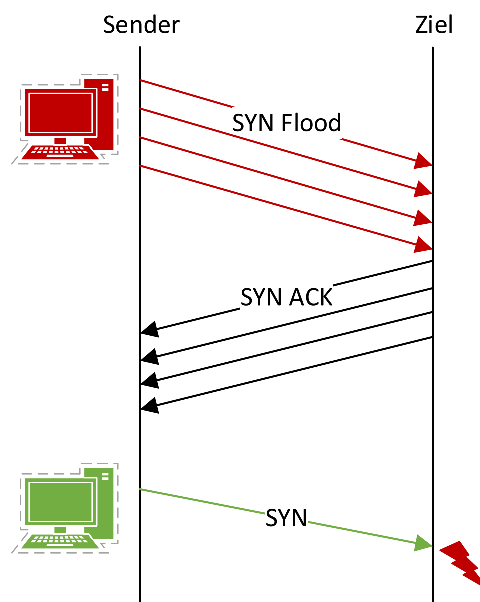


Abbildung 2.4: DoS SYN Flood Angriff.

sodass entweder das maximale Limit dieser erreicht ist und keine neuen Verbindungen mehr aufgebaut werden können oder alle Speicherressourcen aufgebraucht sind und das Ziel nicht mehr reagiert. Dadurch werden legitime Anfragen nicht mehr bearbeitet und das Ziel ist unerreichbar [SKK+97]. Durch solch einen *Flood*-Angriff auf die *PLCs* im Wasserverteilungssystem, wird somit beispielsweise die Verfügbarkeit dieser verletzt. Dadurch kommen legitime Nachrichten, um die Pumpen ein- oder auszuschalten, nicht mehr an. Ein *DoS*-Angriff auf *PLC6* hat zur Folge, dass die Nachricht von der *MTU*, um das Zuflussventil *V1* zu regulieren, nicht das Ziel erreicht. Dies führt zur Beeinflussung des Wasservolumen in den beiden Tanks *T2* und *T3*, sodass beide einen kritischen Wert erreichen. [ATKF13]

2.2 Virtualisierung

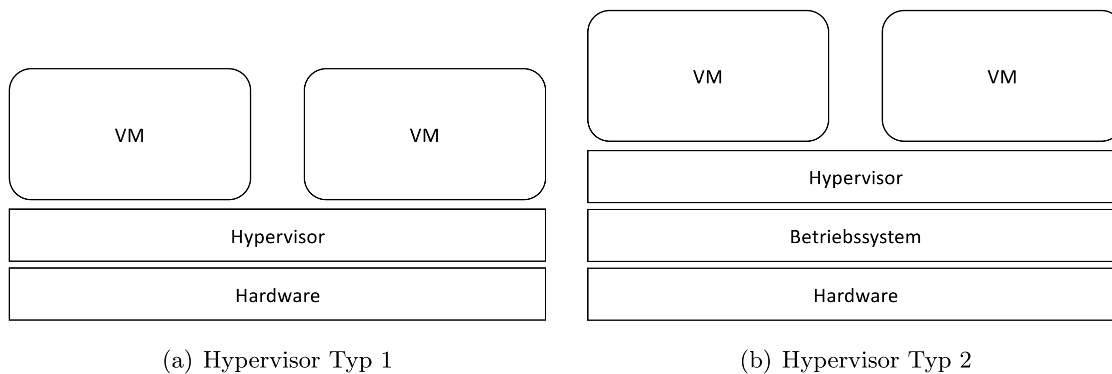
Virtualisierung ermöglicht virtuelle Ressourcen auf Basis physischer Ressourcen zu erzeugen. Dabei wird die virtuelle Ressource von der darunter liegenden physischen Hardware isoliert. Der entscheidende Vorteil bei dieser Technologie ist die Kostensenkung für die Hardwareanschaffung, da durch die Virtualisierung auf einer physischen Hardware mehrere virtuelle Maschinen (VMs) parallel betrieben werden können. Des Weiteren wird durch die Isolation zwischen der virtuellen und physischen Ressource eine gegenseitige Beeinflussung der virtuellen Systeme vermieden. Parallel laufende virtuelle Maschinen können unabhängig voneinander die geteilten physischen Ressourcen nutzen. Die Hardware-Ressourcen einzelner VMs, wie CPU, RAM, Speicher, etc. können jederzeit beliebig angepasst werden, was die Systeme sehr flexibel macht.

2.2.1 Virtualisierungsarchitekturen

Eine virtuelle Maschine wird erzeugt, indem eine abstrahierende Softwareschicht (*Hypervisor* oder *Virtual Machine Monitor (VMM)* genannt) auf einer realen physischen Maschine erstellt wird. Dadurch kann die VM physische Maschinenkompatibilität oder Ressourcenbeschränkungen umgehen [SN05]. Der Hypervisor ist für die Isolation und Verwaltung der virtuellen Ressourcen zuständig. Es werden zwischen zwei Arten von Hypervisor-Typen, Hypervisor Typ 1 und Hypervisor Typ 2, unterschieden [Gol73].

Hypervisor vom Typ 1 (Abb. 2.5a) laufen unmittelbar auf der Hardware und haben dadurch direkten Zugriff auf die physischen Ressourcen. Da auf der Hardware kein Betriebssystem läuft, braucht der Hypervisor entsprechende Treiber, um die Hardware zu unterstützen. Bekannte Typ 1 Hypervisor sind beispielsweise *Xen*, *VMware ESX* oder *Microsoft Hyper-V*.

Hypervisor vom Typ 2 (Abb. 2.5b) laufen als Anwendung auf einem herkömmlichen Betriebssystem. Dadurch werden keine speziellen Treiber seitens Hypervisor benötigt, sondern es werden die Gerätetreiber des Betriebssystems genutzt, um auf die physische Hardware zuzugreifen. Bekannte Vertreter dieses Typs sind *VMware Workstation* oder *VirtualBox*.



(a) Hypervisor Typ 1 (b) Hypervisor Typ 2

Abbildung 2.5: Hypervisor Typ 1 und Typ 2.

2.2.2 Xen Project Hypervisor

In dieser Arbeit wird der Open-Source Hypervisor Xen Project, ein Hypervisor Typ 1, verwendet. Zur Bootzeit wird eine spezielle VM, die *Domain 0* genannt wird, erstellt. Dies ist die einzige VM, die privilegierte Prozessoranweisungen ausführen darf und somit Zugriff auf die Hardware besitzt. Alle anderen *Domains* sind nicht privilegiert und werden als *DomU* bezeichnet. *Domain 0* enthält sämtliche Treiber für die Hardwareunterstützung. Darüber hinaus wird die Verwaltung von VMs, wie das Erstellen, Konfigurieren oder Löschen, mithilfe eines *Toolstack* aus *Domain 0* heraus durchgeführt. Abbildung 2.6 stellt eine solche Xen Architektur dar. Xen unterstützt die hardwareunterstützte Virtualisierung

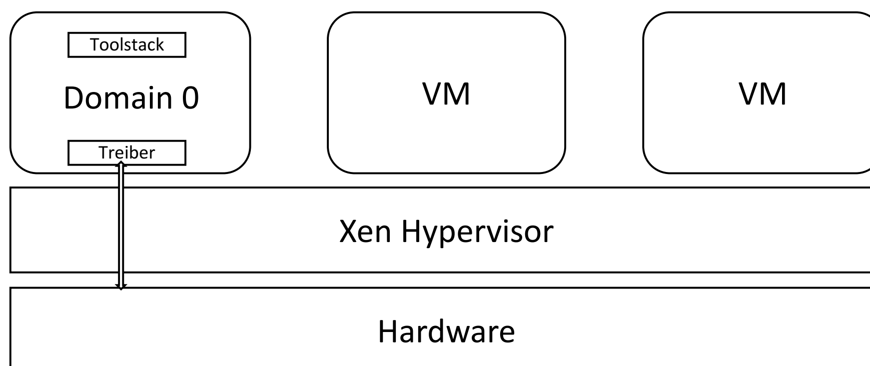


Abbildung 2.6: Xen Architektur.

(*HVM*), Paravirtualisierung (*PV*) und eine Kombination dieser Virtualisierungstechniken (*PVHVM* und *PVH*). Im Folgenden werden diese Techniken kurz erläutert. [xen17c]

HVM Bei der hardwareunterstützten Virtualisierung wird die virtuelle Maschine komplett von der Hardware isoliert ausgeführt. Die Ausführung der VM benötigt hardwareseitige Unterstützung von *Intel Virtualisierungstechnologie (VT-x) oder AMD Virtualisierung (AMD-V)*. Dadurch können Betriebssysteme unmodifiziert virtualisiert werden. *Quick*

Emulator (Qemu) [qem17] wird für die Emulation der Hardware, wie Netzwerkkarte, Festplatte, etc. genutzt, um der VM Eingabe/Ausgabe-Virtualisierung zur Verfügung zu stellen. Die Hardware Erweiterungen beschleunigen zwar diese Emulation, aber dennoch sind vollständig virtualisierte Betriebssysteme langsamer in der Ausführung als paravirtualisierte VMs. Daher ist es möglich spezielle PV-Treiber für Netzwerk- und Festplatten-E/A zu verwenden, um die Leistung der HVMs zu erhöhen. Dies wird in Xen *HVM mit PV Treiber* oder *PV-on-HVM* genannt.

PV Paravirtualisierung benötigt keine hardwareseitige Unterstützung. Stattdessen werden paravirtualisierte Treiber für alle Hardwarekomponenten benötigt, wodurch das Betriebssystem der virtuellen Maschine modifiziert werden muss. Dadurch weiß die VM, dass sie im Gegensatz zu einer HVM auf virtualisierter Hardware läuft. Allerdings wird keine Emulation der Hardware mehr benötigt, was eine Verbesserung der Leistung zur Folge hat. Die Kommunikation zwischen der VM und dem Hypervisor findet dabei durch sogenannte *Hypercalls* statt.

PVHVM Die Kombination von HVM und PV wird bei Xen als PVHVM bezeichnet. Hierbei nutzen HVMs speziell optimierte PV Treiber, um die Hardware Emulation zu umgehen und somit eine bessere Leistung zu erreichen. Der Unterschied zwischen dem anfangs erwähnten *PV-on-HVM* ist hierbei, dass neben Netzwerk und Festplatte auch Interrupts und Timer paravirtualisiert werden. Das Mainboard und PCI werden noch emuliert. Privilegierte Instruktionen werden weiterhin durch die Hardwareunterstützung virtualisiert.

PVH Bei PVH wird auf Qemu gänzlich verzichtet, indem das Mainboard und PCI Geräte ebenfalls paravirtualisiert werden. Lediglich die privilegierten Instruktionen werden weiterhin durch Hardwareerweiterungen virtualisiert. Diese Kombination der Virtualisierungstechniken soll die Vorteile aus hardwareruntersstützter Virtualisierung (HVM) und Paravirtualisierung (PV) vereinen.

Die Unterschiede der einzelnen Virtualisierungstechniken in Xen sind in Tabelle 2.1 nochmals zusammenfassend aufgelistet. Die Tabelle zeigt die einzelnen Komponenten, die entweder vollständig virtualisiert oder paravirtualisiert (*PV*) werden können. Bei der vollständigen Virtualisierung wird dabei zwischen Software (*QEMU*) und Hardwareerweiterungen wie VT-x, AMD-V (*HV*) unterschieden.

Verwaltung von Domains mithilfe des xl Toolstack

Seit Xen Version 4.2 gehört *xl* zum Standard Toolstack von Xen und dient zur Verwaltung von virtuellen Maschinen. Damit werden die VMs erstellt und gestartet oder heruntergefahren. Darüber hinaus sind viele weitere verschiedene Funktionen, wie laufende VMs

	HDD, Netzwerk	Interrupts, Timer	Mainboard, PCI	Speicherzugriff
HVM	QEMU	QEMU	QEMU	HV
PV-on-HVM	PV	QEMU	QEMU	HV
PVHVM	PV	PV	QEMU	HV
PVH	PV	PV	PV	HV
PV	PV	PV	PV	PV

Tabelle 2.1: Übersicht der Virtualisierungstechniken.

auflisten, PCI Geräte durchleiten, XSM Policy laden, Konsole einer VM öffnen etc. verfügbar [xl17].

2.3 Virtual Machine Introspection

Das Analysieren von virtuellen Maschinen von außerhalb wird als *Virtual Machine Introspection (VMI)* bezeichnet. Durch VMI können Informationen auf Hardwareebene ausgelesen werden, indem auf physischen Speicher, vCPU Register und Hardware Events zugegriffen werden. Dies geschieht mithilfe des Hypervisor, der folgende Eigenschaften besitzen muss: Isolation, Inspektion und Interposition. [GR+03]

Isolation Jede VM ist sowohl vom Hypervisor als auch von den anderen VMs vollständig isoliert. Hierdurch wird sichergestellt, dass bei einem unerlaubten Zugriff auf eine VM, weder der Hypervisor noch eine andere VM gefährdet werden kann.

Inspektion Der Hypervisor hat Zugriff auf alle Hardwarekomponenten, wie CPU Register, Speicher und E/A-Operationen.

Interposition Speicherzugriffe einer VM werden durch den Hypervisor abgefangen. Dadurch kann die VMI Anwendung entsprechend reagieren, falls eine Anwendung in einer VM unerlaubt auf ein bestimmtes Speicherregister zugreift.

Durch die Möglichkeit, mithilfe von VMI, die niedrigste Hardwareebene einer VM von außerhalb zu untersuchen, können beispielsweise alle laufenden Prozesse ausgelesen werden. Somit kann VMI schadhaften Code erkennen, welcher für Sicherheitsanwendungen innerhalb der VM unerkannt bleiben kann, da diese auf Funktionen, die vom Betriebssystem bereitgestellt werden, zurückgreifen. Ein *Rootkit* beispielsweise kann diese Funktionen manipulieren, sodass die zurückgegebenen Informationen nicht mehr vertrauenswürdig sind. Da VMI allerdings nur auf Hardwareebene Zugriff hat, besteht die Herausforderung darin, aus diesen Informationen Rückschlüsse auf die höhere Ebene, wie laufende Prozesse, zu ziehen. Diese Diskrepanz zwischen den Abstraktionsebenen wird als *semantic gap* bezeichnet [FL12].

2.3.1 Semantic Gap

Jedes Betriebssystem einer virtuellen Maschine erstellt verschiedene Softwareabstraktionen für Prozesse, Speicher, E/A, etc. Diese interagieren mit dem Hypervisor, wenn die entsprechende Abstraktion eine privilegierte Instruktion (ein sogenannter *Hypercall* in Xen) ausführt, welche vom Hypervisor abgefangen werden [XLXJ12]. Daher ist der Hypervisor in der Lage alle auftretenden *Hardware-Events* der virtuellen Maschine zu erkennen. Allerdings ist es aber schwierig diese *Events* der niedrigsten Abstraktionsstufe den Softwareabstraktionen der höheren Abstraktionsebene zuzuordnen. Der Hypervisor Xen enthält dafür *Xen Control Library (libxc)*, welche eine Kommunikationsschnittstelle für Hypercalls bietet [XLXJ12]. Damit VMI diesen *semantic gap* überwinden kann, ist ein umfangreiches Wissen der internen Kernel-Datenstrukturen des Betriebssystems notwendig. Beispielsweise erhält man innerhalb eines Linux Systems durch den Aufruf der Funktion `getpid()`, die ID des aktuell laufenden Prozesses [DGLZ+11]. Das gleiche Ziel von außerhalb des Systems zu erreichen, ist aufwendiger: Der Linux Kernel speichert eine Liste aller Prozesse in einer doppelt verketteten Liste (s. Abb. 2.7) und jedes Element dieser Liste, welches alle Details zu einem Prozess enthält, ist vom Typ `struct task_struct` [Lov10]. Der Ort dieses `struct` im Speicher und das genaue Layout dieser Datenstruktur muss also bekannt sein, damit die entsprechende Information über *introspection* erhalten werden kann. Bei einem Linux Kernel befindet sich im `/boot/` Verzeichnis eine sogenannte `System.map` Symboltabelle. Diese enthält alle globalen Variablen des Kernels und die dazugehörigen virtuellen Adressen im Speicher. Dadurch kann die Adresse des `task_struct` im Speicher ausfindig gemacht werden. Ein Durchlauf dieser Datenstruktur gibt somit nun die laufenden Prozesse und auch deren IDs aus.

Für eine *Virtual Machine Introspection* und der Bewältigung des *symantic gap*, ist es also generell notwendig, die entsprechende Symboltabelle eines Kernels zu ermitteln und mithilfe dieser interne Kernel-Datenstrukturen zu durchlaufen, um an die entsprechenden semantischen Daten aus der höheren Abstraktionsebene zu gelangen.

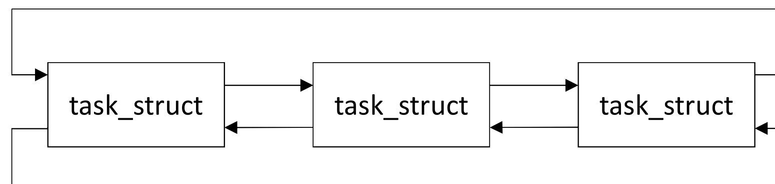


Abbildung 2.7: Linux Taskliste.

2.3.2 LibVMI

LibVMI [lib17a] ist eine C-Bibliothek, die eine Programmierschnittstelle (*Application Programming Interface (API)*) für das Schreiben von VMI-Anwendungen anbietet. Das Ziel dieser Bibliothek ist es, einerseits das *semantic gap* Problem zu beseitigen und andererseits einfache APIs bereitzustellen, die von jeder Anwendung genutzt werden können, um eine *virtual machine introspection* effizient durchzuführen. LibVMI macht dafür unter Xen von der *Xen Control Library (libxc)* Gebrauch. [XLXJ12]

Anforderungen Bei dem Entwurf und der Entwicklung von LibVMI sind verschiedene Anforderungen für die *virtual machine introspection* berücksichtigt worden, die im Folgenden kurz aufgezählt werden [PML07].

- Keine unnötigen Modifikationen am Hypervisor, damit der Code weiterhin übersichtlich und kompakt bleibt.
- Keine Änderungen auf der zu untersuchenden virtuellen Maschine notwendig, damit ein potentiell gefährdetes Betriebssystem den *introspection* Code nicht manipulieren kann.
- Keine bemerkbaren Performanceeinbußen durch die VMI.
- Bereitstellung von einfachen APIs, um schnell und einfach neue VMI Anwendungen für unterschiedliche Angriffsszenarien entwickeln zu können.
- Fähigkeit den *semantic gap* zu überbrücken und sämtliche Daten der virtuellen Maschine einsehen zu können.
- Alle VMI Anwendungen müssen von der zu untersuchenden VM vollständig isoliert sein, damit schädlicher Code diese nicht beeinflussen kann.

LibVMI API Im Folgenden werden einige der wesentlichen LibVMI APIs aufgelistet, die von Anwendungen genutzt werden, um eine VMI durchzuführen.

- **vmi_init**: Diese Funktion muss jedes Mal aufgerufen werden, um Zugriff auf eine VM zu initialisieren.
- **vmi_pause_vm**: Pausiert eine VM, falls ein konsistenter Zugriff auf den Speicher notwendig ist.
- **vmi_resume_vm**: Falls eine VM pausiert wurde, muss diese mit dieser Funktion wieder fortgesetzt werden.
- **vmi_translate_ksym2v**: Übersetzt ein Kernel Symbol in die entsprechende virtuelle Adresse.

- **vmi_read_addr_va**: Liest 8 (64-bit) bzw. 4 (32-bit) bytes aus dem Speicher der VM an der angegebenen virtuellen Adresse.
- **vmi_read_str_va**: Liest einen String aus dem Speicher der VM an der angegebenen virtuellen Adresse.
- **vmi_read_pa**: Liest eine bestimmte Anzahl an Bytes aus dem Speicher der VM an der angegebenen physischen Adresse und speichert den Inhalt in einen Buffer.
- **vmi_destroy**: Am Ende jeder *introspection* muss diese Funktion aufgerufen werden, um die anfangs erzeugte Instanz zu zerstören und Speicher wieder freizugeben.

Eine vollständige Übersicht aller APIs ist auf der LibVMI Webseite [lib17b] zu finden.

Bei einer *virtual machine introspection* mithilfe der LibVMI Bibliothek wird zunächst eine VMI Instanz initialisiert, die jegliche Information der virtuellen Maschine enthält auf die mithilfe der APIs im Laufe der *introspection* zugegriffen wird. Zum Schluss wird die Instanz dann wieder freigegeben. Im Folgenden ist ein typischer Ablauf einer introspection mithilfe von LibVMI skizziert.

```
1  #include <libvmi/libvmi.h>
2  int main (int argc, char **argv) {
3  // initialisieren der LibVMI Bibliothek
4  vmi_instance_t vmi;
5  vmi_init(vmi);
6  // pausieren der VM fuer konsistenten Speicherzugriff
7  vmi_pause(vmi);
8  // auslesen von gewuenschten Speicherstrukturen
9  [...]
10 // fortsetzen der VM
11 vmi_resume(vmi);
12 // freigeben von Speicherressourcen
13 vmi_destroy(vmi);
14 return 0;
15 }
16
```

Quellcode 2.1: Typischer Ablauf einer *introspection* mithilfe von LibVMI.

2.4 Software Defined Networking

Software Defined Networking (SDN) entkoppelt Netzwerkkontrolle und Datenweiterleitung voneinander. Dabei entstehen drei Ebenen, die als *Management Plane*, *Control Plane* und *Data Plane* bezeichnet werden [KRV+15] (s. Abbildung 2.8). Die *Data Plane* ist auf einfache Hardware reduziert, die nur der Weiterleitung der Pakete dient. Die Netzwerkpakete, die auf den Switches eingehen, werden anhand *Flow* Regeln zu einem bestimmten Port weitergeleitet oder verworfen. Die Netzwerklogik liegt zentral beim Controller (s. Abschnitt 2.4.1) in der *Control Plane*. Diese steuern die Switches in der *Data Plane* mithilfe von *Flows*. Die Kommunikation erfolgt dabei mithilfe des *OpenFlow* Protokolls (s. Abschnitt 2.4.2) über die *Southbound API*. In der obersten *Management Plane* befinden sich Anwendungen, die über die *Northbound API* den Controller ansteuern. Dadurch kann diverse Software für die Steuerung des Netzwerkes geschrieben werden. Durch diese Trennung der Schichten kann eine hohe Flexibilität im Netz erreicht werden.

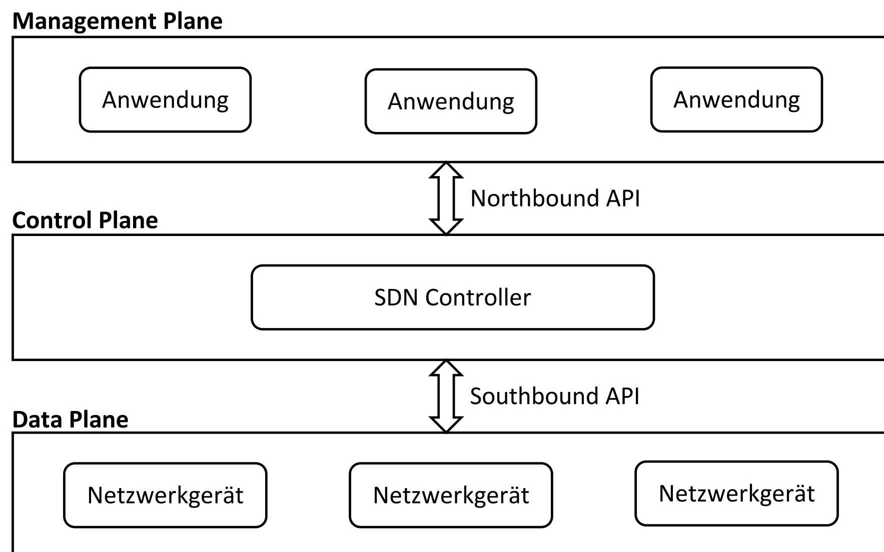


Abbildung 2.8: SDN Architektur.

2.4.1 SDN Controller

Der SDN Controller ist, wie bereits erwähnt, für das Verwalten des Netzwerkes verantwortlich. Dies ist durch die zentrale Lage und Sicht auf das komplette Netzwerk möglich. Dafür werden APIs für die Kommunikation mit der oberen und unteren Ebene angeboten. Während über die *Southbound API* mithilfe des *OpenFlow* Protokolls kommuniziert wird, wird für die Kommunikation mit Anwendungen über die *Northbound API* oft *Representational State Transfer (REST)* verwendet [RR08]. Die Programmierschnittstelle verwendet *HTTP*, um Daten vom Controller anzufragen oder zu senden. Dadurch können diverse

einfache Anwendungen und Weboberflächen geschrieben werden, die Netzwerkstatistiken vom Controller auswerten oder Befehle an die Controller versenden, um das Netzwerk zu kontrollieren. Bekannte SDN Controller sind beispielsweise *Floodlight*, *OpenDaylight* oder *NOX*.

2.4.2 OpenFlow

Der SDN Controller kommuniziert über das *OpenFlow* Protokoll [MAB+08] mit den *OpenFlow*-Switches in der *Data Plane*, um beispielsweise die *Flow*-Tabellen zu aktualisieren. Eine *Flow*-Tabelle eines Switches besteht aus mehreren Einträgen. Solch ein Eintrag ist in 2.9 abgebildet. Dieser besteht aus einer *Regel*, *Aktion* und *Counter*. Eine Regel hat folgende

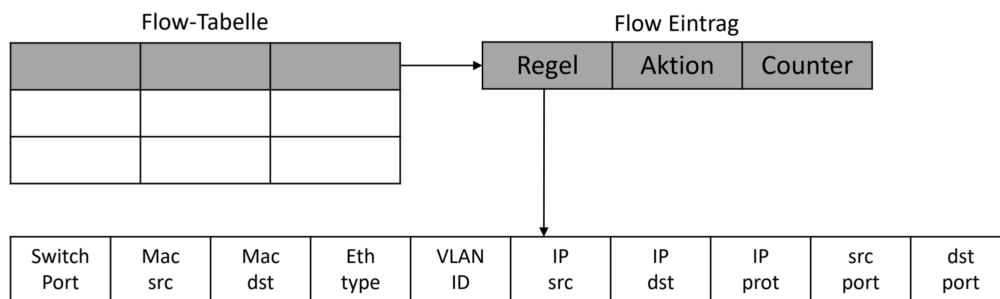


Abbildung 2.9: Eintrag einer Flowtabelle eines *OpenFlow* Switches.

Felder:

- **Switch port**: Ankommender Switch Port.
- **Mac src**: MAC Adresse der Quelle.
- **Mac dst**: MAC Adresse des Ziels.
- **Eth type**: Ethernet Typ, für die Bestimmung des Paketprotokolls. Beispielsweise *0x0800* für das Internet Protokoll version 4 (IPv4).
- **VLAN ID**: VLAN ID.
- **IP src**: IP Adresse der Quelle.
- **IP dst**: IP Adresse des Ziels.
- **IP prot**: IP Protokoll des Pakets.
- **srcport**: Quell-Port des Senders.
- **dstport**: Ziel-Port des Empfängers.

Wenn nun ein ankommendes Paket mit den Feldern der Regel übereinstimmt, wird die entsprechende Aktion ausgeführt. Die Aktion ist entweder eine Weiterleitung zu einem bestimmten Port oder das Verwerfen des Pakets. Alle nicht übereinstimmten Pakete werden zum Controller gesendet (*Packet-In*), damit eine Entscheidung getroffen werden kann, wie mit diesen zu verfahren ist. Anschließend sendet der Controller ein *Packet-Out* mit der Aktion, die ausgeführt werden soll und einen neuen *Flow* für zukünftige Pakete dieser Art. Der Counter dient zur Statistik von den übereinstimmten Paketen.

3 Stand der Forschung

Dieses Kapitel listet die für diese Arbeit relevanten wissenschaftlichen Studien auf.

In den beiden Veröffentlichungen [CAS08] und [CAS+09] wird die Sicherheit in cyber-physischen Systemen analysiert. Es wird festgestellt, dass die Sicherheitsrisiken in CPS steigen und der Fokus bislang hauptsächlich auf die Zuverlässigkeit des Systems (*Safety*), anstatt auf den Schutz vor bösartigen Angriffen (*Security*), gelegen hat. Es werden drei Herausforderungen, um CPS zu schützen, diskutiert: Analyse von möglichen Angriffs-Szenarien und deren Konsequenzen auf CPS, Unterschied zwischen traditioneller IT-Sicherheit und mögliche Sicherheitsmaßnahmen. Dabei werden Entwurfsprinzipien wie das Trennen von höheren Privilegien im System, Redundanz und Fehlertoleranz empfohlen, um Angriffe auf CPS zu verhindern. Diese Eigenschaften werden unter anderem auch durch die Nutzung der Virtualisierungstechnik erfüllt, was in den Studien jedoch nicht erwähnt wird. Zusammenfassend wird gesagt, dass viele der Herausforderungen noch ein offenes Thema der Forschung sind.

In [ATKF13] wird eine virtualisierte Testumgebung vorgestellt, um die Sicherheit von CPS einfach und flexibel untersuchen zu können, ohne die Sicherheit eines realen CPS zu kompromittieren. Damit kann ein CPS emuliert werden, um die möglichen Konsequenzen eines Angriffs zu untersuchen. Die Autoren zeigen, wie die virtualisierte Ausführungsplattform physische Prozesse überwachen und kontrollieren kann. Es werden die Effekte eines Angriffs auf diese Infrastruktur dargestellt. Wie diese Art von Angriffen verhindert werden können, war nicht Thema der Veröffentlichung.

Der, in dieser Arbeit verwendete, Xen Hypervisor als virtualisierte Ausführungsplattform, wird in [BDF+03] vorgestellt. Das Konzept der *Paravirtualisierung* wird erläutert und es wird erwähnt, dass dies notwendig ist, um die *Performance* des Systems deutlich zu verbessern. Dass die Sicherheit von Xen durch eine Disaggregation der *Domain 0* erhöht werden kann, wird in [MMH08] und [Dun13] gezeigt. Dabei werden verschiedene Komponenten aus der *Domain 0* ausgelagert, um die *Trusted Computing Base (TCB)* zu verringern. Diese Trennung von Privilegien, was auch in [CAS+09] als ein empfohlenes Entwurfsprinzip erwähnt wurde, führt dazu, dass die Rechte eines kompromittierten Systems somit eingeschränkt werden. Dieser Ansatz wird in dieser Arbeit verfolgt.

[JBZ+15] und [BJW+10] behandeln *Virtual Machine Introspection* und den Begriff des *Semantic Gap*, der überwunden werden muss, um mithilfe von VMI nützliche Daten von einer VM auslesen zu können. Die beiden Studien gehen dabei auch auf die verschiedenen

Sicherheitsaspekte der Technik ein. [XLXJ12] und [Pay12] stellen die LibVMI-Bibliothek vor, welche das Problem des *Semantic Gap* löst und verschiedene Möglichkeiten bietet, eine VMI auszuführen. Die Studie [DGPL11] stellt einen Zusammenhang zwischen der forensischen Speicheranalyse (*Forensic Memory Analysis (FMA)*) und der VMI her. Beide Bereiche müssen das Problem des *Semantic Gap* lösen, daher wird geschlussfolgert, dass bereits existierende *FMA-Tools* in Kombination mit LibVMI verwendet werden sollten, um Redundanz zu vermeiden. Durch solch eine Kombination der verschiedenen Techniken wird in dieser Arbeit versucht, typische Angriffe auf die virtualisierte Ausführungsplattform abzuwehren.

Die Studien [KRV+15] und [ANYG15] beschäftigen sich eingehend mit *Software Defined Networking*. Die Vorteile gegenüber des traditionellen Netzwerks sowie potentielle neue Sicherheitsrisiken durch SDN werden erörtert. [PPA+09] stellt mit *Open vSwitch* ein Netzwerkswitch vor, der für virtualisierte Umgebungen entwickelt wurde und SDN unterstützt. [GHB+15] zeigt wie Angriffe auf das Kommunikationsnetz von cyber-physischen Systemen effektiv abgewehrt werden können. Dazu wird das Konzept des *Software Defined Networking* vorgestellt. Anschließend werden Möglichkeiten aufgezählt, um mithilfe von SDN DoS-Angriffe abwehren zu können und diese in zwei Experimente umgesetzt. Das Erkennen und Abwehren von DoS-Angriffen, wird dabei durch den SDN Controller erledigt. Dies kann allerdings effektiver umgesetzt werden, wenn die Aufgaben in zwei Komponenten getrennt werden, um den Controller zu entlasten und somit die Latenz im Netzwerk zu verringern. Dieser Ansatz wird in dieser Arbeit berücksichtigt.

Keine der vorgestellten Arbeiten berücksichtigen eine vollständig virtualisierte Infrastruktur für sowohl die Ausführungsplattform als auch das Kommunikationsnetz im Kontext von cyber-physischen Systemen. Die vorliegende Arbeit untersucht die Sicherheitsaspekte solch einer Infrastruktur unter Nutzung von *Virtual Machine Introspection* und *Software Defined Networking*. Die meisten Studien verwenden für die Evaluation von SDN, den Netzwerkemulator *Mininet* [min17] [LHM10]. *Mininet* emuliert eine Netzwerkumgebung mit *Hosts* und *Switches*, die als Linux Prozesse auf einer Maschine laufen. Diese Arbeit evaluiert die verwendeten Sicherheitstechniken auf einer realen virtualisierten Infrastruktur, was realistischere Ergebnisse, im Vergleich zu einer Emulation, liefert.

4 Entwurf und Implementierung einer sicheren virtualisierten Infrastruktur

In diesem Kapitel wird eine durchgängig virtualisierte Infrastruktur für die Ausführungsplattform und das Kommunikationsnetz entworfen und anschließend implementiert. Dabei werden verschiedene bereits verfügbare Sicherheitsmaßnahmen miteinander kombiniert und in die Infrastruktur eingebettet, um damit den anfangs geschilderten Sicherheitsrisiken in cyber-physischen Systemen entgegenwirken zu können. Danach wird auf die Sicherheitsaspekte der Virtualisierung selber und die konzipierten Sicherheitsmaßnahmen eingegangen.

Systemanforderungen Zunächst werden die notwendigen Anforderungen an die Infrastruktur definiert.

1. Die Infrastruktur sollte fehlertolerant sein, um die Zuverlässigkeit und Verfügbarkeit des Systems zu gewährleisten.
2. Die Sicherheit der Infrastruktur sollte gewährleistet werden.
   - Im Falle eines Ausbruchs aus einer VM, sollte nicht das ganze System gefährdet werden
3. Angriffe auf die Ausführungsplattform und das Kommunikationsnetz sollten erkannt werden.
   - Initialisieren von bösartigen Rootkits auf VMs sollten erkannt werden.
   - Klassische DoS Angriffe auf das Kommunikationsnetz sollten erkannt werden.
   - Das System sollte diese Angriffe durch Gegenmaßnahmen abwehren.
4. Die konzipierten Sicherheitsmechanismen sollten effizient sein und die Performance des Systems nicht stark beeinflussen.
   - Die Ressourcenauslastung der VMs sollten vertretbar sein.
   - Die Latenz im Netzwerk sollte akzeptabel sein.

Mit Berücksichtigung dieser Anforderungen wird in den folgenden Abschnitten die Infrastruktur entworfen und implementiert.

Die Virtualisierung bietet ein sehr hohes Maß an Fehlertoleranz, durch die einfache Möglichkeit der Replikation von virtuellen Maschinen, um eine Hochverfügbarkeit von Systemen zu gewährleisten. Durch die Isolation der einzelnen Systemkomponenten aufgrund der Virtualisierung, wird außerdem eine erhöhte Sicherheit erreicht. Bei einem Fehlerfall oder Kompromittierung einer Systemkomponente, wird eine Propagation von Fehlern auf andere Komponenten durch die Isolation verhindert. Darüber hinaus wird durch eine Ausführung mehrerer virtueller Systeme auf einer physischen Maschine, Kostenersparnisse und eine hohe Flexibilität erreicht. Daher wird die Virtualisierung für die Überwachungs- und Verwaltungsinfrastruktur von cyber-physischen Systemen in Betracht gezogen.

4.1 Virtualisierung der Ausführungsplattform

Der Hypervisor ist die wichtigste Komponente bei der Plattformvirtualisierung und gehört zur sogenannten *Trusted Computing Base (TCB).* Die *TCB* enthält alle sicherheitskritischen Komponenten eines Systems. Je kleiner diese *TCB* ist, desto vertrauenswürdiger und sicherer ist auch der Code [MMH08]. Der Xen Project Hypervisor ist ein populärer *Open-Source* Hypervisor vom Typ 1, welcher direkt auf Hardware läuft (vgl. 2.2.2). Mit weniger als 150.000 Zeilen Code [xen17c] hält der Xen Hypervisor die *TCB* klein. Durch die privilegierte VM *Domain 0*, welches ein vollwertiges Unixsystem ist, das auch zur *TCB* gehört, und der Möglichkeit jede beliebige Software in dieser Domain zu installieren, kann die *TCB* allerdings viel größer werden. Xen bietet verschiedene Sicherheitserweiterungen an, die aber nicht standardmäßig aktiviert sind, sondern beim Entwurf eines Systems berücksichtigt werden müssen. Einer dieser Aspekte ist die sogenannte *Domain 0 Disaggregation* [MMH08], welche das Ziel hat bestimmte Dienste aus der privilegierten *Domain 0* in eine unprivilegierte *Domain U* auszulagern.

4.1.1 Domain 0 Disaggregation

Als *Domain 0 Disaggregation* wird eine Aufteilung der *Domain 0* in mehrere Domains bezeichnet. Abbildung 4.1 zeigt eine traditionelle Architektur des *Xen Hypervisor*. Das *Qemu Device Model* und die Netzwerktreiber laufen standardmäßig in *Domain 0*. Durch Auslagerung dieser Komponenten können mehrere Vorteile erzielt werden. Einerseits wird die Performance der *Domain 0* verbessert, da nun nicht mehr sämtliche Netzwerk-*Backends* oder Qemu Prozesse innerhalb *Domain 0* laufen müssen. Andererseits wird die Sicherheit durch eine zusätzliche Schicht erhöht. Denn beispielsweise bei einem Ausbruch durch etwaige Bugs über die Netzwerkebene, wird nun nur Zugriff auf eine unprivilegierte VM erlangt anstatt auf *Domain 0* und damit der vollen Kontrolle des ganzen Systems. Die Domains in die das *Qemu Device Model* und die Netzwerktreiber ausgelagert werden können, werden in Xen als *Device Model Stub Domains* [xen17a] respektive *Network Driver Domains* [xen17b] bezeichnet. Diese werden im Folgenden betrachtet.

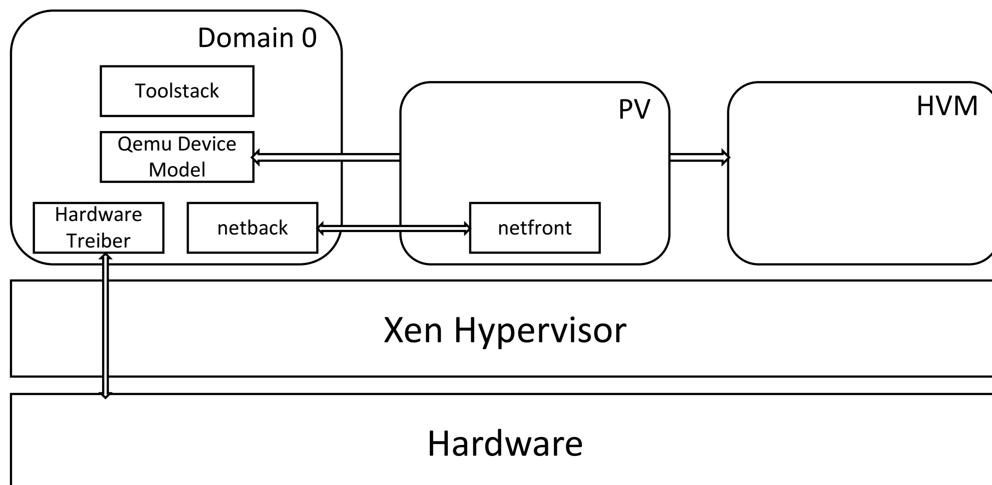


Abbildung 4.1: Traditionelle Xen Architektur.

Device Model Stub Domain

Mit jeder Instanziierung einer *HVM* (vgl. 2.2.2) läuft ein *Qemu Device Model*-Prozess in *Domain 0*. Durch die Auslagerung dieser Prozesse in sogenannte *Stub Domains*, wird die Reaktionszeit der *HVM* erhöht, da *Domain 0* nun die Qemu Prozesse nicht mehr *schedulen* muss. Falls *Domain 0* zu irgendeinem Zeitpunkt eine hohe Last haben sollte, werden die VMs dadurch nicht mehr beeinflusst. Dadurch, dass *Stub Domains* unprivilegiert sind und nur gegenüber der jeweiligen *HVM* privilegiert sind, wird zudem die Sicherheit des Systems erhöht. Als Betriebssystem für die *Stub Domains* wird ein *Mini-OS* genutzt. *Mini-OS* ist ein *Unikernel* mit nur einem Adressbereich [xen08]. In diesem Betriebssystem läuft jeweils nur ein Qemu Prozess, das mit der entsprechenden *HVM* assoziiert ist. Diese, nun von der *Domain 0* entkoppelte, Komponente ist in Abbildung 4.2 dargestellt. Ein Ausbruch aus

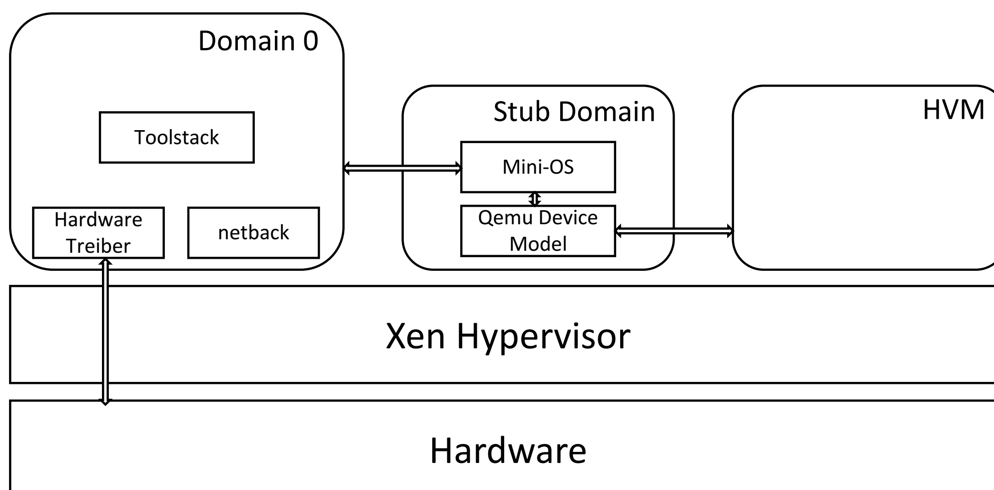


Abbildung 4.2: Xen Device Model Stub Domain.

der *HVM* über etwaige *Exploits* in den emulierten virtuellen Geräten, ermöglicht nun nur Zugriff auf die *Stub Domain* anstelle der privilegierten *Domain 0*.

Die *Stub Domain Binary* wird beim Kompilieren von Xen ebenfalls erstellt und befindet sich unter `/usr/lib/xen/boot/ioemu-stubdom.gz`. Anschließend kann in der *config*-Datei der jeweiligen VM (s. 2.2.2) mit der Zeile `device_model_stubdomain_override = 1` die Nutzung von *Stub Domains* aktiviert werden. [xen17a]

Network Driver Domain

Die Domain, in der die Netzwerk-*Backends* der VMs aus *Domain 0* ausgelagert werden, wird als *Network Driver Domain* bezeichnet. Diese Domain ist eine unprivilegierte VM, welche den anderen VMs Zugriff auf die virtuellen und physischen Netzwerkschnittstellen bietet. Ähnlich wie im vorherigen Abschnitt, wird durch eine Auslagerung neben der Sicherheit auch die Perfomance erhöht. Die von der *Domain 0* entkoppelte Netzwerkkomponente ist in 4.3 abgebildet. Im Falle eines Ausbruchs durch Sicherheitslücken, kann zwar Zugriff

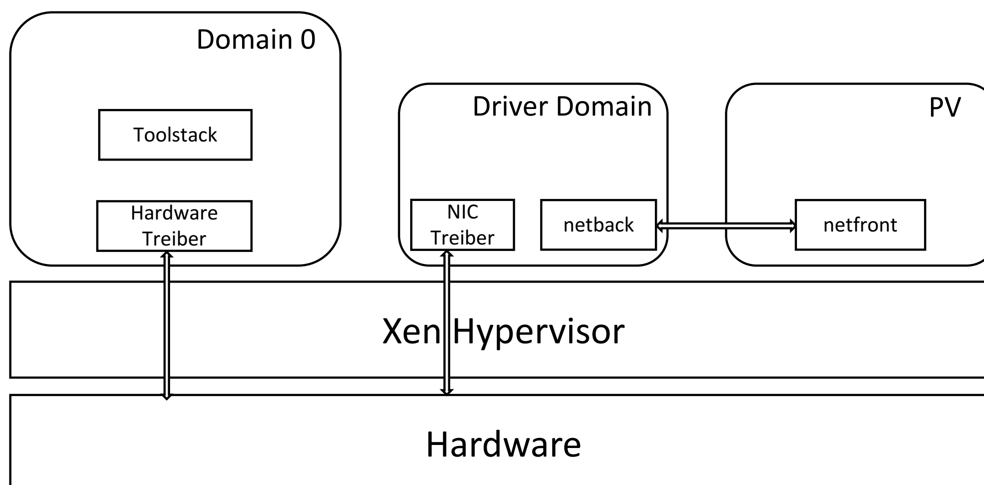


Abbildung 4.3: Xen Network Driver Domain.

auf die *Driver Domain* und somit auch auf die Netzwerkschnittstelle erlangt werden, um beispielsweise die anderen VMs anzugreifen [Dun13]. Aber es wird nun kein Zugriff mehr auf die priviligierte *Domain 0* und somit auf das Gesamtsystem erlangt.

Die Implementierung der *Driver Domain* erfolgt in mehreren Schritten [xen17b]. Zunächst muss eine VM mit den entsprechenden Treibern für die Netzwerkschnittstelle erstellt werden. Diese VM muss ebenfalls die Xen-Skripte zum Erstellen der virtuellen Netzwerkschnittstellen enthalten. Diese dienen dafür, die jeweiligen *xen-netfront Frontends* der VMs mit dem *xen-netback Backend* in der *Driver Domain* zu verbinden. Für diesen Prozess können die *Xen Tools* in der VM installiert werden, die alle notwendigen Komponenten beinhalten.

Als Nächstes muss der *Driver Domain* Zugriff auf die entsprechende physische Netzwerkkarte des *Hosts* gewährt werden. Dies geschieht durch *Xen PCI Passthrough* [pci16]. Mithilfe von *PCI Passthrough* wird die vollständige Kontrolle eines PCI-Gerätes an eine VM durchgereicht. In diesem Fall die Kontrolle einer physischen Netzwerkkarte des *Hosts* an die *Driver Domain*. Dadurch hat *Domain 0* keine Kontrolle mehr zu dem entsprechenden Gerät. Damit das *Passthrough* funktioniert, muss das Modul *xen-pciback* geladen sein. Durch Zuweisung des PCI Gerätes zu diesem Modul, wird das Gerät für ein *Passthrough* zur *Driver Domain* verfügbar gemacht. Welches Gerät durchgereicht werden soll, wird in der *config*-Datei der *Driver Domain* mit `pci = ['BDF-Notation']` spezifiziert. BDF bedeutet *Bus Device Function* und wird benutzt um ein PCI Gerät zu beschreiben. Diese setzt sich wie folgt zusammen: `PCI Bus:PCI Device.PCI Function`. Also beschreibt `pci = ['00:01.0']` beispielsweise den Bus 0, Gerät 1 und Funktion 0. [bdf14] Eine Liste aller PCI Geräte und deren BDF erhält man in *Domain 0* mit dem Befehl `lspci`.

Anschließend muss die zu nutzende Netzwerktopologie aufgebaut werden (s. dafür Abschnitt 4.2). Zum Schluss wird in der *config*-Datei der VMs bei der Netzwerkschnittstelle ein zusätzlicher Parameter angegeben: `vif = ['..., backend=driverdomain']`. Dies hat zur Folge, dass die VMs sich bei der Netzwerkeinrichtung nun nicht mehr mit dem *Backend* der *Domain 0* verbinden, sondern mit der *Driver Domain*.

Die virtualisierte Architektur mit der disaggregierten *Domain 0* sieht also nun wie in 4.4 abgebildet aus. Durch die Disaggregation wird die Sicherheit des Systems erhöht, sodass im Falle eines Ausbruchs aus einer VM nicht das ganze System gefährdet wird. Hiermit wird Punkt 2 der Systemanforderungen berücksichtigt.

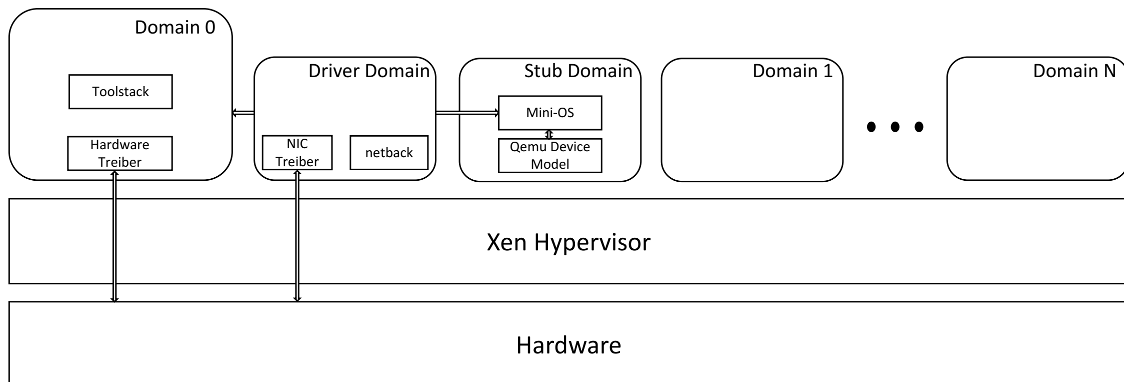


Abbildung 4.4: Xen Architektur mit der disaggregierten *Domain 0*.

4.1.2 Virtual Machine Introspection

Mithilfe von *Virtual Machine Introspection* (s. Kapitel 2.3) können potentielle Integritätsverletzungen einer virtuellen Maschine erkannt werden. Ein großer Vorteil dieser Technik liegt darin, dass die Analyse und Erkennung von bösartigen *Rootkits* außerhalb der VM erfolgt. Durch diese Isolation kann eine kompromittierte VM diesen Sicherheitsmechanismus, im Gegensatz zu einer VM-internen Lösung, nicht so einfach manipulieren. Mit der LibVMI-Bibliothek (s. Kapitel 2.3.2) kann diese Technik genutzt werden, um ein System anhand der Kernel-Strukturen im Speicher zu analysieren. Allerdings muss für jeden Anwendungszweck (Prozesse auslesen, geladene Module auslesen, die Syscall Tabelle überprüfen, etc.) eine eigene Anwendung geschrieben werden. Der Bereich der forensischen Speicheranalyse (*Forensic Memory Analysis (FMA)*) beschäftigt sich ebenfalls mit dem gleichen Problem, aus Speicherstrukturen die semantisch relevanten Daten zu rekonstruieren - also den sogenannten *Semantic Gap* (s. Kapitel 2.3.1) zu überwinden. Sowohl *VMI* als auch *FMA* nutzen also den physikalischen Speicher für ihre Analyse. Der Unterschied zwischen beiden Techniken ist, dass *VMI live* auf den Speicher zugreift, während die VM in Betrieb ist und somit den entscheidenden Vorteil hat, ein Sicherheitsrisiko zeitnah zu erkennen. *FMA* erstellt einen *Memory Dump*, also einen Snapshot, des Speichers und analysiert diesen anschließend. [DGPL11] zeigt, dass durch eine Kombination von *VMI* und *FMA Tools*, von der bereits existierenden Arbeit profitiert werden kann. Dadurch wird die Redundanz, für jeden entsprechenden Anwendungszweck eine VMI Anwendung mit der gleichen Funktionalität zu schreiben, vermieden.

Kombination von LibVMI und FMA Tools

Volatility [vol17a] und *Rekall* [rek17c] sind zwei *Open-Source Memory Forensics Frameworks* für die Analyse von *Memory Dump Files*. Die beiden FMA Tools enthalten zahlreiche Plugins, welche verwendet werden können, um beispielsweise laufende Prozesse oder geladene Kernel Module auszulesen. Diese Tools können allerdings nur auf Speicherformate zugreifen, die als Speicherabbild in einer Datei vorliegen. Damit nun der Speicher einer virtuellen Maschine mit den FMA Tools *live* analysiert werden kann, ist eine Kombination dieser Tools mit der LibVMI API notwendig. Mithilfe von VMI Filesystem (*vmifs*) [vmi17] wird dabei der Speicher der VM als eine Datei dargestellt, welche anschließend mit Volatility oder Rekall analysiert werden kann. In Abbildung 4.5 ist dieser Prozess schematisch dargestellt. VMIFS nutzt dafür das FUSE (*Filesystem in Userspace*) [fus17] Interface. Mit dieser Bibliothek können Anwendungen im *Userspace* Dateisysteme einbinden. Der Befehl `vmifs name|domid <name|domid> <path>` bindet den Speicher einer virtuellen Maschine in dem angegebenen Pfad ein. Es entsteht eine Datei, die anschließend mit Volatility oder Rekall analysiert werden kann. Dabei rufen die FUSE Operationen benutzerdefinierte Methoden auf, welche die LibVMI APIs verwenden (s. Quellcode-Ausschnitt 4.1).

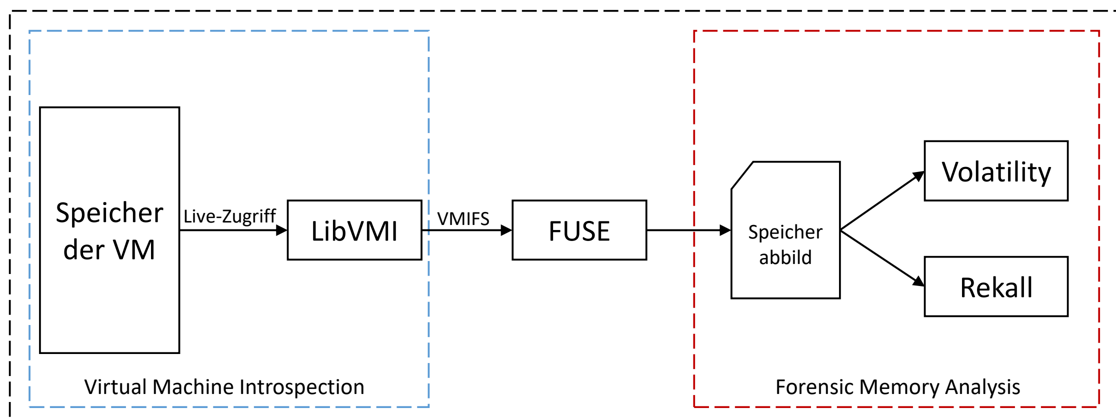


Abbildung 4.5: Kombination von LibVMI und FMA Tools.

```
1    static struct fuse_operations vmifs_oper = {
2      .getattr    = vmifs_getattr,
3      .readdir    = vmifs_readdir,
4      .open    = vmifs_open,
5      .read    = vmifs_read,
6      .destroy    = vmifs_destroy,
7    }
8
```

Quellcode 4.1: Abbildung von fuse_operations auf benutzerdefinierte Methoden. [vmi17]

Beispielsweise ruft FUSE beim Lesen des Speicherabbildes, durch ein FMA Tool, die eigene `read`-Methode auf, welche dann wiederum die `vmifs_read`-Methode aufruft. `vmifs_read` nutzt dann die `vmi_read_pa` LibVMI API (s. Kapitel 2.3.2), um den physikalischen Speicher der VM zu lesen. Durch diese Kombination von LibVMI und FMA Tools können nun sämtliche bereits vorhandene Plugins genutzt werden, um den Speicher der VM zu analysieren und potentielle Integritätsverletzungen zu entdecken. Es gibt zwei Möglichkeiten potentiell bösartige Rootkits zu erkennen: Entweder durch eine passive oder aktive *Virtual Machine Introspection*. Die Unterschiede werden im folgenden Abschnitt erläutert.

Passive und aktive VMI

Bei einer passiven VMI wird in einem bestimmten Zeitintervall der Speicher der VM auf Veränderungen überprüft. Dieser Prozess kann leicht automatisiert werden, indem alle laufenden VMs nacheinander mit den gewünschten Rekall bzw. Volatility Plugins analysiert werden. Beide benötigen Python für die Ausführung. Damit diese FMA Tools wissen wie der Speicher der virtuellen Maschine strukturiert ist, werden sogenannte *Profile* mit Debugging Symbolen benötigt, um den semantischen Kontext herzustellen. Diese Profile sind für jeden Linux Kernel unterschiedlich und enthalten unter anderem die jeweilige *System.Map*

des Kernels. Damit das Speicherabbild analysiert werden kann, muss ein passendes Profil der VM vorhanden sein. Rekall bietet dafür zwei Optionen. In einem öffentlichen Repository [rek17a] befinden sich Profile für die unterschiedlichen Systeme. Falls ein passendes Profil für den Kernel nicht vorhanden ist, wird dies innerhalb der virtuellen Maschine erstellt. Da dieses Profil bei jedem Scan gelesen wird, bietet Rekall für eine bessere Performance noch die Möglichkeit das Profil in ein JSON Format umzuwandeln. Ein Nachteil der passiven VMI besteht darin, dass eine potentielle Gefahr erst beim Scan erkannt wird. Die Erkennungsverzögerung ist also von dem Zeitintervall der Scans abhängig.

Bei einer aktiven VMI werden bestimmte Hardware Events durchgehend beobachtet. Dies ist durch die Interposition (s. Kapitel 2.3) des Hypervisor möglich. Das Beobachten von solchen Hardware Events wird ebenfalls durch LibVMI unterstützt. Es ist auch mit den vorhin erwähnten Rekall Profilen kompatibel. Daher kann dieses Profil auch bei der aktiven VMI für das Interpretieren des Speichers genutzt werden. Das Beobachten von Hardware Events in LibVMI geschieht durch das Registrieren von *Events* mit `vmi_register_event()`, um beispielsweise *Interrupt Events* beobachten zu können. Interrupts können entweder durch externe Hardware oder Software ausgelöst werden. [Int17] Diese werden durch die sogenannte *Interrupt Descriptor Table (IDT)* behandelt, welche Vektoreinträge für entsprechende *Exceptions* und *Interrupts* beinhalten. Wenn nun zum Beispiel ein bestimmter *Syscall*, wie `sys_init_module` für das Laden von Kernel Modulen, beobachtet werden soll, kann mithilfe der INT 3 Instruktion [Int17] ein Software Breakpoint an dieser Adresse im Speicher gesetzt werden. Mit `vmi_events_listen()` werden dann, in einem bestimmten Zeitintervall, die registrierten *Events* überprüft. Wenn das *Event* dann eintritt, also in diesem Fall ein Linux Kernel Modul geladen wird, wird dies der entsprechenden benutzerdefinierten Rückruffunktion übergeben. Anschließend muss das System dann wieder fortgesetzt werden. Ein Beispiel Quellcode-Ausschnitt ist in 4.2 dargestellt. Durch diesen Trap und der anschließenden Fortsetzung des Systems, gibt es einen Systemoverhead beim Erkennen eines Events. Dies stellt einen Nachteil der aktiven VMI dar.

```
1   #include <libvmi/libvmi.h>
2   #include <libvmi/events.h>
3   [...]
4   event_response_t callback_fct(vmi_instance_t vmi, vmi_event_t *event)
5   {
6     printf("Received a trap event!");
7     [...]
8   }
9   vmi_event_t trap_evt;
10  SETUP_INTERRUPT_EVENT(&trap_evt, 0, callback_fct);
11  vmi_register_event(vmi, &trap_evt);
12  while(true)
13  {
```

```
14        status = vmi_events_listen(vmi,500);
15        if (status != VMI_SUCCESS)
16        {
17          printf("Error!\n");
18          break;
19        }
20      }
21      [...]
22
```

Quellcode 4.2: Beispiel Quellcode-Ausschnitt einer aktiven *Virtual Machine Introspection*.

Je nach Anwendungsfall, kann also mit LibVMI zwischen der Nutzung von passiver oder aktiver VMI entschieden werden. Beide Methoden werden in Kapitel 5 evaluiert.

Durch *Virtual Machine Introspection* und der Nutzung von Volatility und Rekall in Kombination, sieht die virtualisierte Ausführungsplattform mit den Sicherheitsmaßnahmen nun wie in 4.6 abgebildet aus. Hiermit wird die Teilanforderung von Punkt 3 der Systemanforderungen berücksichtigt, dass es eine Möglichkeit gibt bösartige Rootkits auf virtuellen Maschinen zu erkennen.

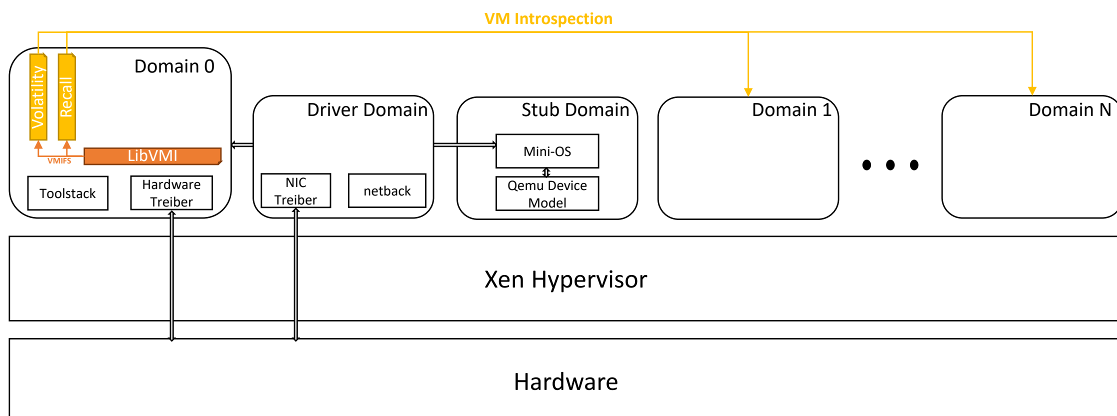


Abbildung 4.6: Virtualisierte Ausführungsplattform mit Sicherheitsmechanismen.

4.2 Virtualisierung der Kommunikation

Nach der Virtualisierung der Ausführungsplattform, wird nun das Kommunikationsnetz virtualisiert. Hier sollte die Teilanforderung von Punkt 3 berücksichtigt werden, dass klassische DoS Angriffe erkannt und abgewehrt werden. Dazu wird im folgenden Abschnitt das Konzept des *Software Defined Networking* angewandt.

4.2.1 Software Defined Networking

Im traditionellen Netzwerk sind *Control Plane* und *Data Plane* (s. Kapitel 2.4) in jedem Switch gekoppelt. Dadurch ist die Verwaltung eines traditionellen Netzwerks bei vielen Switches oder einer dynamischen Umgebung sehr komplex [KRV+15]. Eine automatische Konfigurierung der Switches gestaltet sich, aufgrund herstellerspezifischer Befehle auf niedriger Ebene, ebenfalls sehr schwierig. Durch die Entkopplung der *Control Plane* und *Data Plane* beim *Software Defined Networking*, liegt die Logik zentral beim *SDN-Controller* (s. Kapitel 2.4.1). Diese Logik der Netzwerkkontrolle ist softwarebasiert und wird durch höhere Programmiersprachen gesteuert, welche die *Data Plane* anschließend umsetzt. Durch diese *Trennung der Belange* steigt die Flexibilität und Skalierbarkeit des Netzes enorm. Denn nun kann der *Controller* bei einem Problem das gesamte Netzwerk mittels dem *Openflow* Protokoll (s. Kapitel 2.4.2) rekonfigurieren, da dieser eine komplette Sicht auf alle Switches hat. Dadurch sieht der *Controller* ebenfalls den gesamten Traffic aller Switches, was den Vorteil hat Traffic Anomalien und somit einen etwaigen DoS Angriff zu erkennen und entsprechende Gegenmaßnahmen einzuleiten.

Open vSwitch (OvS) [ovs17a] [PPA+09] ist ein virtueller *Open-Source* Switch, der *SDN* unterstützt. Ein virtueller Switch dient sowohl zur Interkommunikation von VMs als auch zur Kommunikation zwischen den virtuellen und den physikalischen Netzwerkschnittstellen. *OvS* teilt sich in zwei Komponenten auf (s. Abbildung 4.7): ein *ovs-vswitchd Daemon* zur Kontrolle des Switches, der im Userspace läuft und ein *datapath kernel module* zur Paketweiterleitung, der im Kernelspace läuft [ERWC14]. Die Komponenten werden als *fast path* und *slow path* bezeichnet [PPA+09]. Alle Pakete, die einem *Flow* in der *Flow Tabelle* entsprechen, werden direkt durch das Kernel Modul über den *fast path* verarbeitet. Die Pakete, die noch keinen Eintrag in der *Flow Tabelle* haben, gehen über den *slow path* im *Userspace* und werden durch den *ovs-vswitchd Daemon* mittels *OpenFlow* Regeln verarbeitet. [ERWC14] Diese *OpenFlow* Regeln werden durch einen *SDN*-Controller hinzugefügt. *OvS* unterstützt den Xen Hypervisor und wird standardmäßig in *Domain 0* installiert. Hier läuft der Switch, aufgrund der *Domain 0 Disaggregation* (s. Kapitel 4.1.1), aber in der *Network Driver Domain.* Nach der Installation, kann entweder die Standardkonfiguration von Xen mit `vif.default.script="vif-openvswitch"` oder die jeweilige *Config*-Datei einer VM mit `vif = ['script=vif-openvswitch, ...']` angepasst werden, damit Open vSwitch anstelle der Standard Linux Bridge verwendet wird. `vif-openvswitch` ist ein Skript,

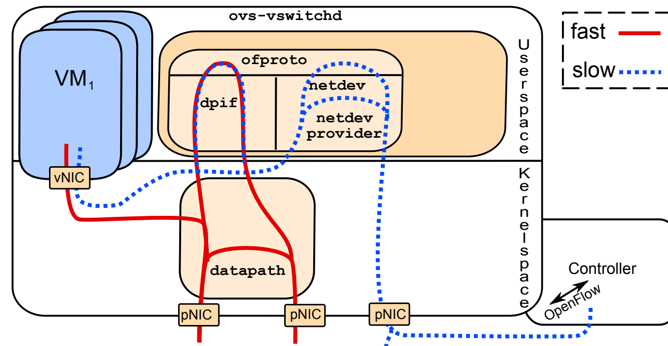


Abbildung 4.7: Open vSwitch Architektur. [ERWC14]

welches dazu dient jede virtuelle Netzwerkschnittstelle einer VM mit dem spezifizierten *OvS* zu verbinden. *OvS* bringt mehrere Kommandozeilenprogramme zur Verwaltung der Switches mit [ovs17b]. `ovs-vsctl` wird beispielsweise für das Konfigurieren des *ovs-vswitchd Daemon* verwendet und ist damit das wichtigste Tool für die Konfiguration der *OpenFlow* Switches. Mit `ovs-ofctl` können die Flows der einzelnen Switches über die Kommandozeile ausgegeben und modifiziert werden. Für eine dynamische und automatische Verwaltung des Netzes wird allerdings, wie erwähnt, ein SDN Controller benutzt.

Floodlight SDN Controller

Der *Floodlight OpenFlow Controller* [flo17] ist ein Open SDN Controller, der über die *Southbound API* mittels *OpenFlow* mit den Switches kommuniziert. Der Controller läuft in Java und unterstützt *Open vSwitch*. Zur Kommunikation über die *Northbound API* wird eine *REST* API angeboten. Dadurch können Anwendungen in beliebiger Programmiersprache geschrieben werden, die mithilfe dieser API den Trafficfluss im Kommunikationsnetz steuern können. Darüber hinaus enthält Floodlight verschiedene Java Module, die beim Start des Controllers mitgestartet werden. Diese können ebenfalls durch neue Java Module erweitert werden. Ein solches Modul ist ein sogenannter *Static Entry Pusher*, das zur Konfiguration von Flow Einträgen auf einem *OpenFlow* Switch genutzt wird. Wenn ein DoS-Angriff im Netz erkannt wird, kann mithilfe dieses Moduls bestimmte *Flows* auf die betroffenen Switches eingetragen werden. Die Erkennung von einem potentiellen Angriff und die darauffolgende Abwehr wird im nächsten Abschnitt behandelt. Nach der Installation des Floodlight Controller müssen alle Switche mit diesem verbunden werden, damit eine Kontrolle möglich ist. Dies geschieht durch `ovs-vsctl set-controller <Switchname> tcp:<Floodlight IP>:<PORT>`.

4.2.2 Erkennung und Abwehr von DoS-Attacken

Der Floodlight Controller kann in bestimmten Intervallen über die *REST API* nach Trafficstatistiken abgefragt werden, um einen DoS-Angriff zu erkennen [GHB+15]. Dabei erhält

der Controller die Metriken von den Switches, indem er diese über *OpenFlow* abfragt. Allerdings ist das Abfragen von Switches durch das sogenannte *pull*-Verfahren ineffizient, da der Controller jeden Switch abfragen und die Daten verarbeiten muss. Diese Daten kommen unter anderem durch das periodische Abfragen auch verzögert und nicht in Echtzeit an. Darüber hinaus muss der Cotroller neben dem *OpenFlow*-Traffic zur Kontrolle des Netzwerks nun auch noch Messdaten der Switche verarbeiten. Bei einer großen Anzahl an Switches im Netzwerk, kann also der Controller überlastet werden und somit einen DoS-Angriff verzögert erkennen. Wenn die Switches allerdings Metriken kontinuierlich an eine zentrale Stelle zur Analyse senden (*push*-Verfahren), ist die Messungslatenz geringer und die Daten werden beim Ereignis in Echtzeit gesendet. Es macht also Sinn die Messung und die Kontrolle des Netzwerkes voneinander zu trennen, sodass die Messung in der *Data Plane* vorgenommen wird und der Controller in der *Control Plane* nur für die Kontrolle des Traffics zuständig ist.

sFlow-RT [sfl17] ist solch eine Anwendung, das für die Analyse und Überwachung von Netzwerkpaketen in Echtzeit im *Software Defined Networking* verwendet werden kann. *sFlow-RT* besteht aus einer *Analytics Engine* und *sFlow-RT Agenten*. Jeder Switch wird mit einem *Agent* konfiguriert, die kontinuierlich Messwerte an die *Engine* senden. Da, aufgrund dieses *Push*-Systems nun jeder Switch seine eigenen Daten an eine zentrale Stelle sendet und nicht ein *Controller* von allen Switches Daten abfragen muss, wird durch diese Lastverteilung, der Overhead und die Latenz gering gehalten. Gleichzeitig wird dadurch die Performance des Gesamtsystems erhöht. Die gesammelten Metriken sind über die *Northbound API*, welche wie bei Floodlight aus einer *REST API* besteht, einsehbar. Zusätzlich bietet *sFlow-RT* auch eine *JavaScript API* an. Mithilfe dieser APIs können *sFlow-RT* Anwendungen geschrieben werden, die verschiedene DoS-Attacken erkennen können. Die *sFlow-RT* Architektur ist in 4.8 dargestellt. Damit die Switche Daten an die *sFlow-RT*

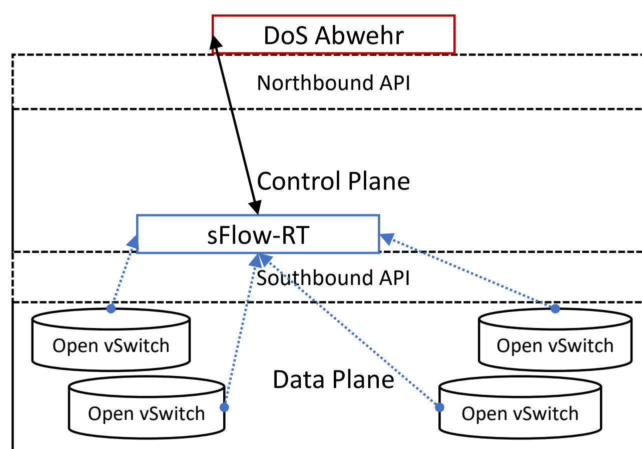


Abbildung 4.8: sFlow-RT Architektur.

Analytics Engine senden können, wird jeder Switch durch folgenden Befehl mit einem

sFlow-RT Agent konfiguriert: `ovs-vsctl -- --id=@sflow create sflow agent=<NIC> target=\"<sFlow-RT IP>:<PORT>\" sampling=<int> polling=<int> -- -- set bridge <Switchname> sflow=@sflow`. Das *Polling* Intervall wird genutzt, um in einem bestimmten Zeitintervall (Standardeinstellung: 30 Sekunden) die Anzahl aller Pakete, die der Switch verarbeitet hat, der *Analytics Engine* zu melden. Dabei werden in diesem Interval die Metriken, wie *link speed*, *link state*, aktueller Traffic, etc. gemeldet. *sFlow-RT* basiert auf *Sampling*, um den Netzwerktraffic zu charakterisieren [Pha02]. Mit der *Sampling* Rate wird eingestellt, dass 1 aus N Paketen zur Analyse an die *Analytics Engine* gesendet wird. Aufgrund des Paket *Samplings* wird eine *Frames* Metrik berechnet, die schneller auf einen Trafficanstieg reagiert als das Exportieren von *Counter* in einem bestimmten *Polling* Intervall. Daher sollte das *Polling* Intervall auf der Standardeinstellung belassen werden, um das Netzwerkrauschen und den Overhead zu verringern. Dennoch wird durch die asynchrone Ankunft von *Sampling* Daten, unabhängig vom *Polling* Intervall, eine Veränderung des Traffics in Echtzeit registriert. Durch längere Polling Intervalle und Paket *Sampling* wird eine effektive und skalierbare Möglichkeit geschafft, um das SDN-Netzwerk zu überwachen [sfl13]. Durch die Kombination von *sFlow-RT* und *Floodlight* können somit DoS-Angriffe erkannt und abgewehrt werden. Dieser Prozess wird im Folgenden beschrieben.

Im Durchschnitt wird 1 aus N Paketen von allen Agenten der *OpenFlow*-Switches in ein *sFlow datagram* verpackt. Dieses enthält alle notwendigen Informationen, wie Paket-Header, Input/Output Ports, Quell-IP, Ziel-IP und sendet dieses per UDP an die *sFlow-RT Analytics Engine* zur Analyse. Eine SDN Anwendung kann nun entweder von extern die *REST API* verwenden oder eingebettet sein und die interne *JavaScript API* verwenden. Eine JavaScript Anwendung ist für die Erkennung und das Starten der Abwehr verantwortlich. Diese definiert zunächst sogenannte *Flows*. Ein *Flow* besteht aus mehreren Paketattributen, die dazu genutzt werden, um das analysierte Paket zu klassifizieren. Für die verschiedenen DoS-Angriffstypen (s. Kapitel 2.1.1), sind also verschiedene *Flows* notwendig. 4.3 zeigt die definierten *Flows* *ddos_syn* und *ddos_icmp* in JavaScript für die Zuordnung von DoS-SYN-Flood und DoS-ICMP-Flood.

```
1  setFlow('ddos_syn',
2    {keys:'ipsource,ipdestination', value:'frames', filter:'tcpflags
     ~.......1.&outputifindex!=discard'}
3  );
4
5  setFlow('ddos_icmp',
6  {keys:'ipsource,ipdestination', value:'frames', filter:'ipprotocol=1&
     outputifindex!=discard'}
7  );
8
```

Quellcode 4.3: Definition von Flows in JavaScript für die Erkennung von DoS-Attacken in sFlow-RT.

Für jeden passenden Flow werden jeweils die Quell- und Ziel-IP Adressen erfasst und die Anzahl der Paketframes pro Sekunde berechnet. Mithilfe des *Filter*-Ausdrucks werden die entsprechenden Angriffsklassen gefiltert. Damit bei *ddos_syn* nicht jede Art von TCP Traffic gefiltert wird, sondern nur potentielle SYN Flood Attacken, überprüft der Filter ob ein SYN Flag gesetzt ist. *Tcpflags* stellt in *sFlow-RT* eine binäre Darstellung des Flags-Feldes im TCP Header (s. Abbildung 4.9) dar. Es wird also überprüft, ob im zweiten Flag-Feld

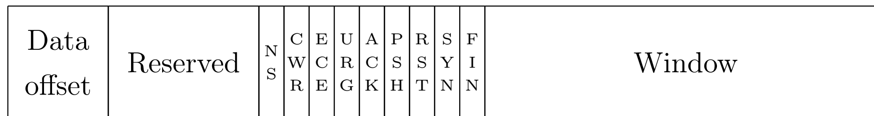


Abbildung 4.9: Ausschnitt aus der TCP Header Definition.

von rechts eine eins steht. Bei *ddos_icmp* wird der IPv4-Header überprüft. Die Protokollnummer eins entspricht dabei dem ICMP-Protokoll. Nachdem die entsprechenden Flows definiert sind, muss nun ein Schwellenwert für den Flow definiert werden. Beim Überschreiten dieses Werts, wird dann ein Ereignis ausgelöst. In diesem Fall wird ab dem Zeitpunkt der Ereignisauslösung ein DoS Angriff erkannt und eine Gegenmaßnahme gestartet. Dazu muss in JavaScript eine *Event Handler Funktion*, wie in 4.4 gezeigt, implementiert werden, welche bei jedem Eintritt des *Events* eine `handleBlock()` Funktion aufruft, um den Angriff zu stoppen.

```
1  setEventHandler(function(evt) {
2
3  switch(evt.metric) {
4       case 'ddos_syn':
5         handleBlock(evt,"6");
6                break;
7       case 'ddos_icmp':
8         handleBlock(evt,"1");
9                break;
10   }
11 },['ddos_syn', 'ddos_icmp']);
12
```

Quellcode 4.4: Registrierung des Event Handlers in sFlow-RT.

Nun beginnt die Abwehr des DoS-Angriffs. `HandleBlock()` ruft dafür ein externes Skript auf, welches über die *Northbound REST API* von Floodlight entsprechende Flows auf den Switches hinzufügen kann, um den Traffic vom Angreifer zu blockieren. Dazu wird zunächst der Switch ermittelt, mit dem die angreifende IP verbunden ist. Dies geschieht durch ein einfaches GET zur Geräte-API `<Floodlight IP:PORT>/wm/device/`, das alle vom Controller verwalteten Switches ausgibt. Anschließend kann über das *Static Entry*

Pusher Modul von Floodlight, welches ebenfalls über eine *REST API* verfügt, ein Flow auf dem entsprechenden Switch mittels *OpenFlow* eingetragen werden. Dabei wird über die API, der Name des Flows, Switch-ID, Ethernet Typ, IP Protokoll, Priorität des Flow sowie die Quell-IP des Angreifers übergeben (s. 4.5).

```
1 curl -X POST -d '{"name":"<flowname>", "switch":"<switchid>", "eth_type":"
    <ethtyp>", "ip_proto":"<ip protocol>", "priority":"<prioritaet>", "
    ipv4_src":"<ip address>", "active":"true"}' http://localhost:5050/wm/
    staticentrypusher/json
2
```

Quellcode 4.5: Flow zum Verwerfen von Paketen in Floodlight.

Mit `"output":"<switchport>"` wird in dem Flow noch zusätzlich angegeben, über welchen Port dieser Traffic weitergeleitet werden soll. Wenn keine Angabe in dem Flow erfolgt, werden die Pakete verworfen. Dies führt dazu, dass der entsprechende Angriff nun geblockt wird, da der Switch alle ankommenden Pakete verwirft und nicht weiterleitet. Der komplette Prozess zur Erkennung eines DoS-Angriffs durch *sFlow-RT* und die anschließende Abwehr durch den *Floodlight Controller* ist in 4.10 skizziert.

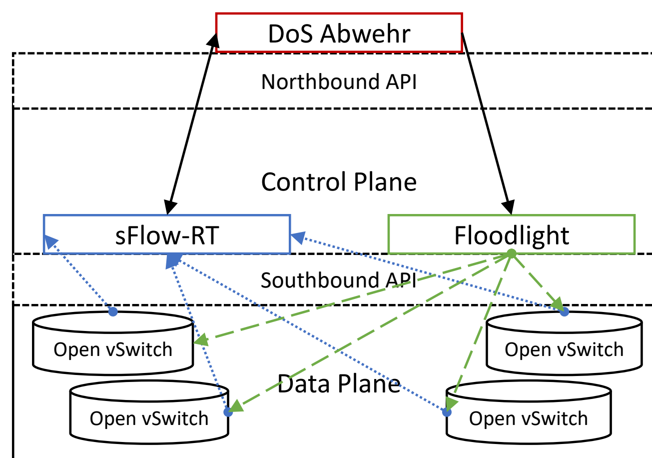


Abbildung 4.10: Erkennung und Abwehr eines DoS Angriffs durch *sFlow-RT* und *Floodlight*.

Die durchgängig virtualisierte Infrastruktur für die Ausführungsplattform und das Kommunikationsnetz, wird in Abbildung 4.11 dargestellt. Die Ausführungsplattform besteht aus einer *disaggregierten Domain 0* mit Sicherheitsmechanismen für die Erkennung einer Integrationsverletzung von laufenden VMs. Das virtualisierte Kommunikationsnetz besteht aus *Open vSwitches* in einer isolierten *Driver Domain*. Der *SDN Controller* und *sFlow* befinden sich in eigenen VMs mit Sicherheitsmechanismen für die Erkennung und Abwehr von DoS-Angriffen auf das Kommunikationsnetz. Diese Sicherheitsmechanismen werden im nächsten

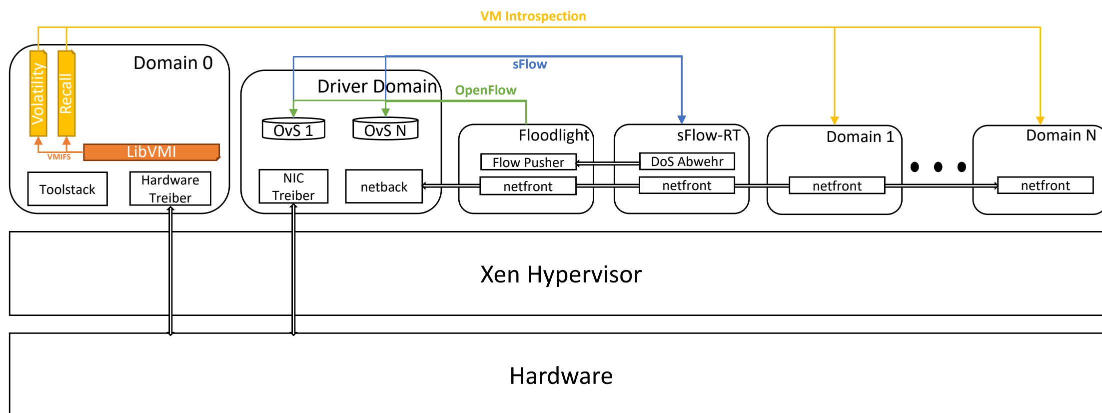


Abbildung 4.11: Durchgängig virtualisierte Infrastruktur für die Ausführungsplattform und das Kommunikationsnetz mit Sicherheitsmaßnahmen.

Kapitel evaluiert. Vorher werden im nächsten Abschnitt noch kurz die Sicherheitsaspekte der verwendeten Technologien selber betrachtet.

4.3 Sicherheitsaspekte der Virtualisierung

Neben den Vorteilen, die die Virtualisierung bietet, gibt es auch potentielle neue Sicherheitsrisiken durch Nutzung der Virtualisierungstechnik. Dieser Abschnitt behandelt die Sicherheitsaspekte der verschiedenen Virtualisierungstechniken.

Hypervisor

Der Hypervisor stellt als abstrahierende Schicht, die wichtigste Komponente der Virtualisierung dar. Dementsprechend ist es notwendig, dass die Angriffsfläche dieser klein gehalten wird. Dennoch erscheinen immer wieder durch Bugs im Hypervisor neue *Exploits*, die diese ausnutzen, um aus einer virtuellen Maschine auszubrechen oder den Hypervisor zum Absturz zu bringen [cve17]. Der Angriff auf den Xen Hypervisor erfolgt dabei über das *Hypercall Interface*. Aufgrund eines Programmierfehlers war es beispielsweise durch CVE-2012-3495 [CVE12] (*Common Vulnerabilities and Exposures (CVE)*) möglich, bei einer wiederholten Ausführung der Funktion *physdevop_get_free_pirq* des *physdev_op* Hypercalls in einer VM, einen DoS Angriff auf den Hypervisor zu starten und diesen zum Absturz zu bringen [MPA+14]. Die Funktion alloziert der aufrufenden VM ein *PCI Interrupt Request (PIRQ)*. Dabei gibt die Funktion als Rückgabewert einen Arrayindex zurück, wenn ein verfügbarer *PIRQ* existiert. Der Grund für den Bug lag in der fehlenden Überprüfung des Rückgabewerts. Der Rückgabewert wird als Arrayindex genutzt, auch wenn es einen negativen Fehlerwert zurückgibt, und zwar dann wenn kein verfügbarer *PIRQ* existiert. Dadurch wird in einen ungültigen Speicherbereich geschrieben, was letztendlich zum Absturz des Hypervisor führt. Durch das wiederholte Allozieren eines *PIRQ*, kann somit gezwungen

werden in einen ungültigen Speicherbereich zu schreiben und den Hypervisor zum Absturz zu bringen (s. Abbildung 4.12). Dieser Bug betraf die Xen Versionen 4.1.x. Durch einen veröffentlichten Patch seitens Xen, wurde dieser Bug anschließend geschlossen.

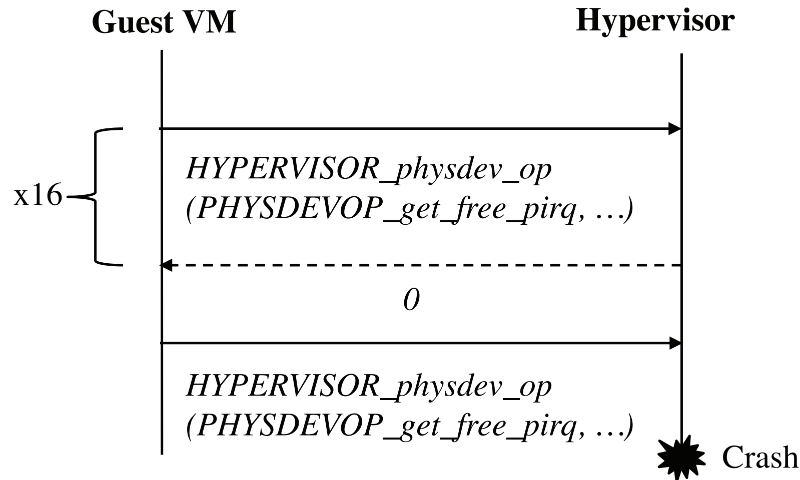


Abbildung 4.12: Mehrfache Ausführung der *physdevop_get_free_pirq* Funktion des Hypercalls *physdev_op*, um den Hypervisor zum Absturz zu bringen (CVE-2012-3495). [MPA+14]

Damit solche Bugs in den *Hypercall Interfaces* nicht ausgenutzt werden, bis Xen Sicherheitspatches veröffentlicht, können Hypercallaufrufe von virtuellen Maschinen eingeschränkt werden. Xen bietet dafür mit *XSM-FLASK* [xsm17] ein Sicherheits-Framework an, wodurch eine feingranulare Kontrolle der virtuellen Maschinen möglich ist. Damit lassen sich *Policies* schreiben, die zulässige Interaktionen zwischen VMs, Hypervisor und Ressourcen zulassen und somit Hypercalls einschränken, die eine VM ausführen darf. Standardmäßig ist XSM in Xen nicht aktiviert, dies muss bei der Kompilierung von Xen expliziert aktiviert werden. Anschließend kann eine Policy mit entsprechenden Richtlinien definiert und kompiliert werden. Sobald die *Policy* aktiv ist, werden sogenannte *security labels* zu jeder VM in deren *config*-Datei definiert. Ein Beispiel dazu sieht wie folgt aus:

```
seclabel='system_u:system_r:domU_t'
```

Ein *security label* besteht dabei aus den Attributen *user (_u), role (_r) und type (_t)*. Jedes Sicherheitsmodul kann in der *modules.conf* aktiviert oder deaktiviert werden. Das Modul selber wird durch die Dateien *<name_des_modules>.te* und *<name_des_modules>.if* beschrieben. Die *.te* Datei enthält dabei die Beschreibung der *Policy* und die *.if* Datei Makros, die in der *.te* Datei benutzt werden. *Users* werden in einer separaten *users*-Datei definiert und können verschiedene Rollen zugewiesen werden. Die Typen und Rollen werden in der *.te*-Datei des jeweiligen Moduls definiert:

```
type domU_t;
role system_r;
```

Mit *type* kann definiert werden, welche Gruppe von Typen einen bestimmten *Hypercall* ausführen darf. Mithilfe von *allow* werden nun diejenigen *Hypercalls*, die für einen bestimmten Typen zugelassen sind definiert, das wie folgt aussieht:

```
allow <source type> <target type>:<security class> <hypercall>;
```

Hypercalls sind in Sicherheitsklassen, die in der Datei *access_vectors* definiert sind, unterteilt. Wenn nun beispielsweise *Domain 0* die Sicherheitsrichtlinie mit *load_policy*, welche sich in der Klasse *security* befindet, ändern darf, geschieht dies wie folgt:

```
allow dom0_t security_t:security {load_policy setenforce};
```

Mehrere *Hypercalls* können dabei in geschweiften Klammern angegeben werden. Abgelehnte Hypercalls werden in der Konsole durch *avc* ausgegeben. *Avc* steht für *Access Vector Cache*, welcher eine Historie von zugelassenen und abgewiesenen Hypercalls speichert, um diese in der Zukunft nicht erneut zu überprüfen und dadurch die *Performance* zu erhöhen. [xen16] Auch wenn die Nutzung von *XSM* Sicherheitslücken durch *Hypercalls* nicht vollständig ausschließt, wird durch die Einschränkung von auszuführenden *Hypercalls* die Sicherheit erhöht.

Neben diesem Anwendungszweck wird *XSM* auch dafür verwendet, um die Kommunikation bestimmter VMs untereinander oder deren Aktionen einschränken zu können. Mithilfe einer *Policy* können dafür verschiedene Richtlinien definiert werden. Wird beispielsweise die Standardpolicy verwendet, kann der Typ `isolated_domU_t` in dem *security label* angegeben werden, wenn die VM nur mit *Domain 0* kommunizieren darf und nicht mit anderen VMs. Mit `prot_domU_t` kann beispielsweise die Erstellung einer VM unterbunden werden.

Virtual Machine Introspection

Ein entscheidender Vorteil der *Virtual Machine Introspection* ist die Fähigkeit virtuelle Maschinen von außerhalb zu analysieren und somit von einem Rootkit unentdeckt zu bleiben. Da hierfür aber unter anderem die *System.map* Datei verwendet wird, um Speicheradressen des jeweiligen Kernel zu bestimmen, ist die Integrität dieser Datei sehr wichtig. Denn durch eine Modifikation der *System.map* können die Ergebnisse einer *VMI* verfälscht werden. Daher sollte das Profil für *Volatility* bzw. *Rekall*, direkt nach Erstellung der virtuellen Maschine angefertigt werden. Dadurch wird sichergestellt, dass diese Datei bzw. die VM zu dem Zeitpunkt nicht kompromittiert ist. Trotzdem ist es durch einen sogenannten *DKOM (Dynamic Kernel Object Manipulation)* oder *DKSM (Direct Kernel Structure Manipulation)* Angriff möglich, *VMI*-Techniken anzugreifen [JBZ+15].

Bei einem *DKOM* Angriff werden Kernel Objekte manipuliert, sodass diese falsche Informationen liefern. Zum Beispiel ist das Verstecken von Linux Kernel Modulen oder Prozessen, indem diese aus der Liste des jeweiligen *struct* (s. Kapitel 2.3.1) gelöst werden, ein klassischer *DKOM* Angriff [JBZ+15]. Dies ist in Abbildung 4.13 dargestellt. Allerdings kann dieser Angriff zwar für VM-interne Sicherheitsanwendungen verborgen bleiben, da

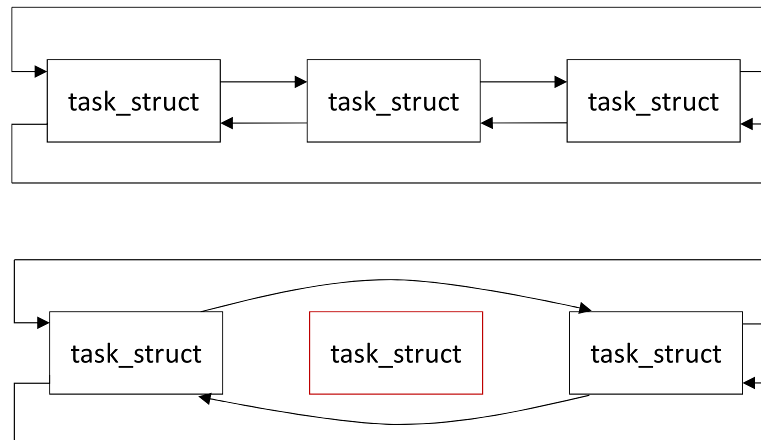


Abbildung 4.13: Linux Taskliste vor und nach einem DKOM Angriff.

diese auf die Betriebssystemstrukturen zugreifen, aber neuere VMI-Anwendungen, die eine externe Sicht auf die VM haben, können versteckte Module oder Prozesse trotzdem erfolgreich erkennen. Dadurch, dass *DKOM* Angriffe nur die interne Sicht aus der VM manipulieren und die externe Sicht vom Hypervisor unberührt ist, kann durch einen Vergleich dieser Sichten, Inkonsistenzen und somit ein potentieller Angriff erkannt werden. Diese Methode wird bei der Evaluation im nächsten Kapitel 5.2.1 gezeigt und näher beschrieben.

DKSM Angriffe hingegen können auch die externe Sicht und somit die Ergebnisse der VMI-Anwendungen beeinflussen [BJW+10]. Wenn eine *VMI* mithilfe einer vordefinierten *System.map* Datei stattfindet, wird erwartet, dass das Betriebssystem der VM sich der vordefinierten Struktur entsprechend verhält und dementsprechend den erhaltenen Informationen vertraut. Durch Manipulation dieser Kernel Strukturen, beispielsweise durch einen Tausch von Datenfeldern des gleichen Typs [BJW+10], interpretiert die *VMI*-Anwendung diese Daten falsch. Allerdings, muss für so einen Angriff ein Kernel Modul geladen werden, damit die Strukturen manipuliert werden können. In [BJW+10] wird ein *DKSM* Angriff gegen *VMI* gezeigt, indem ein für diesen Zweck geschriebenes Kernel Modul geladen wird. Damit solch ein Angriff erkannt werden kann, können Hardwarestrukturen und *Events* genutzt werden. Dadurch werden Kernel Module beim Laden erkannt (s. Kapitel 4.1.2, sowie das nächste Kapitel 5.2.1), wodurch ein *DKSM* Angriff mithilfe eines Kernel Moduls auch erkannt werden würde.

Software Defined Networking

Neben den Vorteilen die *Software Defined Networking*, durch die Entkopplung des Netzwerkes und dem zentralen Controller, mit sich bringt, birgt das Konzept auch Schwachstellen, die nun näher betrachtet werden. Der SDN Controller ist die Hauptkomponente im Netzwerk und somit verantwortlich für die gesamte Verwaltung des Netzes. Bei einem Ausfall

oder Kompromittierung des Controllers, kann das gesamte Netz gefährdet werden. Der Controller kann daher zu einem *Single Point of Failure* werden [ANYG15]. Es können aber verschiedene Sicherheitsmaßnahmen getroffen werden, um dies zu vermeiden. Diese werden im Folgenden beschrieben.

Zunächst können die Verbindungen zwischen dem *Controller* und den OpenFlow Switches mit TLS/SSL verschlüsselt werden, um *Man-in-the-Middle-Angriffe (MITM)* zu verhindern. In *Floodlight* müssen dafür in der Konfiguration folgende Zeilen gesetzt werden:

```
net.floodlightcontroller.core.internal.OFSwitchManager.keyStorePath=/path/
  to/your/keystore-file.jks
net.floodlightcontroller.core.internal.OFSwitchManager.keyStorePassword=
  your-keystore-password
net.floodlightcontroller.core.internal.OFSwitchManager.useSsl=Yes
```

Damit dies funktioniert, müssen für *Floodlight* und *Open vSwitch* Zertifikate, private und öffentliche Schlüsselpaare erstellt werden. Durch Zertifikate werden die Vertraulichkeit, Authentizität und Integrität von Daten geschützt. Da *Floodlight* in Java geschrieben ist, kann hier das *Java Keytool* verwendet werden, um ein *Java Keystore* zu erzeugen. Der *Keystore* enthält danach die Schlüssel und das Zertifikat, welche mit einem Passwort geschützt sind. Das Zertifikat von *Floodlight* muss anschließend *OvS* zur Verfügung gestellt werden, damit dieser den öffentlichen Schlüssel von *Floodlight* verifizieren kann. In *OvS* kann das Schlüsselpaar und Zertifikat mithilfe des Kommandozeilenprogramms `ovs-pki` erstellt werden. Mit `ovs-vsctl set-ssl` müssen nun dem *OvS*, sein privater Schlüssel, sein Zertifikat und das Zertifikat von *Floodlight* bekanntgemacht werden. Zum Schluss wird das Zertifikat von *OvS* noch dem *Floodlight Controller* bereitgestellt, damit dieser ebenfalls den öffentlichen Schlüssel von *OvS* verifizieren kann. Dies wird durch ein Importieren des Zertifikats in den *Java Keystore* bewerkstelligt. Nun ist jede Kommunikation zwischen dem Controller und den Switches vor *MITM-Angriffen* geschützt.

Damit der SDN Controller bei einem Ausfall nicht zu einem *Single Point of Failure* wird, sollte die Zuverlässigkeit und Verfügbarkeit der *Control Plane* durch Fehlertoleranz erhöht werden. Dies kann beispielsweise durch einen zweiten *Backup Controller* geschehen, der bei einem Ausfall des primären Controller eingreift und das Netzwerk somit aufrecht erhält (s. auch Kapitel 5.3.2). Dafür müssen beide Controller Nachrichten miteinander austauschen können, damit sich der sekundäre Controller, bei einer Verbindungstrennung des primären Controllers, mit allen Switches verbinden kann. In Floodlight ist dafür ein *FT Modul* vorhanden, welches konfiguriert und genutzt werden kann [ft16]. Das Modul verwendet einen *Java Keystore*, für den Nachrichtenaustausch zwischen zwei Controllern. Beim Start des primären Controller werden die Zustände aller Switches auf *ROLE_MASTER* gesetzt und beim sekundären Controller sind die Initialzustände auf *ROLE_SLAVE*. Bei einem erreichbaren primären Controller, ist somit der sekundäre Controller mit keinem der Switches verbunden. Sobald der primäre Controller die Verbindung verliert (*disconnection*

event), setzt das Modul des sekundären Controllers alle Switches, die mit diesem verbunden waren, auf *ROLE_MASTER*. Durch diese Fehlertoleranz, ist bei einem Ausfall des SDN Controller das Netzwerk nicht gefährdet.

Da der SDN Controller eine virtuelle Maschine ist, kann von der Hochverfügbarkeit, als Alternative zur Absicherung bei einem Ausfall, profitiert werden. Dazu kann unter Xen, *Remus* [CLM+08] verwendet werden. *Remus* bietet ein hohes Maß an Fehlertoleranz, sodass bei einem Ausfall einer VM, die Ausführung auf einer alternativen Maschine fortgesetzt werden kann. Dazu wird die VM repliziert und während der Ausführung fortlaufend migriert. Die primäre VM muss dafür nicht modifiziert werden und die sekundäre VM ist dabei eine exakte Kopie der VM, sodass ein Ausfall keine Auswirkungen auf den Systemablauf hat. Im Kontext von cyber-physischen Systemen, die zeitkritische Anforderungen einhalten müssen, sollte der Replikationszeitpunkt explizit durch die VM bestimmt werden können. Dies wird durch eine Erweiterung von *Remus*, in [JS15] erreicht.

Neben dem SDN Controller, können auch die Switches in der *Data Plane* durch einen Link- oder Portausfall die Verbindung kurzzeitig verlieren. Der SDN Controller würde dann eine Alternativroute berechnen, um die Verbindung zwischen den Switches wieder herzustellen. Falls im Netz statisch vorkonfigurierte Routen existieren, können die sogenannten *Fast-Failover (FF) OpenFlow Gruppen* verwendet werden, um bei einem Link- oder Portausfall einen vorkonfigurierten Alternativpfad zu wählen, anstatt eine neue Route beim Controller anzufragen. Dadurch wird die Zeit für die Wiederherstellung der Verbindung deutlich reduziert [VAVAK14]. Mithilfe von *Fast-Failover OpenFlow Gruppen* können bestimmte Ports beobachtet werden. Dazu hat jede *FF Gruppe* eine Liste von *Buckets*. Jedes *Bucket* hat eine Aktion und einen zu beobachtenden Port definiert. Wenn nun dieser primäre Port *offline* ist, wird ein anderes *Bucket* und somit auch ein anderer Port automatisch aus der Liste gewählt und die Verbindung wird dadurch nicht unterbrochen. Solch ein Beispiel ist in Abbildung 4.14 dargestellt. Die FF Gruppe von Switch 1 enthält 3 Buckets,

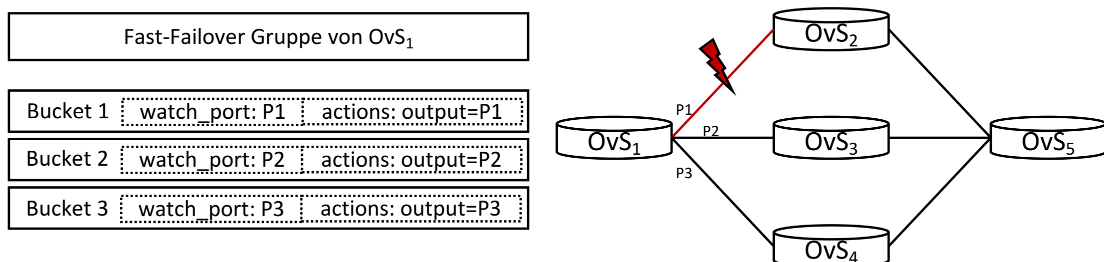


Abbildung 4.14: Fast-Failover OpenFlow Gruppe an einem Beispiel.

um die Ports P1-P3 zu beobachten. Wenn nun P1 ausfällt, wird zu einem anderen *Bucket* gewechselt, welcher einen *watch_port* enthält, der *online* ist. Anschließend wird die entsprechende Aktion des *Buckets* ausgeführt und die Daten werden somit über den jeweiligen

Port weitergeleitet. Dadurch kann direkt in der *Data Plane* eine Entscheidung getroffen werden, ohne dass der Controller vorher aktiv werden muss.

5 Evaluation

In diesem Kapitel wird die virtualisierte Infrastruktur mit den konzipierten Sicherheitsmaßnahmen evaluiert. Dafür werden zunächst entsprechende Szenarien definiert und anschließend implementiert (5.1), um die Sicherheit der Ausführungsplattform unter Verwendung von *Virtual Machine Introspection* und die Sicherheit des Kommunikationsnetzes mit dem *Software Defined Networking* Konzept, auszuwerten. Die Evaluation ist dabei in zwei Abschnitte unterteilt. Im ersten Teil (5.2) wird ein rudimentärer Nachweis der funktionalen Eigenschaften gegeben. Im zweiten Teil (5.3) werden die nichtfunktionalen Eigenschaften evaluiert. Die Evaluation wird auf folgender Server-Umgebung durchgeführt:

Hardware
Dell PowerEdge R620
Intel(R) Xeon(R) CPU E5-2650 v2 @ 2.60GHz
16GB RAM

OS
Ubuntu 14.04.5 LTS (Trusty Tahr)
Linux Kernel 4.9.0

Hypervisor
Xen 4.9.0

Netzwerk
Open vSwitch 2.7.1
Floodlight Controller 1.2
sFlow-RT 2.1

5.1 Definition der Szenarien

Damit die anfangs, in Kapitel 2.1.1, erwähnten typischen Angriffe auf diese virtualisierte Infrastruktur ausgeführt werden können, um diese zu evaluieren, müssen unterschiedliche Szenarien definiert werden. Für die Evaluation von *Virtual Machine Introspection* wird dafür ein *VMI Szenario* (5.1.1) und für *Software Defined Networking* respektive ein *SDN Szenario* (5.1.2) definiert. Die Basistopologie bleibt dabei in beiden Szenarien gleich: Ein vermaschtes Netz von Switches über das die VMs kommunizieren können. Das vermaschte Netz ist eine häufig verwendete Topologie, um eine hohe Konnektivität und Redundanz beim Ausfall eines Switches zu gewährleisten. Das Netzwerk besteht aus acht virtuellen

Switches ($S_1 - S_8$) und acht virtuellen Maschinen ($VM_1 - VM_8$), wie in Abbildung 5.1 dargestellt. Durch mehrere *Links* zwischen den Switches ist zwar eine hohe Verfügbarkeit

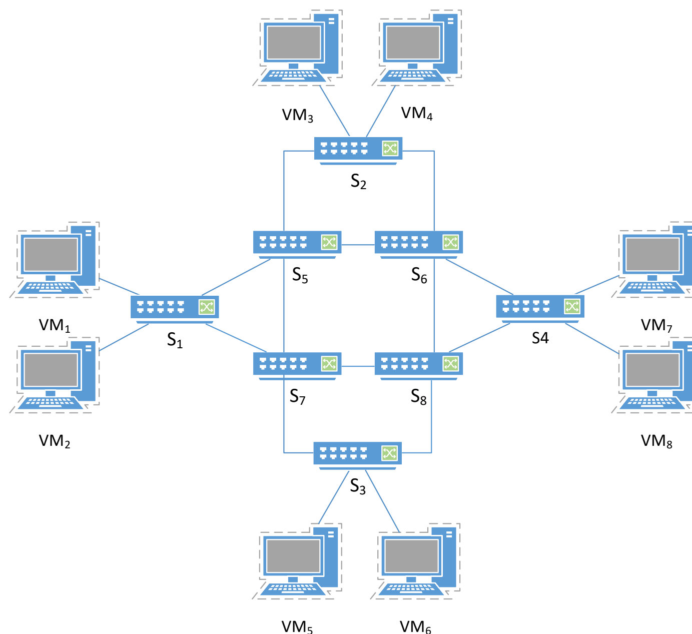


Abbildung 5.1: Basis-Netzwerktopologie für die Definition der Szenarien.

gewährleistet, allerdings besteht die Gefahr eines *Netzwerk-Loops*, was dazu führt, dass das Netzwerk mit der Zeit, aufgrund eines *Broadcast-Sturms*, einbrechen kann. Auf allen Switches ist daher das *Spanning Tree Protocol (STP)* aktiviert, um solch einen *Loop* im Netzwerk zu verhindern. Dadurch wird bei jeder Änderung der Topologie, durch den SDN Controller gewährleistet, dass sich keine Schleife zwischen den Switches bildet. Wenn ein *Link* ausfallen sollte, wird der SDN Controller dynamisch die anderen *Links* wieder aktivieren, sodass eine Konnektivität gewährleistet ist. Eine Beispiel-Topologie ist in Abbildung 5.2 dargestellt.

Im Folgenden werden die entsprechenden Szenarien für die Evaluation der Ausführungsplattform und des Kommunikationsnetzes definiert.

5.1.1 VMI Szenario

Dieses Szenario dient der Evaluation der Ausführungsplattform, um Angriffe auf die Integrität zu erkennen. Zu diesem Zweck wird VM_1 durch das Instanziieren von *Linux Kernel Rootkits* kompromittiert. Dazu werden zu unterschiedlichen Zeitpunkten verschiedene *Rootkits* geladen. In *Domain 0* wird eine *VMI*-Anwendung ausgeführt, welche versucht diese Rootkits zu erkennen. Dabei werden zwei unterschiedliche Techniken untersucht. Zunächst findet ein *passives Monitoring* statt, indem in einem bestimmten Intervall die VMs auf Rootkits gescannt werden. Anschließend wird ein *aktives Monitoring* durchgeführt, indem bestimmte *Events* beobachtet werden (s. Kapitel 4.1.2).

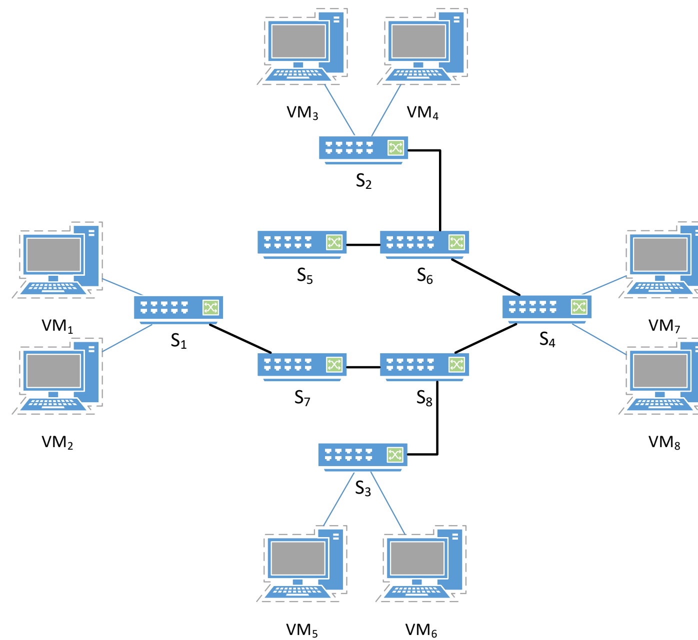


Abbildung 5.2: Basis-Netzwerktopologie mit aktiviertem STP-Protokoll.

5.1.2 SDN Szenario

In diesem Szenario wird das Kommunikationsnetz evaluiert, indem die Verfügbarkeit einer VM angegriffen wird. VM_4 sendet für eine gewisse Periode legitime Netzwerkpakete an VM_7. Währenddessen startet die kompromittierte VM_1, einen DoS-Angriff auf VM_7 mit dem Ziel diese zu überlasten, damit die Pakete von VM_4 den Bestimmungsort nicht erreichen und somit die Verfügbarkeit von VM_7 nicht mehr garantiert ist. Dieses Szenario ist in Abbildung 5.3 dargestellt.

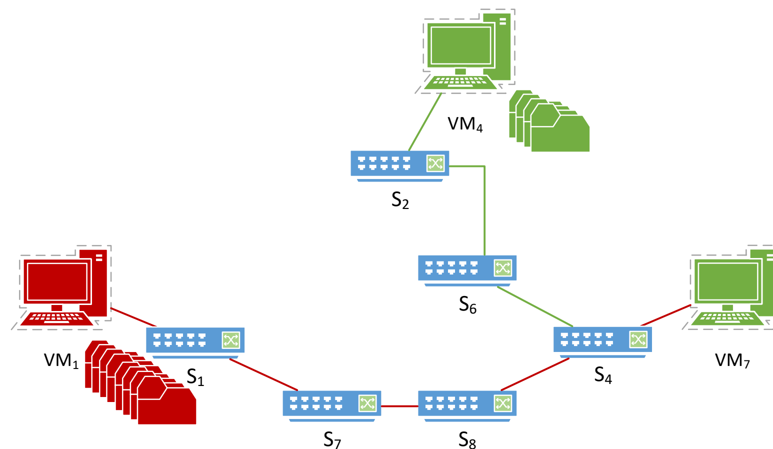


Abbildung 5.3: SDN Szenario.

5.2 Evaluation der funktionalen Eigenschaften

Dieser Abschnitt stellt ein *Proof of Concept* der verwendeten Technologien dar. Es wird gezeigt, dass durch *VMI* verschiedene Rootkits erkannt werden können und durch Nutzung von *SDN DoS*-Angriffe abgewehrt werden können. Auf die nichtfunktionalen Eigenschaften wird im nächsten Abschnitt 5.3 eingegangen.

5.2.1 VMI

In diesem Abschnitt wird das *Passive* und *Aktive Monitoring* durch *VMI* evaluiert.

Passives Monitoring

Zunächst wird die Technik des *passiven Monitorings* untersucht. Dazu wird in *Domain 0* in einem bestimmten Intervall ein *Bash*-Skript ausgeführt, welches durch *Virtual Machine Introspection* den Speicher aller laufenden virtuellen Maschinen nach Rootkits scannt. Dafür wird mithilfe von *LibVMI* und *VMIFS* der Speicher der betroffenen VM als Laufwerk *gemounted*. Anschließend werden die Forensik-Tools *Volatility* und *Rekall* genutzt, um diesen Speicher zu analysieren (s. auch Kapitel 4.1.2).

Rootkit Diamorphine Ähnlich wie der *task_struct* (s. Kapitel 2.3.1) bei der Liste von Prozessen, ist *module* ein *struct* welches Informationen zu den geladenen Modulen enthält. Diese Liste ist ebenso doppelt verkettet mit den jeweiligen *next* und *prev* Pointern [Lov10]. Der Linux Befehl `lsmod` listet alle geladenen Module auf, indem es die Datei */proc/modules* liest und somit die Liste der Module durchläuft. Das virtuelle Dateisystem *sysfs* des Linux-Kernels enthält *ksets* (*kernel object sets*) mit ihren jeweiligen *kobjects* (*kernel objects*). *module_kset* ist solch ein *kset*, welches Referenzen zu allen geladenen Kernel Modulen, sowohl aus dem *struct module* als auch aus dem *sysfs kset*, enthält. Die meisten Rootkits verstecken sich, indem sie sich von der Liste des *struct module* lösen, sodass keine next oder prev Pointer auf das Modul des Rootkits zeigen [vol12]. *Diamorphine* [dia15] ist solch ein Rootkit, welches sich nach dem Laden vor dem Kernel verbirgt. Allerdings können solche Rootkits, aufgrund der Referenzen aus *sysfs*, als ein potentielles Rootkit erkannt werden. Das Rekall Plugin *check_modules* [rek17b] durchläuft das *module_kset* und vergleicht diese Liste der Module mit der Ausgabe der Module aus `lsmod` und erkennt somit das Rootkit *Diamorphine*.

```
Scanned domain: debian1.
Hidden Module detected! 0xffffa02c4000 diamorphine
```

Die Ausgabe zeigt, dass das Rootkit als verstecktes Modul erfolgreich erkannt wird.

Rootkit Xingyiquan Mit Hilfe von *System Calls* können Anwendungen aus dem User Kontext heraus, vom Kernel bereitgestellte Funktionen ausführen. Die *System Call* Tabelle ist ein *Array* von Pointern und jeder zeigt auf einen *System Call Handler*. Rootkits manipulieren diese Einträge der Tabelle, um Dateien und Prozesse zu verstecken und somit den Betriebsablauf zu manipulieren. *Xingyiquan* [xin14] ist solch ein Rootkit. Es manipuliert mehrere *System Calls* und ist dadurch neben den grundlegenden Funktionen, wie sich zu verstecken und unerkannt zu bleiben, ebenfalls in der Lage dem normalen Nutzer *Root*-Rechte zu verschaffen oder Dateien und Ordner zu verstecken. Dazu werden beim Initialisieren des Kernel Moduls verschiedene *System Call Handler* mit eigenen *gehookt*. Das Volatility Plugin *linux_check_syscall* [vol17b] durchläuft und überprüft alle Einträge dieser *System Call* Tabelle und erkennt somit das Rootkit bzw. die *hooked System Calls*.

```
Scanned domain: debian1.
Hooked syscall detected!
 Table  Index  Syscall   Handler Address     Symbol
 64bit  2      open      0xffffffffa031ba80  HOOKED: xingyiquan/cr_open
 64bit  6      lstat     0xffffffffa031b770  HOOKED: xingyiquan/cr_lstat
 64bit  32     dup       0xffffffffa031b260  HOOKED: xingyiquan/cr_dup
 64bit  62     kill      0xffffffffa031b2c0  HOOKED: xingyiquan/cr_kill
 64bit  78     getdents  0xffffffffa031b180  HOOKED: xingyiquan/cr_getdents
 64bit  80     chdir     0xffffffffa031b530  HOOKED: xingyiquan/cr_chdir
 64bit  82     rename    0xffffffffa031b5f0  HOOKED: xingyiquan/cr_rename
 64bit  84     rmdir     0xffffffffa031b470  HOOKED: xingyiquan/cr_rmdir
 64bit  263    unlinkat  0xffffffffa031b6b0  HOOKED: xingyiquan/cr_unlinkat
```

Die Ausgabe zeigt, dass verschiedene *System Calls*, wie *open*, *kill*, etc. von dem Rootkit manipuliert werden und beim Aufruf, anstelle der eigentlichen Funktion, die vom Rootkit gewünschte Funktion ausführt, wie beispielsweise dem Nutzer erhöhte Rechte zu erteilen.

Aktives Monitoring

Beim *aktiven Monitoring* können bestimmte *Hardware Events* der VM durchgehend beobachtet werden. In diesem Fall wird mithilfe von *Interrupt Events*, der *System Call* `sys_init_module` beobachtet, um das Instanziieren des Rootkits *Diamorphine* zu erkennen (s. auch Kapitel 4.1.2). Sobald das Rootkit instanziiert wird, wird ein *Interrupt* ausgelöst.

```
Syscall sys_init_module is located at VA 0xffffffff810db690.
Received a trap event for syscall sys_init_module!
```

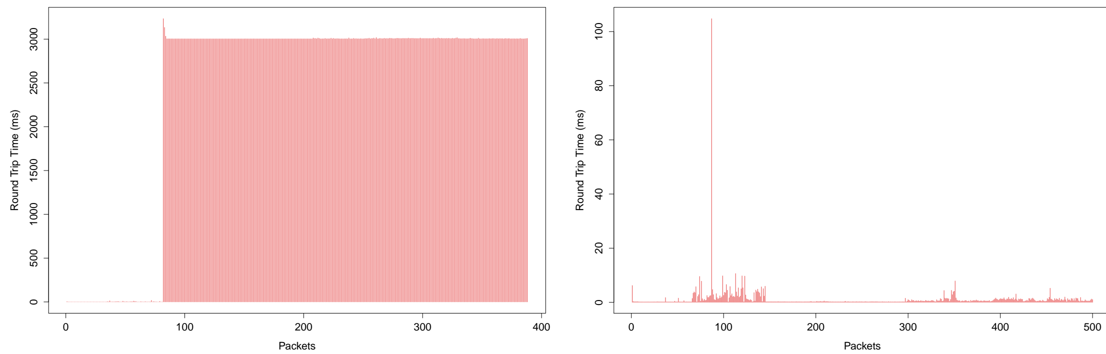
Die Ausgabe zeigt, dass das Laden eines Kernel Moduls erkannt wurde. Eine Betrachtung der nichtfunktionalen Eigenschaften beider *VMI*-Methoden findet in Abschnitt 5.3.1 statt.

5.2.2 SDN

Dieser Abschnitt soll funktional zeigen, dass DoS-Angriffe auf das Kommunikationsnetz erkannt und abgewehrt werden. Dazu wird eine *Client-Server Anwendung* verwendet. Auf einer VM läuft ein Client, der UDP Pakete sendet und auf der anderen VM läuft ein Server, der diese empfängt. Der Client erstellt dabei eine Datei mit den Sequenzzahlen der Pakete und der jeweiligen *Round-Trip-Time (RTT)*. Anhand der Antwortzeiten des Servers, können die Auswirkungen eines DoS-Angriffs erkannt werden. Dabei kann die Anzahl der zu sendenden Pakete und die Sendefrequenz am *Client* eingestellt werden. VM_4 sendet im 100 ms Intervall 500 UDP Pakete an VM_7. Mithilfe von *Hping* [hpi06] startet VM_1 währenddessen eine *SYN Flood DoS*-Attacke auf VM_7. *sFlow-RT* erkennt den Angriff und sendet einen entsprechenden *Flow* an den *Floodlight Controller*, um den Angriff zu blocken:

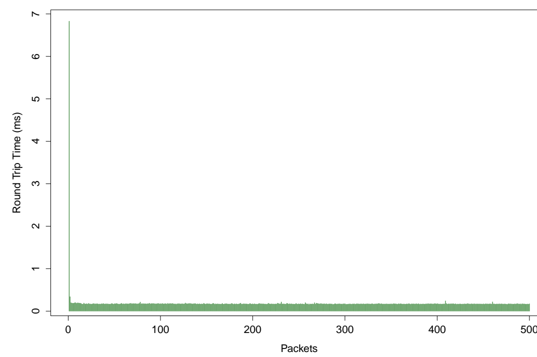
```
sFlow runtime: 1 ms. Pushing flow to controller to block attack 192.168.2.1-
SYN. Runtime: 30.17 ms.
```

Abbildung 5.4 veranschaulicht die *Round-Trip-Time (RTT)* der Pakete grafisch. 5.4(a) zeigt die Antwortzeiten des Servers, während des Angriffs ohne jegliche Sicherheitsmechanismen. Die Antwortzeiten sind so hoch, bis die Pakete 418-500 (s. auch Tabellenauszug 5.1) VM_7 schließlich nicht mehr erreichen. 5.4(b) zeigt den gleichen Angriff, diesmal jedoch mit eingeschalteten Sicherheitsmechanismen. Nachdem *sFlow-RT* den Angriff erkannt und die entsprechende Maßnahme zum Blocken des Angriffs eingeleitet hat, wird die anfangs etwas höhere Antwortzeit anschließend wieder geringer. Es ist kein Paket während des Angriffs verloren gegangen. Eine Betrachtung der nichtfunktionalen Eigenschaften findet in Abschnitt 5.3.2 statt.



(a) DoS Angriff ohne Abwehr

(b) DoS Angriff mit Abwehr



(c) Normale RTT ohne Angriff

Abbildung 5.4: Round Trip Time (RTT) von UDP Paketen im 100ms-Takt.

Pakete	RTT in ms
...	...
410	3004.170616
411	3008.926048
412	3004.501347
413	3003.959185
414	3003.736235
415	3005.086112
416	3007.085620
417	3008.830980

Tabelle 5.1: Round Trip Time (RTT) von UDP Paketen im 100ms-Takt während eines *DoS*-Angriffs.

5.3 Evaluation der nichtfunktionalen Eigenschaften

In diesem Abschnitt werden die nichtfunktionalen Eigenschaften der verwendeten Technologien evaluiert.

5.3.1 VMI

Für das Messen des Overheads einer *Virtual Machine Introspection*, wird die bereits erwähnte *Client-Server Anwendung* verwendet. Dabei muss die Sendefrequenz der Pakete niedrig eingestellt werden, weil der Overhead einer *VMI* gering ist und das Ereignis sonst keine sichtbaren Auswirkungen auf die Antwortzeiten hat. Es werden zwei Benchmarks für das *passive Monitoring* und eins für das *aktive Monitoring* ausgeführt.

Passives Monitoring

Beim *passiven Monitoring* einer VM, kann diese für einen konsistenten Speicherzugriff pausiert werden (s. Quellcode Ausschnitt 2.1). In der Zeit, während die VM pausiert ist, findet dann die eigentliche *Introspection* statt. Dies ist mit einem Overhead verbunden, da die Ausführung der VM in der Zeit unterbrochen ist. Dennoch ist durch das Pausieren keine absolute Konsistenz gewährleistet und die Vorteile sind im Verhältnis zum Overhead gering [SILB15]. Abbildung 5.5 zeigt die Antwortzeiten der UDP Pakete, die im 10 Millisekunden Takt gesendet werden, während solch eine *VMI* mit Pausieren der VM stattfindet. Dabei

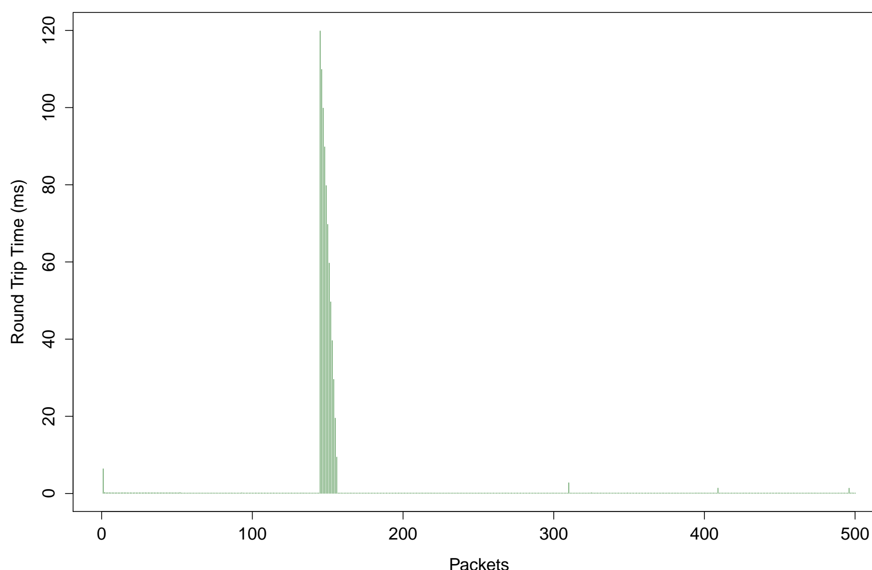


Abbildung 5.5: RTT von UDP Paketen im 10ms-Takt während einer VMI mit Pausieren der VM.

werden mithilfe der *VMI* die aktuell laufenden Prozesse ausgelesen. Bei einer längeren *In-*

trospection wäre der Overhead dementsprechend auch größer. *Volatility* und *Rekall* scannen den *gemounteten* Speicher der VM mithilfe von *VMIFS*, ohne diese Pausieren zu müssen. Daher wird die Leistung der VM nicht beeinträchtigt und es konnten keine Interferenzen erfasst werden, welche sich in Form von Latenzen äußern würden, wie in Abbildung 5.6 zu sehen ist.

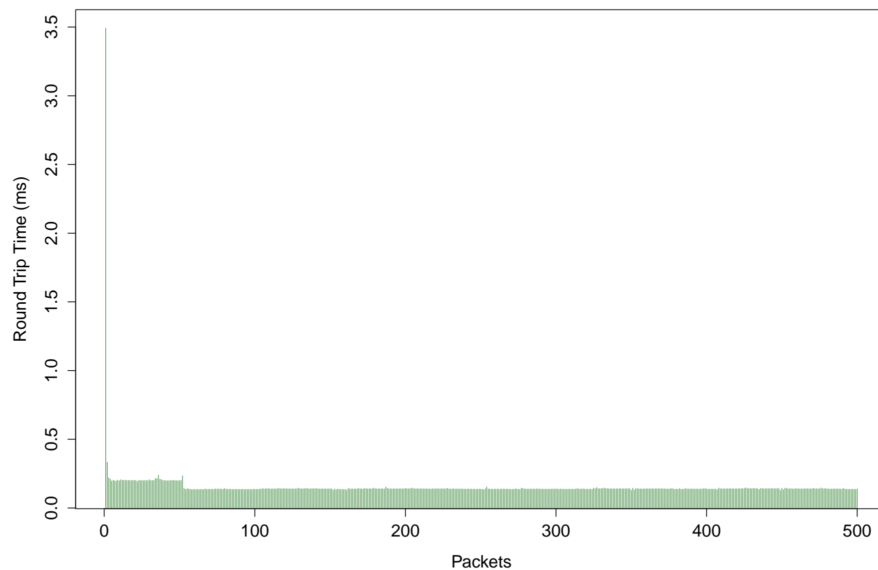


Abbildung 5.6: RTT von UDP Paketen im 10ms-Takt während einer VMI mit Volatility.

Aktives Monitoring

Beim *aktiven Monitoring* wird ein bestimmtes *Event*, wie beispielsweise das Laden eines Kernel Moduls, beobachtet. Da dieser beim Auslösen einen kurzzeitigen Interrupt auslöst, ensteht dadurch ein geringer Overhead. Abbildung 5.7 zeigt die Antwortzeiten der UDP Pakete, wenn das Rootkit *Diamorphine* geladen wird und somit der *Trap* des Events erfolgt.

Nun wird ein Dhrystone Benchmark [byt17] auf der inspizierten VM ausgeführt, um den CPU Overhead dieser VM zu ermitteln, wenn eine *VMI* stattfindet. Bei der *VMI* wird mit *Volatility* versteckte Module ausgelesen, um *Diamorphine* zu erkennen. Der Benchmark wird ausgeführt, während *Diamorphine* instanziiert wird und die *VMI* läuft. Ohne *VMI*, erreicht der Benchmark im Durchschnitt bei 5 Iterationen 29874715.6 lps (*loop per second*). Beim *passiven Monitoring* erreicht der Benchmark im Durchschnitt 29859293.1 lps. Beim *aktiven Monitoring* erreicht der Benchmark im Durchschnitt 29678992.6 lps. Die Best- und Worst-Case Iterationen sind zusätzlich zu den Durchschnittwerten in Tabelle 5.2 dargestellt. Damit werden die obigen Ergebnisse nochmals bestätigt, dass das *aktive Monitoring* bei der Erkennung eines Rootkits einen CPU-Overhead verursacht. Allerdings wird

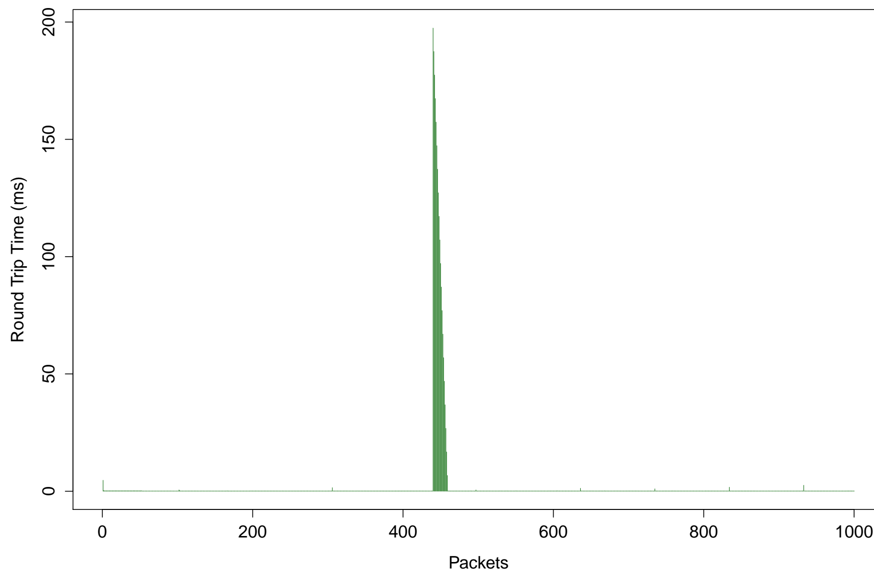


Abbildung 5.7: RTT von UDP Paketen im 10ms-Takt während einer aktiven VMI.

	Ohne VMI	Passive VMI	Aktive VMI
Durchschnitt	29874715.6	29859293.1	29678992.6
Best-Case	29886587.1	29865934.3	29723883.8
Worst-Case	29867822.5	29849398.3	29639181.0

Tabelle 5.2: Durschnitt-, Best- und Worst-Case Durchläufe aus 5 Iterationen des Dhrystone Benchmarks in *loop per second (lps).*

dadurch die Möglichkeit gegeben, bei einer Kompromittierung direkt benachrichtigt zu werden. Das *passive Monitoring* zu bestimmten Zeitintervallen hat den Nachteil, dass die Kompromittierung erst beim Scan auffällt. Dafür ist der CPU Overhead verhältnismäßig gering und die Latenzen in der Kommunikation nicht vorhanden.

5.3.2 SDN

Für die nichtfunktionale Evaluation von *SDN* wird das Netzwerktool *iPerf* [ipe17] verwendet, um einen realen und konstanten *Traffic* (5.00 Mbits/sec) zwischen VM_4 und VM_7 zu generieren. Mithilfe von *tcpdump* [tcp17] werden alle ankommenden Pakete zu VM_7 mitgeschnitten. Zunächst wird *sFlow* deaktiviert, damit ein DoS Angriff ohne Sicherheitsmechanismen evaluiert werden kann. Abbildung 5.8 zeigt, dass der *SYN Flood* von VM_1 ab Sekunde 33 bei VM_7 ankommt. Ab Sekunde 70 ist VM_7 so überlastet, dass keine Pakete mehr von VM_4 ankommen.

Transfer	**Bandwidth**
31.5 MBytes	2.94 Mbits/sec

Aufgrund des erfolgreichen DoS Angriffs, geht der *Traffic*, durch die resultierende Downtime, auf 2.94 Mbits/sec herunter.

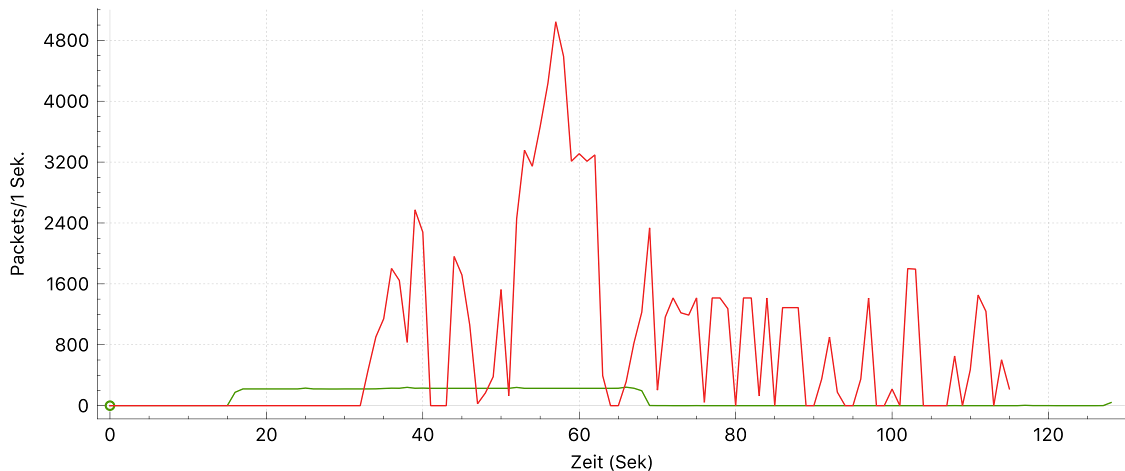


Abbildung 5.8: DoS Angriff ohne Sicherheitsmechanismen.

Mit aktiviertem *sFlow-RT* und einem definierten Schwellenwert für die Anzahl der Pakete, ab wann dieser als Angriff erkannt werden soll (s. Kapitel 4.2.2), wird der *SYN Flood* als Angriff erkannt und Maßnahmen zum Blockieren eingeleitet. Der Schwellenwert kann beliebig vom Administrator, je nach Anwendungszweck, in *sFlow-RT* eingestellt werden. Hier wurde der Schwellenwert auf 500 gesetzt und bei 560 von *sFlow-RT* als Angriff erkannt. Die Pakete des Angreifers kommen ab Sekunde 40 bei VM_7 an (s. Abb. 5.9). Ab Sekunde 65 kommen keine Pakete mehr von VM_1 an, aufgrund des *drop Flows* durch den SDN Controller. Im Vergleich zu Abbildung 5.8 kommen somit alle Pakete aus dem legitimen *Traffic* von VM_4 nach VM_7 an und der *Traffic* bleibt konstant.

Transfer	**Bandwidth**
53.6 MBytes	5.0 Mbits/sec

Die benötigte Zeit von *sFlow*, um den Angriff zu erkennen beträgt 1 Millisekunde. Die Zeit, um einen *drop Flow* zu generieren und an den Floodlight Controller zu senden 28,21 Millisekunden:

```
sFlow runtime: 1 ms. Pushing flow to controller to block attack
192.168.2.1-SYN. Runtime: 28.21 ms.
```

Da in dieser Zeit bereits Pakete von VM_1 angekommen sind, dauert es insgesamt 25 Sekun-

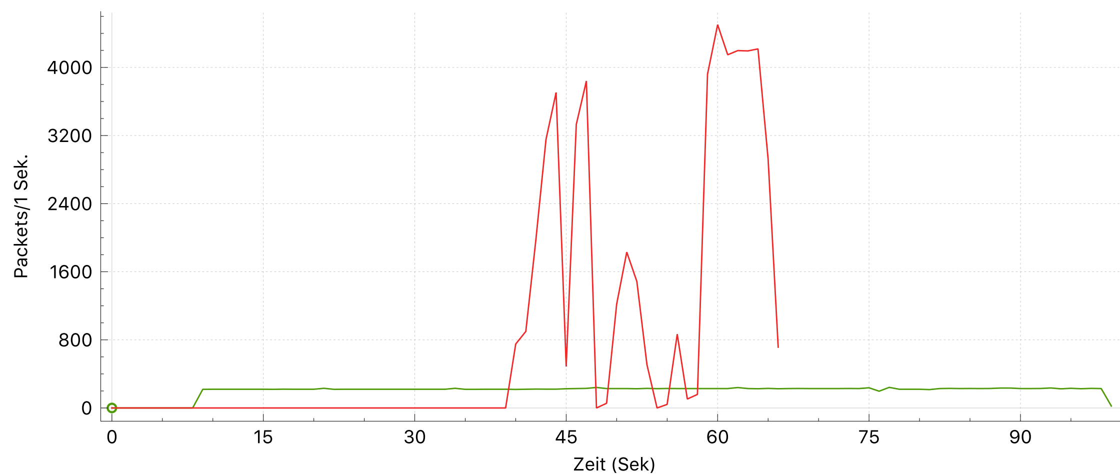


Abbildung 5.9: DoS Angriff mit Sicherheitsmechanismen.

den bis alle Pakete von VM_7 verarbeitet werden. Dadurch, dass der Angreifer aber geblockt wurde und nun keine weiteren Pakete mehr ankommen, wird der *Traffic* zwischen VM_4 und VM_7 nicht gestört und es kommen somit alle Pakete mit einem konstanten *Traffic* an.

Fehlertoleranz durch zweiten SDN Controller

Damit der SDN Controller bei einem Ausfall nicht zu einem *Single Point of Failure* wird, kann wie in Kapitel 4.3 erwähnt, ein zweiter Controller als Fehlertoleranz in *Floodlight* genutzt werden. Für die Evaluation der benötigten Zeit, um vom primären Controller zum sekundären Controller zu wechseln, wird *iPerf* zur Generation des *Traffic* genutzt, während der primäre Controller beendet wird. Die Abbildung 5.10 zeigt die Anzahl der

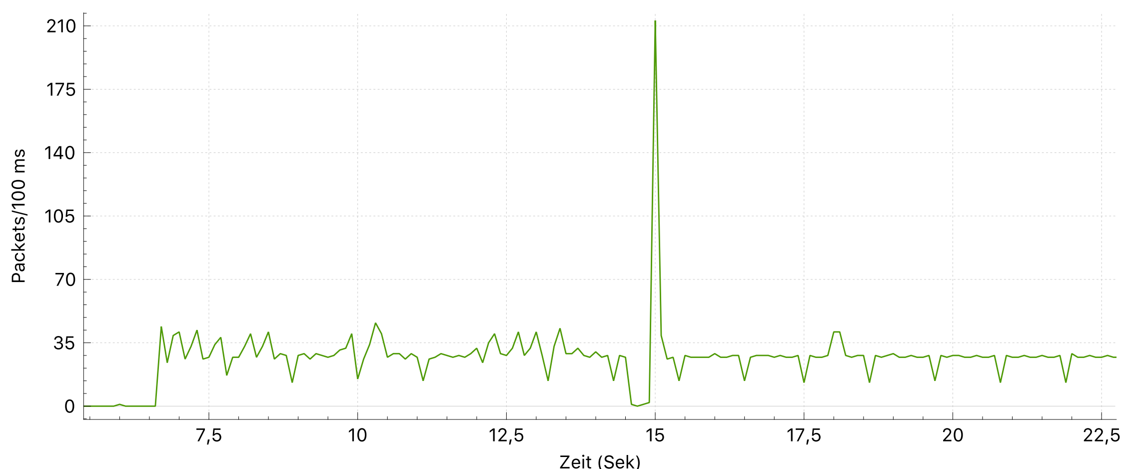


Abbildung 5.10: Fehlertoleranz durch sekundären SDN Controller, beim Ausfall des primären.

empfangenen Pakete im 100 Millisekunden Takt. In dem Intervall 14,6 - 14,9 Sekunden

kommen keine Pakete an. Der sekundäre Controller braucht somit 300 Millisekunden, bis er die Kontrolle übernimmt. Da *iPerf* die Pakete mit *TCP* versendet, werden diese, sobald die Verbindung wieder hergestellt wurde, erneut gesendet, was in der Abbildung den *Peak* ausmacht. Somit sind keine Pakete verloren gegangen und anschließend kehrt der *Traffic* wieder zum normalen Zustand zurück.

6 Fazit

Im ersten Teil dieser Arbeit wurden allgemeine Sicherheitsrisiken in cyber-physischen Systemen analysiert. Dadurch wurde die Motivation für diese Arbeit geschaffen, Sicherheitsaspekte virtualisierter cyber-physischer Systeme zu untersuchen, um zu überprüfen ob durch die Virtualisierungstechnik die Voraussetzungen, wie Fehlertoleranz und Sicherheit, für einen zuverlässigen Betrieb einer Infrastruktur für CPS gegeben sind. Dafür wurden zunächst die Grundbegriffe der Virtualisierung, von den Virtualisierungsarchitekturen bis hin zu den verschiedenen Virtualisierungstechniken, die der Xen Hypervisor unterstützt, erläutert. Danach wurden die Technologien *Virtual Machine Introspection (VMI)* und *Software Defined Networking (SDN)* vorgestellt, mithilfe dieser, Gefahren sowohl auf der virtualisierten Ausführungsplattform als auch im virtualisierten Kommunikationsnetz erkannt werden sollten.

Im zweiten Teil der Arbeit wurden Anforderungen an die virtualisierte Infrastruktur definiert. Die wichtigsten Anforderungen sind die Fehlertoleranz und Sicherheit des Systems, wobei sich diese Arbeit hauptsächlich mit den Sicherheitsaspekten auseinander gesetzt hat. Diese sollten sowohl auf der Ausführungsplattform als auch im Kommunikationsnetz gewährleistet werden. Dafür wurden, basierend auf die im ersten Teil vorgestellten Technologien VMI und SDN, Sicherheitsmaßnahmen konzipiert und implementiert. Es wurde hier ebenfalls betrachtet, welche Arten von neuen Sicherheitsrisiken sich durch die Nutzung der Virtualisierungstechnik ergeben können und wie diese minimiert werden können. Die implementierten Sicherheitsmaßnahmen wurden anschließend in diversen Szenarien funktional und nichtfunktional evaluiert.

Zusammenfassend kann festgestellt werden, dass die typischen Angriffe auf CPS in einer durchgängig virtualisierten Infrastruktur erfolgreich erkannt und entsprechend abgewehrt werden können. Ein Punkt der definierten Systemanforderungen im zweiten Teil dieser Arbeit, war die Effizienz der konzipierten Sicherheitsmaßnahmen. Die Erkennung bzw. Abwehr der untersuchten Angriffe sollten mit einer, im Kontext für CPS, vertretbaren Ressourcenauslastung und Verzögerung erfolgen. Eine Kompromittierung der Ausführungsplattform kann entweder in Echtzeit mit *aktiver VMI* oder durch ein *passives Monitoring* zu bestimmten Scan Intervallen, erkannt werden. Die erste Methode hat durch den Kontextwechsel der CPU, aufgrund des *Interrupts*, einen geringen *Overhead* und die zweite Methode eine Erkennung zum Zeitpunkt des Scans, aber keine größere Ressourcenauslastung auf der inspizierten VM. Es muss daher je nach Anwendungsfall abgewägt werden,

ob die Sicherheit oder Verzögerungen eine wichtigere Rolle darstellt. Durch die Anwendung des SDN-Konzepts in der Kommunikationsebene wurde eine hohe Flexibilität und Zuverlässigkeit erreicht. Angriffe auf das Kommunikationsnetz konnten in Echtzeit, 1ms mit sFlow, entdeckt und entsprechende Gegenmaßnahmen eingeleitet werden. Dadurch wurde der legitime Traffic zwischen den VMs nicht unterbrochen und alle Pakete haben das Ziel erreicht. Dennoch darf die Tatsache nicht außer Acht gelassen werden, dass Virtualisierung ebenfalls neue Sicherheitsrisiken, wie beispielsweise einen Ausbruch aus der virtuellen Maschine, mit sich bringen kann. Hier wurden einige Ansätze, wie die *Domain 0 Disaggregation* berücksichtigt, um diese Sicherheitsrisiken zu minimieren. Dadurch wird bei einem Ausbruch kein Zugriff auf eine privilegierte VM erlangt, was die Sicherheit des Systems erhöht. Aber viele Lösungen in diesem Bereich, wie beispielsweise *Xen Security Modules*, die *Hypercalls* einschränken, sind noch nicht vollständig ausgereift und befinden sich fortlaufend in Entwicklung. Hier besteht also noch Forschungsbedarf. Trotzdem überwiegen die Vorteile der Virtualisierung durch einfache und flexible Integration von Systemkomponenten, sowohl Kostenreduzierung als auch eine hohe Skalierbarkeit von Hardware-Ressourcen zu erreichen. Um von diesen Vorteilen auch auf der Kommunikationsebene zu profitieren, ist es naheliegend das Kommunikationsnetz ebenfalls zu virtualisieren. Eine hohe Verfügbarkeit und Zuverlässigkeit bei einem Fehlerfall, durch Isolation und Fehlertoleranz, sind wesentliche Gründe um eine durchgängig virtualisierte Infrastruktur für CPS zu verwenden. Wie die Ergebnisse in dieser Arbeit zeigen, kann mithilfe der Virtualisierungstechnologie auch von der Sicherheit profitiert werden, um Arbeitsabläufe in cyber-physischen Systemen zu schützen.

Abkürzungsverzeichnis

ACK Acknowledge-Flag

AMD-V AMD Virtualisierung

API Application Programming Interface

BDF Bus Device Function

CPS Cyber-Physische Systeme

CVE Common Vulnerabilities and Exposures

DKOM Dynamic Kernel Object Manipulation

DKSM Direct Kernel Structure Manipulation

DoS Denial-of-Service

IDT Interrupt Descriptor Table

FF Fast Failover

FMA Forensic Memory Analysis

FUSE Filesystem in Userspace

HVM Hardware Virtual Machine

ICMP Internet Control Message Protocol

libxc Xen Control Library

LKM Loadable Kernel Module

MITM Man-In-The-Middle

MTU Master Terminal Unit

OvS Open vSwitch

PCI Peripheral Component Interconnect

PIRQ	PCI Interrupt Request
PLC	Programmable Logic Controller
PV	Paravirtualisierung
REST	Representational State Transfer
RTT	Round-Trip-Time
SDN	Software Defined Networking
STP	Spanning Tree Protocol
SYN	Synchronisations-Flag
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
VM	Virtuelle Maschine
VMM	Virtual Machine Monitor
VMI	Virtual Machine Introspection
VMIFS	VMI Filesystem
VT-X	Intel Virtualisierungstechnologie
XSM	Xen Security Modules
QEMU	Quick Emulator

Abbildungsverzeichnis

1.1 Virtualisierte cyber-physische Systeme. 2

2.1 Beispielszenario eines Wasserverteilungssystems. [ATKF13] 5
2.2 Cyber-Physisches System für das Wasserverteilungssystem. [ATKF13] . . . 6
2.3 TCP Handshake. 8
2.4 DoS SYN Flood Angriff. 8
2.5 Hypervisor Typ 1 und Typ 2. 10
2.6 Xen Architektur. 10
2.7 Linux Taskliste. 13
2.8 SDN Architektur. 16
2.9 Eintrag einer Flowtabelle eines *OpenFlow* Switches. 17

4.1 Traditionelle Xen Architektur. 23
4.2 Xen Device Model Stub Domain. 23
4.3 Xen Network Driver Domain. 24
4.4 Xen Architektur mit der disaggregierten *Domain 0*. 25
4.5 Kombination von LibVMI und FMA Tools. 27
4.6 Virtualisierte Ausführungsplattform mit Sicherheitsmechanismen. 29
4.7 Open vSwitch Architektur. [ERWC14] 31
4.8 sFlow-RT Architektur. 32
4.9 Ausschnitt aus der TCP Header Definition. 34
4.10 Erkennung und Abwehr eines DoS Angriffs durch *sFlow-RT* und *Floodlight*. 35
4.11 Durchgängig virtualisierte Infrastruktur für die Ausführungsplattform und das Kommunikationsnetz mit Sicherheitsmaßnahmen. 36
4.12 Mehrfache Ausführung der *physdevop_get_free_pirq* Funktion des Hypercalls *physdev_op*, um den Hypervisor zum Absturz zu bringen (CVE-2012-3495). [MPA+14] 37
4.13 Linux Taskliste vor und nach einem DKOM Angriff. 39
4.14 Fast-Failover OpenFlow Gruppe an einem Beispiel. 41

5.1 Basis-Netzwerktopologie für die Definition der Szenarien. 44
5.2 Basis-Netzwerktopologie mit aktiviertem STP-Protokoll. 45
5.3 SDN Szenario. 45

5.4 Round Trip Time (RTT) von UDP Paketen im 100ms-Takt. 49
5.5 RTT von UDP Paketen im 10ms-Takt während einer VMI mit Pausieren der VM. 50
5.6 RTT von UDP Paketen im 10ms-Takt während einer VMI mit Volatility. . . 51
5.7 RTT von UDP Paketen im 10ms-Takt während einer aktiven VMI. 52
5.8 DoS Angriff ohne Sicherheitsmechanismen. 53
5.9 DoS Angriff mit Sicherheitsmechanismen. 54
5.10 Fehlertoleranz durch sekundären SDN Controller, beim Ausfall des primären. 54

Tabellenverzeichnis

2.1 Übersicht der Virtualisierungstechniken. 12

5.1 Round Trip Time (RTT) von UDP Paketen im 100ms-Takt während eines *DoS*-Angriffs. 49

5.2 Durschnitt-, Best- und Worst-Case Durchläufe aus 5 Iterationen des Dhrystone Benchmarks in *loop per second (lps)*. 52

Quellcodeverzeichnis

2.1 Typischer Ablauf einer *introspection* mithilfe von LibVMI. 15

4.1 Abbildung von fuse_operations auf benutzerdefinierte Methoden. [vmi17] . 27
4.2 Beispiel Quellcode-Ausschnitt einer aktiven *Virtual Machine Introspection*. . 28
4.3 Definition von Flows in JavaScript für die Erkennung von DoS-Attacken in sFlow-RT. 33
4.4 Registrierung des Event Handlers in sFlow-RT. 34
4.5 Flow zum Verwerfen von Paketen in Floodlight. 35

Literaturverzeichnis

[ANYG15] Ahmad, Ijaz ; Namal, Suneth ; Ylianttila, Mika ; Gurtov, Andrei: Security in software defined networks: A survey. In: *IEEE Communications Surveys & Tutorials* 17 (2015), Nr. 4, S. 2317–2346

[ATKF13] Almalawi, Abdulmohsen ; Tari, Zahir ; Khalil, Ibrahim ; Fahad, Adil: SCADAVT-A framework for SCADA security testbed based on virtualization technology. In: *Local Computer Networks (LCN), 2013 IEEE 38th Conference on* IEEE, 2013, S. 639–646

[BDF+03] Barham, Paul ; Dragovic, Boris ; Fraser, Keir ; Hand, Steven ; Harris, Tim ; Ho, Alex ; Neugebauer, Rolf ; Pratt, Ian ; Warfield, Andrew: Xen and the art of virtualization. In: *ACM SIGOPS operating systems review* Bd. 37 ACM, 2003, S. 164–177

[bdf14] *BDF Notation.* `https://wiki.xenproject.org/wiki/Bus:Device.Function_(BDF)_Notation`, 2014

[Bis02] Bishop, Matthew A.: *The art and science of computer security.* Addison-Wesley Longman Publishing Co., Inc., 2002

[BJW+10] Bahram, Sina ; Jiang, Xuxian ; Wang, Zhi ; Grace, Mike ; Li, Jinku ; Srinivasan, Deepa ; Rhee, Junghwan ; Xu, Dongyan: Dksm: Subverting virtual machine introspection for fun and profit. In: *Reliable Distributed Systems, 2010 29th IEEE Symposium on* IEEE, 2010, S. 82–91

[byt17] *UnixBench.* `https://github.com/kdlucas/byte-unixbench`, 2017

[CAS08] Cárdenas, Alvaro A. ; Amin, Saurabh ; Sastry, Shankar: Research Challenges for the Security of Control Systems. In: *HotSec*, 2008

[CAS+09] Cardenas, Alvaro ; Amin, Saurabh ; Sinopoli, Bruno ; Giani, Annarita ; Perrig, Adrian ; Sastry, Shankar: Challenges for securing cyber physical systems. In: *Workshop on future directions in cyber-physical systems security* Bd. 5, 2009

[CLM+08] Cully, Brendan ; Lefebvre, Geoffrey ; Meyer, Dutch ; Feeley, Mike ; Hutchinson, Norm ; Warfield, Andrew: Remus: High availability via

asynchronous virtual machine replication. In: *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation* San Francisco, 2008, S. 161–174

[CVE12] *CVE-2012-3495.* `https://www.cvedetails.com/cve-details.php?t=1&cve_id=CVE-2012-3495`, 2012

[cve17] *XEN : Security Vulnerabilities.* `https://www.cvedetails.com/vulnerability-list/vendor_id-6276/XEN.html`, 2017

[DGLZ+11] Dolan-Gavitt, Brendan ; Leek, Tim ; Zhivich, Michael ; Giffin, Jonathon ; Lee, Wenke: Virtuoso: Narrowing the semantic gap in virtual machine introspection. In: *Security and Privacy (SP), 2011 IEEE Symposium on* IEEE, 2011, S. 297–312

[DGPL11] Dolan-Gavitt, Brendan ; Payne, Bryan ; Lee, Wenke: Leveraging forensic tools for virtual machine introspection / Georgia Institute of Technology. 2011. – Forschungsbericht

[dia15] *Linux Kernel Rootkit Diamorphine.* `https://github.com/m0nad/Diamorphine`, 2015

[Dun13] Dunlap, George: *Securing your cloud with Xen's advanced security features.* 2013

[ERWC14] Emmerich, Paul ; Raumer, Daniel ; Wohlfart, Florian ; Carle, Georg: Performance characteristics of virtual switching. In: *Cloud Networking (CloudNet), 2014 IEEE 3rd International Conference on* IEEE, 2014, S. 120–125

[FL12] Fu, Yangchun ; Lin, Zhiqiang: Space traveling across vm: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection. In: *Security and Privacy (SP), 2012 IEEE Symposium on* IEEE, 2012, S. 586–600

[flo17] *Floodlight OpenFlow Controller.* `http://www.projectfloodlight.org/floodlight/`, 2017

[ft16] *How to Add Fault Tolerance to the Control Plane.* `https://floodlight.atlassian.net/wiki/spaces/floodlightcontroller/pages/36143107/`, 2016

[fus17] *FUSE.* `https://github.com/libfuse/libfuse`, 2017

[GHB+15] Genge, Béla ; Haller, Piroska ; Beres, Adela ; Sándor, Hunor ; Kiss, István: Using Software-Defined Networking to Mitigate Cyberattacks in Industrial Control Systems. In: *Securing Cyber-Physical Systems* (2015), S. 305

[Gol73] Goldberg, Robert P.: Architectural principles for virtual computer systems / DTIC Document. 1973. – Forschungsbericht

[GR+03] Garfinkel, Tal ; Rosenblum, Mendel u. a.: A Virtual Machine Introspection Based Architecture for Intrusion Detection. In: *Ndss* Bd. 3, 2003, S. 191–206

[hpi06] *Hping Security Tool.* `http://www.hping.org`, 2006

[Int17] Intel: Intel® 64 and IA-32 Architectures Software Developer's Manual. In: *Volume 3 (3A, 3B, 3C & 3D): System Programming Guide* (2017)

[ipe17] *iPerf Network Tool.* `https://iperf.fr`, 2017

[JBZ+15] Jain, Bhushan ; Baig, Mirza B. ; Zhang, Dongli ; Porter, Donald E. ; Sion, Radu: Introspections on the Semantic Gap. In: *IEEE Security & Privacy* 13 (2015), Nr. 2, S. 48–55

[JS15] Jablkowski, Boguslaw ; Spinczyk, Olaf: CPS-Remus: Eine Hochverfügbarkeitslösung für virtualisierte cyber-physische Anwendungen. In: *Betriebssysteme und Echtzeit.* Springer, 2015, S. 39–48

[KRV+15] Kreutz, Diego ; Ramos, Fernando M. ; Verissimo, Paulo E. ; Rothenberg, Christian E. ; Azodolmolky, Siamak ; Uhlig, Steve: Software-defined networking: A comprehensive survey. In: *Proceedings of the IEEE* 103 (2015), Nr. 1, S. 14–76

[Lee08] Lee, Edward A.: Cyber physical systems: Design challenges. In: *Object oriented real-time distributed computing (isorc), 2008 11th ieee international symposium on* IEEE, 2008, S. 363–369

[LHM10] Lantz, Bob ; Heller, Brandon ; McKeown, Nick: A network in a laptop: rapid prototyping for software-defined networks. In: *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks* ACM, 2010, S. 19

[lib17a] *LibVMI.* `http://libvmi.com`, 2017

[lib17b] *LibVMI - APIs.* `http://libvmi.com/api/`, 2017

[Lov10] Love, Robert: *Linux kernel development.* Pearson Education, 2010

[MAB+08] McKeown, Nick ; Anderson, Tom ; Balakrishnan, Hari ; Parulkar, Guru ; Peterson, Larry ; Rexford, Jennifer ; Shenker, Scott ; Turner, Jonathan: OpenFlow: enabling innovation in campus networks. In: *ACM SIGCOMM Computer Communication Review* 38 (2008), Nr. 2, S. 69–74

[min17] *Mininet.* http://mininet.org, 2017

[MMH08] MURRAY, Derek G. ; MILOS, Grzegorz ; HAND, Steven: Improving Xen security through disaggregation. In: *Proceedings of the fourth ACM SIGPLAN/-SIGOPS international conference on Virtual execution environments* ACM, 2008, S. 151–160

[MPA+14] MILENKOSKI, Aleksandar ; PAYNE, Bryan D. ; ANTUNES, Nuno ; VIEIRA, Marco ; KOUNEV, Samuel: Experience report: an analysis of hypercall handler vulnerabilities. In: *Software Reliability Engineering (ISSRE), 2014 IEEE 25th International Symposium on* IEEE, 2014, S. 100–111

[ovs17a] *Open vSwitch.* http://openvswitch.org, 2017

[ovs17b] *Open vSwitch Manpages.* http://openvswitch.org/support/dist-docs/, 2017

[Pay12] PAYNE, Bryan D.: Simplifying virtual machine introspection using libvmi. In: *Sandia report* (2012), S. 43–44

[pci16] *Xen PCI Passthrough.* https://wiki.xenproject.org/wiki/Xen_PCI_Passthrough, 2016

[Pha02] PHAAL, Peter: *sFlow-RT Packet Sampling Basics.* http://www.sflow.org/packetSamplingBasics/, 2002

[PML07] PAYNE, Bryan D. ; MARTIM, DP de A. ; LEE, Wenke: Secure and flexible monitoring of virtual machines. In: *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual* IEEE, 2007, S. 385–397

[PPA+09] PFAFF, Ben ; PETTIT, Justin ; AMIDON, Keith ; CASADO, Martin ; KOPONEN, Teemu ; SHENKER, Scott: Extending Networking into the Virtualization Layer. In: *Hotnets*, 2009

[qem17] *Quick Emulator.* http://www.qemu.org, 2017

[rek17a] *Public Profile Repository for Rekall Memory Forensic.* https://github.com/google/rekall-profiles, 2017

[rek17b] *Rekall check_modules Plugin.* http://www.rekall-forensic.com/docs/Manual/Plugins/Linux/#check_modules, 2017

[rek17c] *Rekall Forensics.* http://www.rekall-forensic.com, 2017

[RR08] RICHARDSON, Leonard ; RUBY, Sam: *RESTful web services.* O'Reilly Media, Inc., 2008

[sfl13] *sFlow Counters vs. Packet Samples.* http://blog.sflow.com/2013/02/measurement-delay-counters-vs-packet.html, 2013

[sfl17] *sFlow-RT.* http://sflow-rt.com, 2017

[SILB15] SUNEJA, Sahil ; ISCI, Canturk ; LARA, Eyal de ; BALA, Vasanth: Exploring vm introspection: Techniques and trade-offs. In: *Acm Sigplan Notices* Bd. 50 ACM, 2015, S. 133–146

[SKK+97] SCHUBA, Christoph L. ; KRSUL, Ivan V. ; KUHN, Markus G. ; SPAFFORD, Eugene H. ; SUNDARAM, Aurobindo ; ZAMBONI, Diego: Analysis of a denial of service attack on TCP. In: *Security and Privacy, 1997. Proceedings., 1997 IEEE Symposium on* IEEE, 1997, S. 208–223

[SM07] SLAY, Jill ; MILLER, Michael: Lessons learned from the maroochy water breach. In: *Critical infrastructure protection* (2007), S. 73–82

[SN05] SMITH, James E. ; NAIR, Ravi: The architecture of virtual machines. In: *Computer* 38 (2005), Nr. 5, S. 32–38

[tcp17] *Tcpdump.* http://www.tcpdump.org, 2017

[TPSJ12] TEIXEIRA, André ; PÉREZ, Daniel ; SANDBERG, Henrik ; JOHANSSON, Karl H.: Attack models and scenarios for networked control systems. In: *Proceedings of the 1st international conference on High Confidence Networked Systems* ACM, 2012, S. 55–64

[VAVAK14] VAN ADRICHEM, Niels L. ; VAN ASTEN, Benjamin J. ; KUIPERS, Fernando A.: Fast recovery in software-defined networks. In: *Software Defined Networks (EWSDN), 2014 Third European Workshop on* IEEE, 2014, S. 61–66

[vmi17] *VMI Filesystem.* https://github.com/libvmi/libvmi/tree/master/tools/vmifs, 2017

[vol12] *Volatility Plugins.* https://volatility-labs.blogspot.de/2012/09/movp-15-kbeast-rootkit-detecting-hidden.html, 2012

[vol17a] *Volatility Forensics.* http://www.volatilityfoundation.org, 2017

[vol17b] *Volatility linux_check_syscall Plugin.* https://github.com/volatilityfoundation/volatility/wiki/Linux-Command-Reference#linux_check_syscall, 2017

[xen08] *Device Model Stub Domains.* http://blog.xen.org/index.php/2008/08/28/xen-33-feature-stub-domains/, 2008

[xen16] *Xen Summit 2016: XSM-FLASK.* `http://schd.ws/hosted_files/xensummit2016/d0/XSM-Flask.pdf`, 2016

[xen17a] *Device Model Stub Domains.* `https://wiki.xenproject.org/wiki/Device_Model_Stub_Domains`, 2017

[xen17b] *Driver Domain.* `https://wiki.xenproject.org/wiki/Driver_Domain`, 2017

[xen17c] *Xen Project Hypervisor.* `https://wiki.xenproject.org/wiki/Xen_Project_Software_Overview`, 2017

[xin14] *Linux Kernel Rootkit Xingyiquan.* `https://packetstormsecurity.com/files/128945/Xingyiquan-Linux-2.6.x-3.x-Rootkit.html`, 2014

[xl17] *Xen xl Toolstack Manpages.* `http://xenbits.xen.org/docs/unstable/man/xl.1.html`, 2017

[XLXJ12] Xiong, Haiquan ; Liu, Zhiyong ; Xu, Weizhi ; Jiao, Shuai: Libvmi: A library for bridging the semantic gap between guest os and vmm. In: *Computer and Information Technology (CIT), 2012 IEEE 12th International Conference on* IEEE, 2012, S. 549–556

[xsm17] *Xen Security Modules.* `https://wiki.xenproject.org/wiki/Xen_Security_Modules_:_XSM-FLASK`, 2017

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet sowie Zitate kenntlich gemacht habe.

Dortmund, den 12. September 2017

Majuran Rajakanthan