Entwicklung eines Build-Systems für Android Virtual Devices für Sicherheitsforscher

Bachelorarbeit

ausgearbeitet von

Siarhei Sheludzko
3092139

zur Erlangung des akademischen Grades
Bachelor of Science (B.Sc.)

vorgelegt an der
Rheinische Friedrich-Wilhelms-Universität Bonn
Institut für Informatik IV
Arbeitsgruppe für IT-Sicherheit

im Studiengang
Informatik (B.Sc.)

Erstprüfer:	Prof.Dr. Peter Martini Universität Bonn
Zweitprüfer:	Prof.Dr. Michael Meier Universität Bonn
Betreuer:	Daniel Baier, M.Sc. Fraunhofer FKIE

Bonn, 27. Februar 2021

Inhaltsverzeichnis

1 Einleitung

In den letzten Jahren stieg die Verwendung der Smartphones. Die Grafik 1 zeigt, dass im Jahr 2020 weltweit 3.5 Milliarden Benutzer regelmäßig Smartphones verwenden [47]. Das populärste Betriebssystem von mobilen Geräten ist Android OS. Der Anteil von Smartphones mit Android im Jahre 2020 liegt bei 75% [46]. Android basiert auf dem Linux-Kernel und ist open-source. Aufgrund von der Popularität und offenem Quellcode ist Android ein attraktives Ziel für die Entwickler von Malware. Die Grafik 2 zeigt, wie die Anzahl von detektierten Android-Malware Packages im Zeitraum vom zweiten Quartal 2019 bis zum zweiten Quartal 2020 steigt [40]. Um die Android-Benutzer zu schützen und das Betriebssystem sicherer zu machen, müssen die Malware analysiert werden. Die Analyse kann statisch und dynamisch erfolgen.

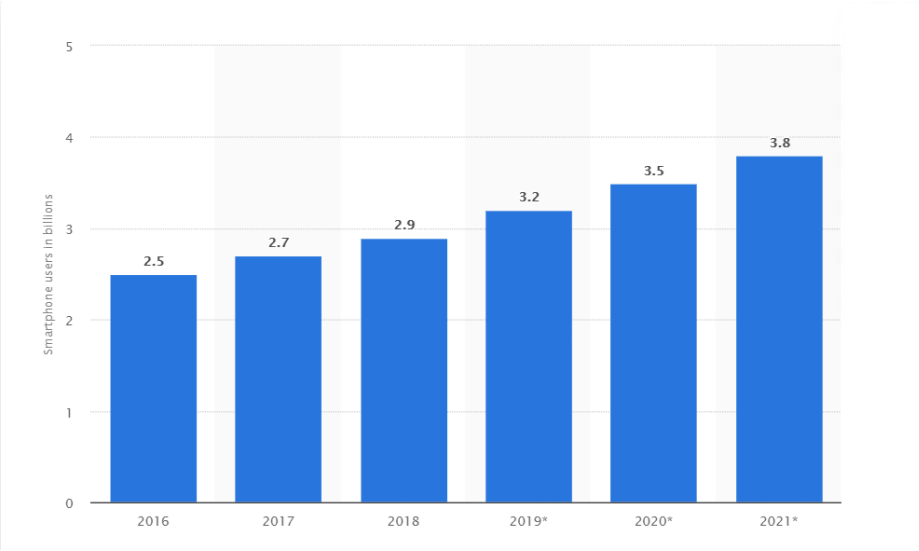

Abbildung 1: *Anzahl von Smartphone Benutzern weltweit von 2016 bis 2021. [47]*

Bei der statischen Analyse werden die Quelldateien von der Applikation analysiert. So kann man alle Prozess-Pfade erkennen, ohne tatsächlich Programme auszuführen. Im Gegensatz zur statischen Analyse wird die Applikation bei der dynamischen Analyse während seiner Laufzeit beobachtet. Hierdurch wird nur ein dedizierter Programmpfad analysiert, jedoch mit konkreten Werten, welche meist im Rahmen einer statischen Analyse fehlen.[22] Für die dynamische Analyse ist es wichtig, dass der Forscher jegliche Prozesse auf dem Android OS verfolgen kann. Dafür braucht man die erweiterten Berechtigungen, welche dem Standard-Benutzer im Android Betriebssystem fehlen. Der Root-Benutzer hingegen besitzt diese Berechtigungen. Aus diesem Grund müssen Sicherheitsforscher der Android-Plattform sich zunächst Root-Berechtigungen für

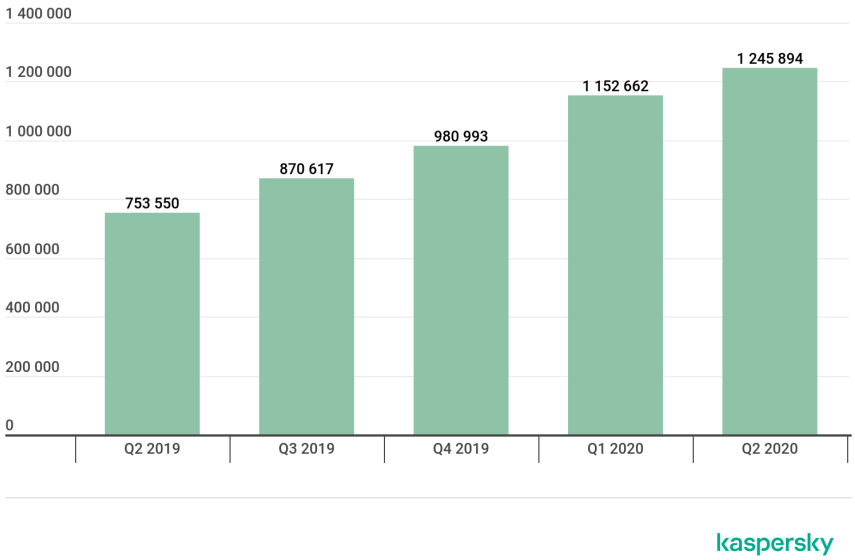

Abbildung 2: *Anzahl der erkannten schädlichen Installationspakete, Q2 2019 – Q2 2020. [40]*

ihr Forschungsgerät verschaffen. Dieser Prozess wird meist als "Rooting"bezeichnet [30]. Eine Malware (falls das keinen Rootkit ist) funktioniert wie die normale Applikationen im Userspace und für den Zugriff auf die System-Ressourcen verwendet die Systemcalls[21]. Das ist eine mögliche Stelle für das Beobachten der Aktivitäten vom Malware Prozess. Zusätzlich kann man sowohl die Labrary-Funktionen als auch die Funktionen von Malware hooken und so die Funktionalität von Schadsoftware überwachen oder auch verändern [32]. Falls die Malware die Rootkit-Eigenschaften hat (das Prozess wird im Kernelspace ausgeführt), ist es wichtig, dass man die Prozesse im Kernelspace tracen kann. Deswegen sollte für den vollumfänglichen dynamischen Analyse der Analyst die Möglichkeit haben, die Prozesse sowohl im Userspace als auch im Kernelspace zu überwachen und ggfs. zu verändern. Neben des Monitorings im User- und Kernelspace benötigen Analysten darüber hinaus die Möglichkeit das Android-System ihren Bedürfnissen anzupassen. Hierzu existieren verschiedene Möglichkeiten. Das Nachladen von eigenen Kernelmodulen bietet hierbei die größte Flexibilität [48]. Das Forschungsgerät soll die folgende Eigenschaften haben:

- Aktuelle Version von Android
- Root-Rechte
- Werkzeug für das Tracing inkl. Hooking im Userspace
- Werkzeug für das Tracing im Kernelspace
- Erweiterbarkeit des Kernels

Leider existiert momentan kein Werkzeug, welches ein angepasstes Android-System mit diesen Eigenschaften für eine dynamische Analyse erstellt. Sicherheitsforscher müssen deshalb erst aufwendig ihr Android-Image anpassen, bevor sie eine Analyse starten können.

Die Bachelorarbeit hat die folgende Struktur:
Kapitel 2 beschreibt die notwendigen Grundlagen die für das weitere Verständnis dieser Arbeit

erforderlich sind.
Kapitel 3 verschafft einen Überblick zum gegenwärtigen Stand der Forschung hinsichtlich der Entwicklung eines Build-Systems für Android Virtual Devices.
Kapitel 4 macht den Vergleich von möglichen Varianten der Implementierung einzelner Aspekte des Frameworks und beschreibt die wichtigsten Schritte der Implementierung vom Framework.
Kapitel 5 evaluiert den Ansatz von dieser Bachelor Arbeit anhand des implementierten Frameworks aus dem Kapitel 4.
Kapitel 6 fasst den Ansatz der Arbeit zusammen.
Kapitel 7 diskutiert einige Vorschläge für zukünftige Arbeiten und weitere Entwicklung des Build-Systems für Android Virtual Devices für Sicherheitsforscher.

2 Grundlagen

In diesem Kapitel wird auf die für das Framework und für das Android-System Image relevanten Konzepte eingegangen.

2.1 Android Open Source Project

Android ist ein Betriebssystem für mobile Endgeräte. Android ist auch ein Open Source-Projekt, welches als Open-Source Projekt von Google geleitet wird. [3]
Um das Funktionieren vom Android-System und die damit verbundene Implementierung von dem Framework zu verstehen, werden in diesem Unterkapitel die Grundlagen von Android dargestellt.

2.1.1 Android-Architektur

Das Android-System Architektur besteht aus den folgenden Komponenten [9]:

- Application framework
- Binder IPC
- System services
- Hardware abstraction layer (HAL)
- Linux kernel

Die Abbildung 3 stellt die Architektur von Android-System dar.

Application framework

Das "Application Framework" bietet APIs für die Applikationen. Das wird am häufigsten von Applkationentwicklern verwendet. Jede Android-Applikation läuft in einer isolierten Umgebung (Dalvik Virtual Machine), so kann man das Applikationen-Niveau und das System-Niveau ganz klar voneinander trennen.[57]

Binder IPC

Binder IPC ist ein "inter-process communication"-Mechanismus, welcher die Kommunikation zwischen den Prozessen von Application Framework und Android system services ermöglicht. Unterschiedliche Android system services verwenden ebenfalls miteinander die Binder IPC für die Kommunikation.[57]

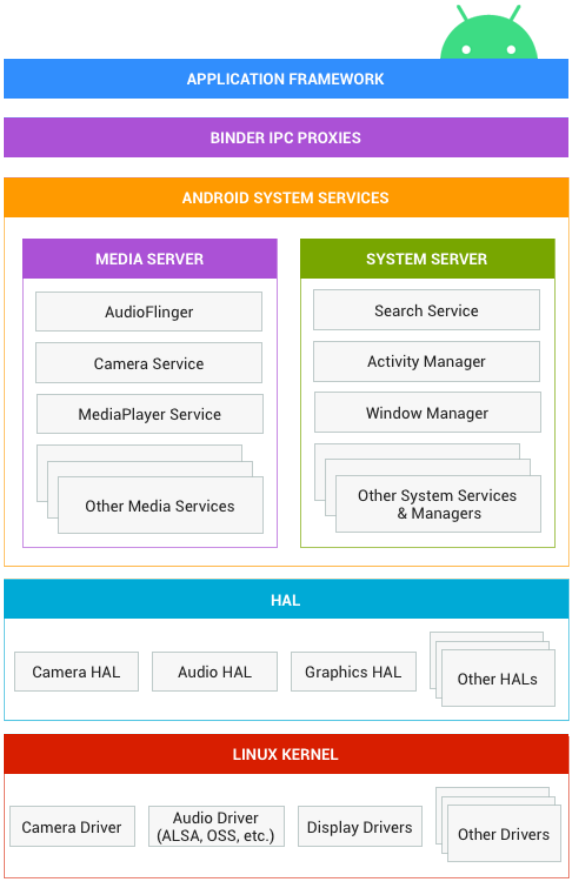

Abbildung 3: *Android-Systemarchitektur [9]*

Android system services

Android system services Layer umfasst 2 Gruppen von Services: ***system*** (zB. Window Manager) und ***media*** (Services, die im Spielen und Aufnehmen vom Media beteiligt sind). Parallel dazu funktionieren noch die native Services, deren Funktionsweise den Linux-Daemons ähnelt. Die Linux-Daemons sind die Dienstprogramme, die die verschiedenen Dienste verwalten, die das Betriebssystem für eine ordnungsgemäße Funktion benötigt. Der Begriff "Daemon" ist Referenz für Maxwells Dämon, ein imaginäres Wesen, das ständig im Hintergrund arbeitet und Moleküle sortiert[51].[57]

Hardware Abstraction Layer (HAL)

HAL definiert den standardisierten Interface für Hardware und isoliert das "high-level" Android-System von der "low-level" Hardware-Driver Implementierung. [57]

Linux Kernel

Um den weiteren Verlauf dieser Arbeit nachzuvollziehen, ist es wichtig zu verstehen, dass Android eine wenig veränderte Version von Linux Kernel verwendet. Die Mehrheit von den Veränderungen wurde als Device-Drivers realisiert, deswegen wurde der Linux Core Kernel-Code kaum geändert.

Die größte Veränderung von Core Kernel-Code war “wakelocks“ (powermanagment Solution von Android). Die Kommunikation zwischen Kernel-Space und User-Space in Android erfolgt genau so wie unter Linux.[57]

2.1.2 Android Debug Bridge

Android Software Development Kit (SDK) umfasst ein zahlreiches Set von Entwicklungs-Tools. “Platform Tools“ ist ein Subset von Android SDK und besteht aus Command-Line Tools. Ein Werkzeug aus den “Platform Tools“ wird oft im Rahmen dieser Arbeit benutzt - die Android Debug Bridge (adb oder ADB). [54] Diese ermöglicht die Kommunikation zwischen dem Entwicklungscomputer und dem Android Device. Dieses Device (auch Endgerät) kann sowohl ein physisches Endgerät sein als auch ein emuliertes. Mit Hilfe von ADB ist es möglich, die Applikationen unter Android zu installieren und debuggen. ADB basiert auf einer Client-Server Architektur und besteht aus folgenden Teilen:

- Client
- Daemon (adbd)
- Server (adb)

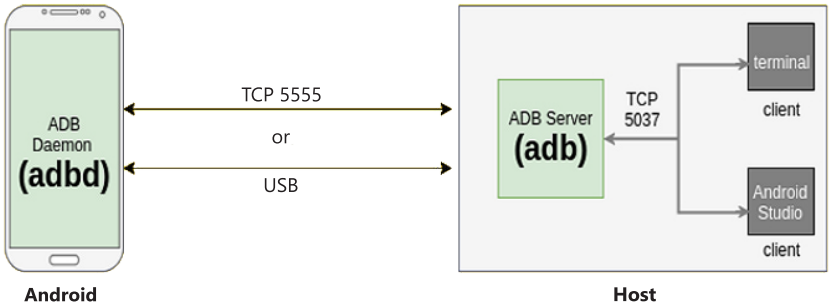

Abbildung 4: *Schematische Darstellung von Android Debug Bridge [2]*

Der Client läuft auf dem Entwicklungscomputer und sendet die Befehle zum Server. Der Daemon ruft die Befehle unter dem Android und läuft im Hintergrund auf dem Gerät/Emulator. Der Server realisiert die Kommunikation zwischen Client und Daemon und ist ebenfalls im Hintergrund auf dem Entwicklungscomputer aktiv. Das Funktionieren von Android Debug Bridge ist schematisch auf der Abbildung 4 gezeigt. Beim Starten vom ADB Client wird überprüft, ob der Server schon läuft. Wenn nicht, wird der Server gestartet. Die Client-Server Kommunikation erfolgt durch den lokalen TCP-Port 5037. Dann der Server stellt die Verbindung mit dem ADB Daemon. Dafür können die ungeradezahlige TCP-Ports von 5555 bis 5585 verwendet werden.[7]

2.2 Containervirtualisierung

Containervirtualisierung ist eine Methode, mit welcher man die Instanzen von einer Anwendung in eine isolierte Umgebung einfügen und ausführen kann. Diese Umgebung wird als Container bezeichnet. Container sind "lightweight", weil sie das Laden eines Hypervisor nicht benötigen,

sondern direkt im Kernel vom Host-System ausgeführt werden. Eine der populärsten Lösungen für die Containervirtualisierung ist Docker.[23]

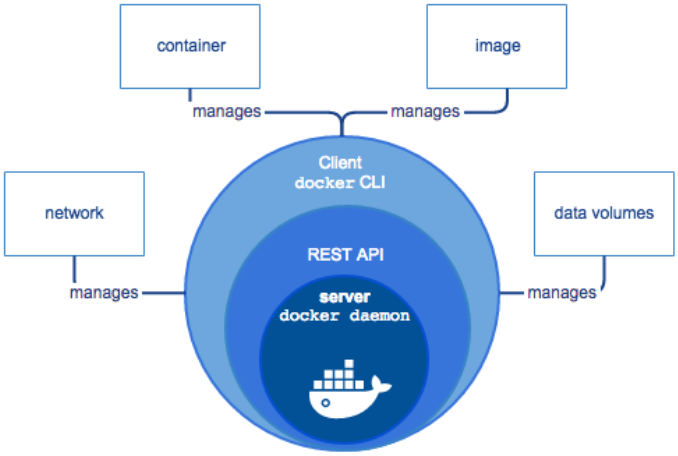

Abbildung 5: *Docker Engine Architektur [23]*

2.2.1 Docker

Docker ist eine open-source Plattform zur Isolierung von Anwendungen durch die Containervirtualisierung. Die Docker Engine ist eine Client-Server Applikation. Die Abbildung 5 zeigt die Struktur von der Docker Engine. Der Server ist ein Daemon und funktioniert wie ein Hintergrundprozess. Der Server erstellt und verwaltet die Docker-Objekte wie Images, Container, Netzwerke und Volumes. Die REST-API ist die Schittstelle, über welche die Programme mit dem Server kommunizieren können. Das CLI (command line interface) verwendet Docker REST-API für die Interaktion mit Docker Server mit Hilfe von Skripts oder direkt mit CLI Commands.[23]

2.3 Rooting unter Android

Wie in dem Abschnitt 3 geschrieben wird, basiert Android auf Linux Kernel. Deswegen hat Android das ähnliche Benutzer-Berechtigungsystem. Root (oder Superuser) ist der Benutzer vom Kernel. Darum kann Root alle Kernel-Berechtigungsprüfungen umgehen. Root-Berechtigung zu erhalten bedeutet, dass ein Prozess mit dem `UID 0` ausgeführt wird. Diese werden durch das Ausführen einer Binärdatei erworben, die entweder: `set-user-ID-root` (SUID) Bit [55], oder "File capabilities" (`setgid, setuid+ep`) hat[10]. Die Befehle su und sudo verwenden die erste Variante. Die auführende Binärdatei sendet einen `setuid` Systemcall, sodass der Prozessstatus auf "privilegiert" erhöht wird. Die Rooting-Methode ist von der Android-Version abhängig. Um die Root-Rechte auf dem Gerät bis Android 4.3 zu bekommen, muss man einfach eine `set-user-ID-root su` Binärdatei ausführen. Eine solche Methode wird bei den folgenden Tools verwendt: CF-Auto-Root, Framaroot, Towelroot und KingRoot[31, 45].
Ab Version 4.3 hat Android auf "File capabilities" gewechselt, weil das ein sicherer Mechanismus ist.

Nur die System-Daemons und Services können die "File capabilities" verwenden, weil der Code von Applikationen von zygote mit dem Prozesssteuerungsattribut `NO_NEW_PRIVS` ausgeführt wird. Darum kann man keine `set-user-ID` oder "File capabilities" anwenden. Zusätzlich ab Android 4.3 kann man UID nur wechseln, falls der Prozess `SETUID/SETGID` Capabilities in dem "Bounding Set" hat. Bei allen Android Applikationen werden diese Capabilities mit dem Prozess-Steuerungsparameter gedropt. Ab Android 8 verwendet man noch den Filter von Systemcalls (seccomp filter)[43], der einige Systemcalls blockieren kann. Daraus ergibt sich, dass die su-Binärdatei nicht mehr für das Rooten verwendet werden kann. Deswegen hat das Rooting zur Methode mit `su daemon` gewechselt. Der Daemon wird während der Durchführung des Boots gestartet und dann verarbeitet dieser alle Superuser-Requests von den Applikationen, wenn sie eine spezielle Binärdatei ausführen. Für das Starten vom SU Daemon verwendet man das Skript `install-recovery.sh`. Das Skript ist durch den `flash_recovery` Init-Service ausgeführt. Zusätzlich wird in Android 5 der `flash_recovery` Init-Service zu dem "restricted SELinux context" hinzugefügt. Deshalb darf sogar der Superuser nur eine begrenzte Menge von Aufgaben ausführen. Die einzige Möglichkeit diese Begrenzung zu umgehen, ist das Patchen von der SELinux Policy. [8, 45] SuperSU und Magisk implementieren die oben geschriebene Rooting-Methode. Man kann diese ab Android 6.0 anwenden[31].

2.3.1 SuperSU

SuperSU ist ein Dienstprogramm zum Rooten von Android-Geräten und zum Verwalten von Superuser-Berechtigungen. Mit Hilfe von SuperSU man kann Android rooten und Superuser-Anforderungen von den Apps verwalten. SuperSU ändert die System-Dateien und fügt die neuen Dateien in die System-Partition von Android-System hinzu.[1]

2.3.2 Magisk

Magisk ist eine freie und open source all-in-one Root-Lösung für Android Smartphones. Magisk verwendet eine "systemless" Root-Methode, weil das Android-Rooting durchgeführt wird, ohne die System-Partition und Boot-Partition zu modifizieren. Dank der "systemless" Mounting-Methode von Magisk können die Benutzer den Root-Status des Geräts für bestimmte Dienste ausblenden. [38]

2.4 Tracing und Hooking Frameworks im User-Space

In dieser Sektion beschreibt man die existierenden Instrumente für die dynamische Analyse von Applikationen im Userspace.

2.4.1 Uprobes

Uprobes (User-Probes) ist eine dynamische Methode zum Einfügen von"probe points" in User-Space Applicationen. Uprobes bearbeitet Userspace breakpoints im Kernel und funktioniert analog zu kprobes. Die genauere Beschreibung von kprobes befindet sich in dem Abschnitt 2.5.2. Uprobes können nur für Tracing im Userspace verwendet werden.[50]

2.4.2 Ptrace

`ptrace()` ist ein Systemcall, mit welchem ein Prozess (“tracer“) die Ausführung eines anderen Prozesses (“tracee“) beobachtet und gesteuert werden kann. Tracer kann auch den Speicher und Register von Tracee untersuchen und modifizieren. Es gibt 2 Möglichkeiten für Tracen mit ptrace(): mit Hilfe von `fork()` und mit `PTRACE_ATTACH`. Beim `fork()` wird der Vaterprozess dann zum Tracer und der Kindprozess zum Tracee. Beim `PTRACE_ATTACH` wird der Zielprozess(Tracee) als das Kind von dem Aufrufprozess(Tracer) behandelt. Der Ablauf von dem Tracing mit `ptrace()` ist in dem Schema 6 vorgestellt.[41]

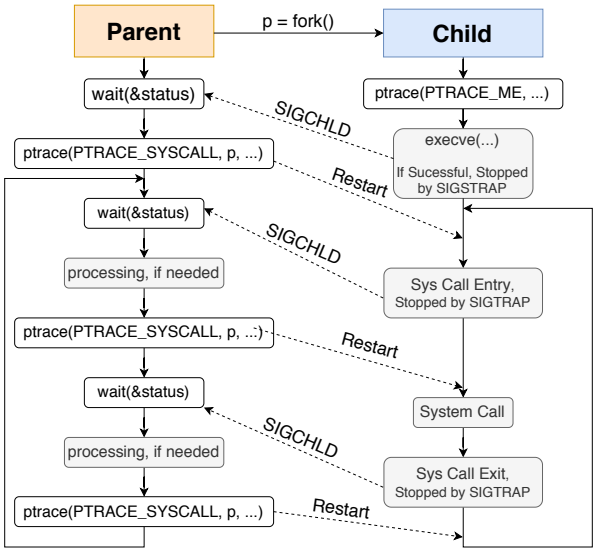

Abbildung 6: *Vater- und Kind-Prozess Interaktion bei ptrace-Tracing. Nach [35]*

2.4.3 Frida

Frida ist ein dynamisches Instrumentations Toolkit. Es kann unter Windows, macOS, QNX, iOS und Android verwendet werden. Mit Hilfe von Frida kann man Javascript Code in Prozessen von nativen Applikationen injizieren. Unter Linux und Android das Hooking ist über den `ptrace`-Systemcall realisiert.[42]

Frida Architektur

Frida basiert auf einer Client-Server Architektur, bei der das Scriptobjekt der Client ist und der Frida-Agent der Server. Der Client und der Server kommunizieren miteinander über einen JSON-basierten Kommunikationskanal. Der Client sendet einen Payload zum Server und bietet ein Benutzer-Interface. Der Server führt die Payload aus. [42]

Frida Agent-Injektion

Wie in der Abbildung 7 gezeigt, injiziert der Debugger den Frida-Agenten in den Debuggee. Auf dem ersten Schritt (Nummer 1 auf der Abbildung) allokiert der Frida-Server den Speicher im Zielprozess und lädt dann den Bootstrapper-Code. Dann startet der Frida-Server den Bootstrapper-Thread

(Nummer 2) im Zielprozess. Der Bootstrapper lädt den Frida-Agenten (Nummer 3). Der Frida-Agent lädt eine Javascript Engine (Google V8 oder Ducktape) und erstellt einen bidirektionalen Kommunikationskanal (Nummer 4). Mit Hilfe von diesem Kanal kann der Debugger die Payload zum Frida-Agenten senden und der Frida-Agent führt die Payload aus (Nummer 5).[42]

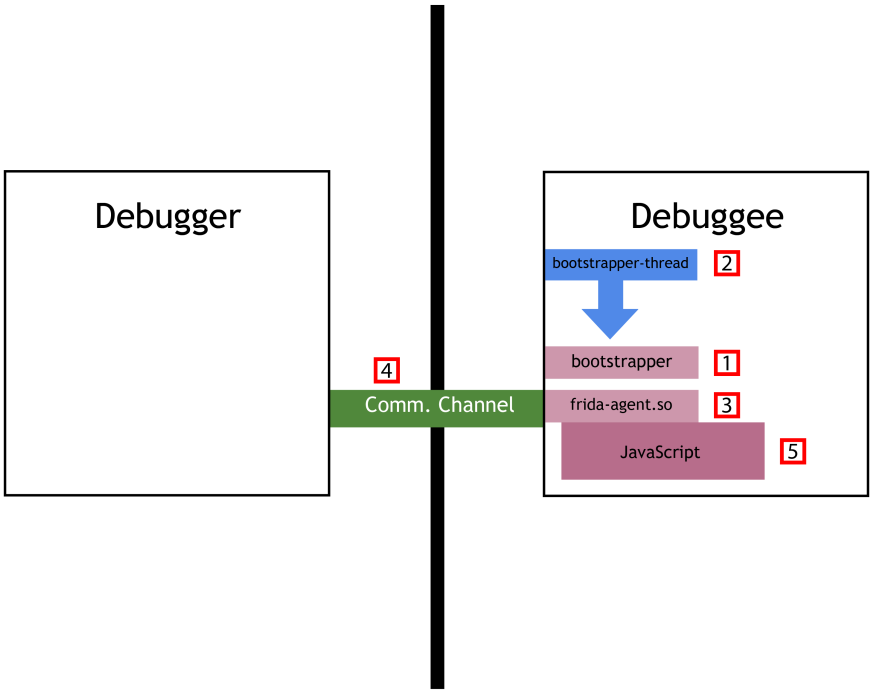

Abbildung 7: *Frida Agent-Injektion. Nach [42]*

2.5 Tracing Frameworks im Kernel-Space

Tracing ist eine spezielle Verwendung der Protokollierung von Informationen über den Ausführungsfluss eines Programms. Tracing kann für viele Zwecke verwendet werden, z.B. für Debugging, Leistungstests, Überwachung und Erkennung von Bedrohungen und die entsprechende Reaktion. Es gibt unterschiedliche Instrumente für Tracing im Kernelspace unter Linux. Im Folgenden werden die verbreitetsten Kernel-Tracing Verfahren vorgestellt. [49]

2.5.1 Tracepoints

Der Tracepoint befindet sich im Code und bietet einen Hook zum Aufrufen einer Funktion (Probe), die man zur Laufzeit bereitstellen kann. Tracepoints werden anstelle von Hooks im Code definiert. Deswegen ist die Lösung mit Tracepoints schneller als andere Technologien. Aber Tracepoints haben auch einen Nachteil, da sie statisch sind und nicht dynamisch hinzugefügt werden können. [49]

2.5.2 Kprobes

Kprobes ist ein mächtigeres Linux Tool, welches Kernel-Probes implementiert. Mit Hilfe von kprobes kann man einen Händler erstellen, welcher vor oder nach dem Aufrufen einer bestimmten Kernel-Funktion ausgeführt sein kann. Kprobes hängt die Probes direkt zu den sys_*-Funktionen oder anderen Funktionen im Kernel an. Zusätzlich kann man eigene Probes in Kernel mit Hilfe von Kernel-Modulen hochladen. Eine Tracing-Lösung mit Kprobes ist dynamisch und nur etwas langsamer als Tracepoints. [49]

2.5.3 Extended Berkley Packet Filter(eBPF)

Der extended Berkley Packet Filter (eBPF) ist ein Framework für Tracing unter Linux-Systemen. Das Tool existiert in Linux seit der Kernel 4.9 Version [34]. eBPF ist als "in-kernel" virtuelle Maschine realisiert, auf der vom Benutzer bereitgestellte eBPF-Programme ausgeführt werden können. Diese Programme sind mit Sonden oder Events im Kernel verbunden und können Statistiken sammeln, die zum Tracing oder Debuggen verwendet werden.[5] eBPF Applikationen sind ereignisgesteuert. Diese werden ausgeführt, wenn der Kernel oder eine Applikation einen bestimmten Hook-Punkt überschreitet. Zu den pre-definierten Hook-Punkten gehören syscalls, enter\exit-Funktionen, Kernel-Ereignisse und Netzwerk-Ereignisse. Es ist möglich, die Kernel-Probe (kprobe) oder User-Probe (uprobe) zu erstellen, falls die pre-definierten Hooks für die bestimmte Stelle nicht existieren. So kann eine eBPF-Application fast überall angehängt werden. Die Lösung mit eBPF ist langsamer als die Lösung mit Tracepoints. Laut dem Patchset für eBPF ist das Anwenden von kprobes + bpf 20% langsamer als Tracepoint [13]. Die letzten Versionen von eBPF unterstützen Just-in-Time Kompilieren (JIT). Das beschleunigt die Ausführung und macht den Unterschied mit Tracepoint noch kleiner. Die schematische Darstellung des Funktionierens von eBPF befindet sich auf dem Diagramm 8. Auf dem ersten Schritt (Nummer 1 auf dem Diagramm) generiert man BPF Bytecode. Dann lädt man den BPF Bytecode in Kernel-Runtime und wählt die Event Targets zum Attachen (Nummer 2 und 3). Danach schreibt die BPF Applikation die Ergebnisse in eBPF Map oder Perf Buffer (Nummer 4). Im Userspace sind diese Daten in "perf-event data" oder "statistics" erreichbar. [24, 34, 49]

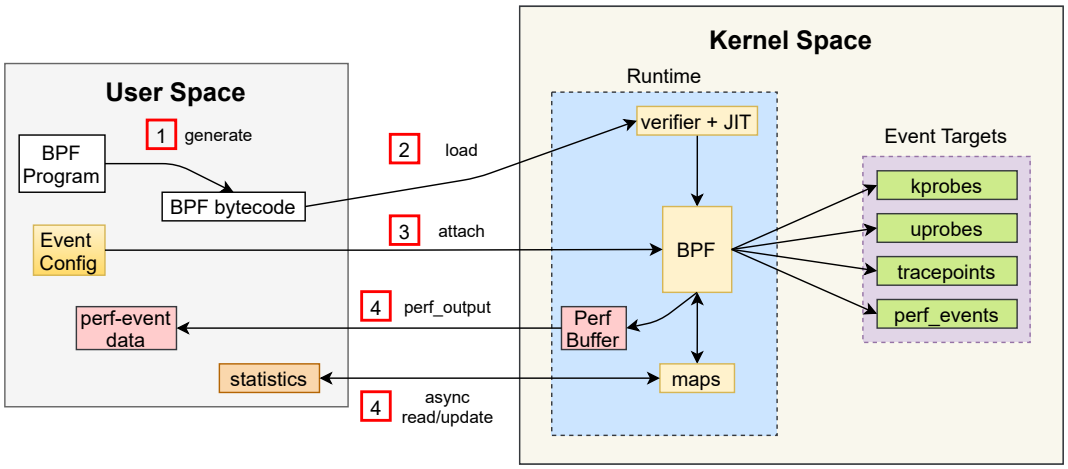

Abbildung 8: *Schematische Darstellung von Tracing mit eBPF [34]*

Android eBPF Lader

Das Android-Build-System unterstützt das Kompilieren von C-Programmen zu eBPF-Maschinenanweisungen. Android enthält eBPF-Lader und Libraries, die eBPF-Programme während des Bootens laden. Beim Booten von Android werden alle eBPF-Programme aus dem Pfad `/system/etc/bpf/` mit Hilfe von `bpf(2)` Systemcall geladen.[5]

2.6 adeb

androdeb (adeb) [27] liefert eine leistungsstarke Linux-Shell-Umgebung auf einem Android-Gerät. In der adeb Umgebung kann man wichtige Linux-Tools für das Tracing, Kompilieren und Editieren ausführen. adeb unterstützt alle Befehle, die auf einem modernen Linux-System verfügbar sind.[27]

2.7 BPF Compiler Collection

BPF Compiler Collection (BCC) ist ein Toolkit zum Erstellen effizienter Tracing- und Manipulations-Programme für Kernelspace. BCC enthält auch mehrere nützliche Tools und Beispiele. Es verwendet extended Berkley Packet Filter (eBPF) und für die korrekte Ausführung braucht Linux Kernel Version 4.1 oder höher. BCC erleichtert das Schreiben von BPF-Programmen. Für das Kernel-Instrumentieren verwendet man C-Sprache und für Front-End - Python und Lua. BCC eignet sich für unterschiedliche Aufgaben, einschließlich Performance-Analyse und Netzwerkverkehrskontrolle.[12]

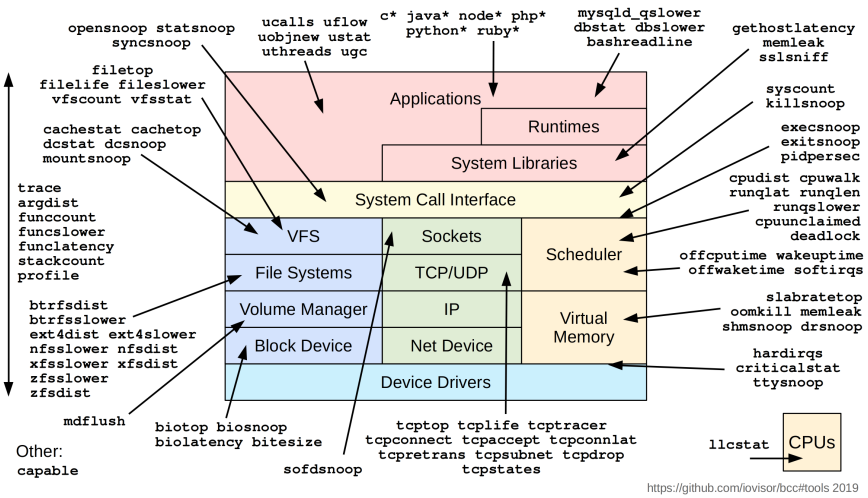

Abbildung 9: *Linux bcc/BPF Tracing Tools [12]*

Die Abbildung 9 listet die BCC-Tools auf und ordnet die BCC-Tools den entsprechenden Teilen von Linux Kernel zu.

2.8 BCC auf Android

Es existieren 2 Möglichkeiten für das Verwenden von BPF Compiler Collection (BCC) unter Android: die erste Lösung mit einem Daemon und einem Proxy für alle eBPF Requests und die zweite Lösung mit BCC in androdeb (adeb). Beide Lösungen wurden von Joel Fernandes entwickelt. In den folgenden Untersektionen 2.8.1 und 2.8.2 wird beschrieben, wie beide Lösungen funktionieren.[25]

2.8.1 Berkeley Packet Filter Daemon

Berkeley Packet Filter Daemon (BPFd) ermöglicht die Cross-Entwicklung und Ausführung der Tools über Maschinen- und Architektur-Grenzen. Die Abbildung 10 veranschaulicht das Funktionieren von BPFd. Kernel Sources, die für die BCC Tools notwendig sind, befinden sich auf dem Entwicklungscomputer (Local Development Machine), wobei eBPF-Code auf einen Remotecomputer (Remote Target) geladen und ausgeführt wird.[14]

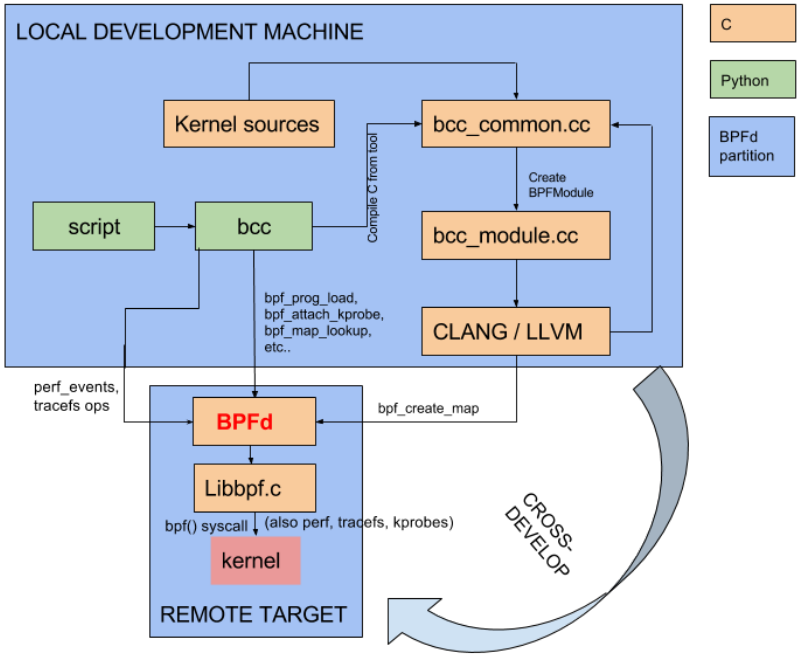

Abbildung 10: *Schematische Darstellung der Arbeit mit Berkeley Packet Filter Daemon [14]*

2.8.2 BCC im androdeb

androdeb (adeb) ist das Hauptinstrument für die Ausführung von BCC unter Android. adeb unterstützt die Vorbereitung von Android-Gerät, das Klonen der BCC Repository und das Bauen von BCC auf dem Gerät. Für das Verwenden von BCC in adeb braucht man Linux Kernel Version 4.9 oder höher. Linux Kernel soll zusätzlich mit bestimmten Parametern gebaut werden. Diese Parameter sind in dem folgenden Listing dargestellt.[27]

```
1 CONFIG_KPROBES=y
2 CONFIG_KPROBE_EVENT=y
3 CONFIG_BPF_SYSCALL=y
```

```
4 CONFIG_IKHEADERS=m
```

Listing 2.1: *Kernel Konfiguration für BCC*

Die Vorteile der Anwendung von androdeb anstelle mit BPFd:

- androdeb hat trace-cmd, perf und alle open source Debian Tools
- Das Ausführen von BCC Tools, die viele Daten ausgeben können (z.B. bcc/trace)
- Für Geräte mit arm64-Architektur dauert das Setup nur 5 Minuten, weil `rootfs` aus dem Internet heruntergeladen wird

Die Nachteile der Anwendung von androdeb:

- Die installierte `rootfs` braucht etwa 300Mb Speicher auf dem Gerät
- Man braucht "adb root"

[25]

2.9 Kernel-Module

In der Sektion 2.1 hat man schon geschrieben, dass Android auf dem Linux Betriebssystem basiert und Linux-Kernel nur minimal verändert wurde. Deswegen kann man die Linux-Kernel-Module in Android nachladen. Kernel Module bezeichnet einen Binärcode, welcher zur Laufzeit bei Bedarf in den Kernel geladen und entladen werden kann, um die Funktionalität des Kernels ohne Neustart zu erweitern. Das Modul kann als "build-in" oder "loadable" konfiguriert werden. Um ein Modul dynamisch zu laden oder entladen, muss man das Laden und Entladen von Modulen im Kernel aktivieren.[53] Die Abbildung 11 stellt diesen Prozess schematisch dar.

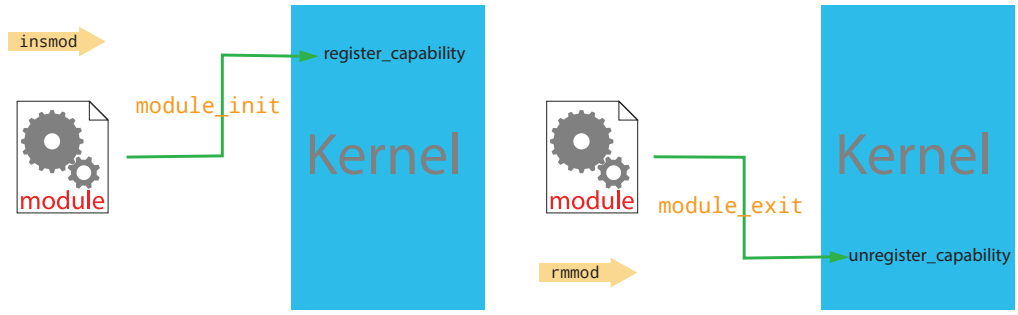

Abbildung 11: *Schematische Darstellung von Laden und Entladen eines Kernel-Moduls. Nach [36]*

3 Verwandte Arbeiten

In Rahmen der Arbeit konnten wenig verwandte Arbeiten in dem Gebiet "Entwicklung eines Build-Systems für Android Virtual Devices für Sicherheitsforscher" identifiziert werden, da sich recht wenige generell mit dem Thema befassen (oder nicht öffentlich publizieren). Corellium ist die einzige Firma, welche in diesem Gebiet ein fertiges Produkt hat. Das Produkt heißt CORSEC und ist ein cloud-basierter Emulator, welcher auf einem ARM64 Server läuft. CORSEC wurde im November 2020 veröffentlicht [16]. Das Projekt ist nicht ´´open source´´ und man kann in der Zeit des Schreibens dieser Arbeit keine öffentliche Publikationen finden. Aber man kann Teilaspekte der Arbeit nach verwandten Arbeiten weiter recherchieren.
Im vorliegenden Kapitel werden zunächst die Arbeiten auf den Gebieten "Rooting des Android-Emulators", "Dynamische Tracing Frameworks", "eBPF-Programme unter Android" und "Laden und Entladen von Kernel-Modulen unter Android" dargestellt.

3.1 Rooting Methoden

In den nachfolgenden Quellen werden die Magisk- und SuperSU-Rooting Methoden vorgestellt. Man vergleicht auch die mögliche Variante der Umgehung von Root-Erkennung.

3.1.1 Android 10: Emulation of Magisk/SuperSU on an AOSP AVD

In dem Artikel "Android 10: Emulation of Magisk/SuperSU on an AOSP AVD" [4] beschreibt Tony Tannous den Prozess der Installation von SuperSU und Magisk auf Android AOSP 10 Virtual Device (AVD) mit dem Emulator aus Android SDK Tools. Man fasst die Constraints und die Begrenzungen von Android Emulatoren im Vergleich mit den echten Android-Geräten zusammen. Zum Beispiel die Emulatoren unterstützen `recovery/fastboot/bootloader` nicht, deswegen unterscheiden sich die Installation von SuperSU und Magisk auf dem Smartphone und im Emulator. Seit Version 4.3 hat Android das Security Enhenced Linux (SELinux) System. Um das Rooting vom Emulator zu realisieren, soll man diesen mit dem SELinux-Parameter `permissive` starten. Für beide SuperSU und Magisk soll man die Daemon-Prozesse nach dem Reboot manuell neu starten. In dem Artikel wird beschrieben, dass nach der Installation von Magisk die Magisk Module nicht korrekt funktionieren.

3.1.2 Comparison of Different Android Root-Detection Bypass Tools

Dieser Artikel [15] fasst die Root-Erkennung-Umgehung Techniken zusammen. Der Autor vergleicht 6 "Root-Detection Bypass Tools" miteinander. Das sind RootCloak, UnRootBeer, Fridan-

tiroot, Objection und Magisk mit MagiskHide. Als Root-Erkennungs Applikation benutzt man "RootBeer Sample", welche ein Open Source Root-Erkennung Library "RootBeer" verwendet. "RootBeer" Library implementiert die folgenden Prüfungen: RootManagement Apps, BusyBox Binary, SU Binary, SU Exists, Dangerous Props, RootNative und SELinux Flag Enabled. Die einzige Lösung, welche alle diese Prüfungen umgeht, ist Magisk mit MagiskHide.[15]

3.2 Dynamische Tracing Frameworks

Android OS basiert auf Linux Kernel, deswegen kann man viele Linux Tracing Tools unter Android auch verwenden. Im Folgenden wird ein Artikel vorgestellt, welcher die Linux Kernel-Tracing Werkzeuge zusammenfasst.

3.2.1 Tracing Linux: Fast, Compatible, Complete

In dem Artikel "Tracing Linux: Fast, Compatible, Complete" [49] versucht Christopher Arges alle Tracing Tools unter Linux zusammenzufassen. Man beschreibt wie diese Tools funktionieren und welche Vor- und Nachteile diese Tools haben. Am Ende werden alle Tools für Kernelspace entsprechend der Kriterien "Complete", "Fast" und "Compatible" verglichen. Dann wählt man mit Hilfe dieser Kriterien das entsprechende Tool für das Entwickeln von "Autonomous Detection & Response platform".

3.3 eBPF-Programme unter Android

BPF Compiler Collection (BCC) ist ein mächtigeres Werkzeug für die Analyse von Linux Kernel. Default Android hat BCC nicht, aber es gibt die Möglichkeiten, BCC unter Android zu verwenden. In der folgenden Arbeit fasst man die Aspekte der Anwendung von BCC unter Android zusammen.

3.3.1 eBPF super powers on ARM64 and Android. Powerful Linux Tracing for Android

Die "eBPF super powers on ARM64 and Android. Powerful Linux Tracing for Android"[25] von Joel Fernandes (der Entwickler von androdeb) beschreibt die Verwendung von eBPF unter Android und zeigt die Vorteile von BPF Compiler Collection (BCC) auf. Der Autor stellt 2 Varianten der Anwendung von BCC dar: die erste Lösung mit einem Daemon und einem Proxy für alle eBPF Requests und die zweite Lösung mit BCC in androdeb. Am Ende zeigt man in Beispielen die Verwendung einiger BCC Tools.

3.4 Laden und Entladen von Kernel-Modulen unter Android

Mit Hilfe von ladbaren Kernel-Modulen ist es möglich das Android Betriebssystem den Bedürfnissen des Researchers dynamisch anzupassen. In dem nachfolgenden Artikel stellt man den Prozess der Erstellung von ladbaren Kernel-Modulen unter Android vor.

3.4.1 How-to Write a Kernel Module for Android

Abdullah Yousafzaii beschreibt in seinem Artikel "How-to Write a Kernel Module for Android" [33] den Prozess des Kompilierens vom Kernel und Kernel-Modulen für CyanogenMod 10. Das ist ein Open Source Betriebssystem für mobile Geräte, welches auf AOSP basiert und sehr ähnliche Strukturen aufweist [20]. Der Artikel behandelt die Unterschiede zwischen dem Erstellen von Kernel-Modulen in Linux OS und Android. Der Autor stellt ein "hello world"-Beispiel von dem Kernel-Modul dar.[33] Es ist wichtig zu beachten, dass seit Android 8 Source Code von Android-Kernel aus AOSP Repository in ein unabhängiges Repository verschoben wurde. Darum beschreibt der Artikel nicht alle Schritte vom Kompilieren des Kernels und Kernel-Modulen für Android 10.

4 Framework

Im Rahmen dieser Bachelorarbeit wird ein Framework für das automatisierte Erstellen von Android Virtual Device für Sicherheitsforscher entwickelt. Das Framework erstellt das spezialisierte Android Virtual Device schrittweise. Einige Schritte des Frameworks können durch unterschiedliche Ansätze implementiert werden. Diese werden im Abschnitt 4.1 vorgestellt und die Auswahl von einem Ansatz für jeden Schritt diskutiert. Abschließend wird in Abschnitt 4.3 die Implementierung der ausgewählten Konfiguration des Frameworks vorgestellt.

4.1 Auswahl von Werkzeugen für das Framework

In diesem Unterkapitel vergleicht man die Werkzeuge für das Rooting des AVDs, Tracing inkl. Hooking im Userspace und Tracing im Kernelspace. Man macht den Vergleich, um für jede Position eine optimale Variante zu finden, welche den Zielen der Arbeit am bestens entspricht. Zusätzlich werden noch die Version von Linux-Kernel und die Programmiersprache für das Framework ausgewählt.

4.1.1 Rooting Tool

Im Rahmen dieser Bachelorarbeit wird mit Hilfe vom Build-System ein Android-System Image auf Basis von Android 10 automatisch erstellt. Um ein geeignetes Mittel für das Rooting zu finden, muss man alle möglichen Rooting-Ansätze miteinander vergleichen. Wie in dem Abschnitt 2.3 dargestellt wird, sind die populärsten Lösungen für das Rooting von Android ab Version 6.0 SuperSU und Magisk. Um die optimale Variante zu finden, wird entschieden Magisk und SuperSU anhand der Kriterien “Mounting Methode“, “Module“ und “Open Source“ zu vergleichen. Die Mounting Methode ist sehr wichtig, weil beim System Mounting die Android Applikationen die Möglichkeit für die Erkennung von Root-Status des Geräts haben. Im Gegensatz dazu kann man bei “Systemless“ Mounting den Root-Status des Geräts für bestimmte Applikationen ausblenden. Mit Hilfe von Extramodulen lässt sich das System für bestimmten Aufgaben besser anpassen. Das “Open Source“ Kriterium ist relevant, weil man die Applikation auf schädliche Funktionalitäten, wie zum Beispiel Backdoors, überprüfen kann. Die nachkommende Tabelle 1 fasst die Kriterien zusammen. [31]

Wie in dem Abschnitt 3.1 beschrieben wird, funktionieren die Magisk Module unter dem Android Emulator nicht korrekt. Deswegen hat Magisk laut diesem Kriterium keinen Vorteil im Vergleich mit SuperSU. Jedoch erkennt man anhand der Tabelle 1, dass nur Magisk in den Spalten “Mounting Methode“ und “Open Source“ alle notwendigen Eigenschaften besitzt. Zusätzlich, wie in dem

Tabelle 1: *Vergleich von SuperSU und Magisk Rooting Methoden [56]*

	Mounting Methode	Module	Open Source
SuperSU	System	⊖	⊖
Magisk	"Systemless"	⊖	⊕

⊕ - Rooting Technik hat die entsprechende Eigenschaft
⊖ - Rooting Technik hat die entsprechende Eigenschaft nicht

Abschnitt 3.1.2 geschrieben wird, hat Magisk das eingebaute Werkzeug für die Umgehung von Root-Erkennung. Deswegen ist Magisk die optimalste Rooting-Lösung.[38]

4.1.2 Tracing Tool im Userspace

Das weit verbreitete Tool Frida erfüllt den Zweck, dass man automatisiert das Research-Tool erstellen kann, mit welchem das Instrumentieren von Prozessen im User-Space erfolgt. Viele Sicherheit-Research Instrumente basieren auf Frida (z.B. Dwarf, r2frida). Darum ist Frida als Tracing inkl. Hooking Tool im Userspace ausgewählt.

4.1.3 Tracing Tool im Kernelspace

Im Abschnitt 2.5 sind die Instrumente für das Tracing im Kernelspace beschrieben. Das sind Tracepoint, Kprobes und Extended Berkley Packet Filter(eBPF). Um die optimalste Technologie für das Tracing im Kernelspace unter Android Emulator zu finden, werden sie anhand der Kriterien "Vollständigkeit" und "Geschwindigkeit" verglichen. Das Tracing ermittelt die relevanten Informationen aus System-Calls und anderen Ereignissen. Je mehr unterschiedliche Varianten von solchen Ereignissen in einem Tool enthalten sind, desto mehr Varianten existieren von der Anwendung des Tools. Deswegen ist die Vollständigkeit des Tracings wichtig. Der Parameter Geschwindigkeit ist wichtig, weil der Emulator begrenzte Rechenressourcen hat und die langsamere Lösung die Leistung von dem System signifikant verringern kann. Die Tabelle 2 stellt die Eigenschaften von den Kernel-Tracing Werkzeugen hinsichtlich der oben genannten Kriterien gegenüber.

Tabelle 2: *Vergleich von Kernel Tracing Tools[49]*

	Vollständigkeit	Geschwindigkeit
Tracepoints	syscalls + static points	extrem schnell
Kprobes	syscalls + dynamic functions	wenig langsamer als tracepoints
eBPF	tracepoints + kprobes	20% langsamer als kprobes

Aus der Tabelle 2 folgt, dass eBPF alle mögliche Varianten von Tracing umfasst und nur 20% langsamer als kprobes ist. Deshalb wird eBPF als Tracing Tool im Kernelspace ausgewählt.

eBPF unter Android

Wie in dem Abschnitt 2.5.3 geschrieben wird, unterstützt Android das Kompilieren und Laden von eBPF-Programmen. Nur beim Booten werden diese Programme aus dem Pfad `/system/etc/bpf/` mit Hilfe von `bpf(2)` Systemcall geladen. Diese Lösung ist nicht sonderlich flexibel, da der Emulator

für jede neue eBPF-Applikation neu gestartet werden muss. Die bessere Alternative ist die BPF Compiler Collection (BCC). Die normale Version von Android hat keine Kernel-Headers, clang und Python, und es fehlen einige andere Abhängigkeiten, deswegen unterstützt das normale Android BCC nicht. Wie bereits im Abschnitt 2.8 geschrieben, existieren zwei Lösungen für das Verwenden von BCC mit Android: die erste Lösung mit einem Daemon (BPFd) und einem Proxy für alle eBPF Requests und die zweite Lösung mit BCC in androdeb (adeb). Diese zwei Lösungen werden anhand der Kriterien "Vollständigkeit" und "aktive Entwicklung" verglichen. "Vollständigkeit" ist wichtig, weil nicht alle BCC-Tools in beiden Lösungen funktionieren. Das Kriterium "aktive Entwicklung" ist wichtig, weil beide Projekte offene Issues auf github.com haben. Einige Instrumente wie trace-cmd, perf und bcc/trace funktionieren nur in androdeb (Vergleich Abschnitt 2.8.2) . Deswegen ist nur die Lösung mit androdeb vollständig. Wenn man die Geschichte der Commits von beiden Projekten anschaut [12, 14], dann sieht man, dass das letzte Commits für androdeb im July 2020 war und für BPFd im Oktober 2018. In dem Abschnitt2.8 beschreibt man die Nachteile der Anwendung von der Lösung mit androdeb. Aber die sind in Rahmen dieser Bachelor Arbeit irrelevant. Der standard Android Emulator hat 2Gb von Speicher frei, deswegen sind die 300Mb von `rootfs` nicht kritisch. In Emulatoren kann man "adb root" problemlos verwenden. Die Entscheidungsmatrix 3 fasst diese Ergebnisse zusammen.

Tabelle 3: *Anwenden von BCC unter Android[25]*

	Vollständigkeit	Aktive Entwicklung
BPFd	⊖	⊖
androdeb	⊕	⊕

⊕ - Methode hat die entsprechende Eigenschaften
⊖ - Methode hat die entsprechende Eigenschaften nicht

Aus der Tabelle 3 sieht man, dass nur die Lösung mit androdeb alle notwendige Eigenschaften hat. Darum wird man BCC Lösung mit androdeb in Rahmen dieser Bachelor Arbeit verwenden.

4.1.4 Custom Kernel für Android

In dem Abschnitt 4.1.3 wird entschieden, dass für das Anwenden von eBPF im Rahmen dieser Arbeit BCC in androdeb verwendet wird. Wie im Abschnitt 2.8.2 erläutert wird, muss das Kernel zuerst konfiguriert werden (Listing 2.1), und die Kernel Headers sind notwendig. Zusätzlich braucht auch das Verwenden von ladbaren Kernel Modulen eine bestimmte Konfiguration von Kernel (genauere Beschreibung ist in der Sektion 4.3.2). Das default prekompilierte Kernel von AOSP entspricht diesen Forderungen nicht. Um das Kernel richtig zu konfigurieren und die Kernel Headers da zu haben, muss man das Kernel aus dem Source Code kompilieren. Seit Android 8 befindet sich der Quellcode von Android Kernel in einem eigenen Repository. Wie in dem Abschnitt 2.8.2 geschrieben ist, braucht man für das vollständige funktionieren von BCC Kernel Version 4.9 oder höher. Aus diesem Grund wird das Branch `q-goldfish-android-goldfish-4.14-dev` von Android Kernel Source mit Kernel Version 4.14 ausgewählt.[6]

4.1.5 Programmiersprache des Frameworks

Da die Bachelorarbeit eine zeitliche Grenzen hat, ist es unmöglich in dieser Zeit ein fertiges Produkt zu erstellen. Deswegen ist das Framework ein Prototyp vom Build-System für Android Virtual

Devices für Sicherheitsforscher. Die Geschwindigkeit der Ausführung des Framework-Codes wird dadurch unwichtig. Wichtig ist jedoch die Geschwindigkeit vom "Editieren&Testen-Schleife" und das Vorhandensein zahlreicher Bibliotheken. Darum wird für die Entwicklung des Frameworks Python 3.8 genutzt. Zusätzlich verwendet man in einigen Teilen vom Framework Shell-Skripts.

4.2 Workflow vom Framework

Hier wird dargestellt, in welcher Reihenfolge das Framework die einzelnen Schritte vom Erstellen des AVDs für Sicherheitsforscher macht.

Der Workflow des Erstellens von AVD für Sicherheitsforscher ist auf der Abbildung 12 schematisch dargestellt und umfasst die folgenden Schritte (auf dem Diagramm sind diese als Ziffern in roten Quadraten markiert erkennbar):

1. Erzeugen von Android System Image
2. Erzeugen vom Custom Kernel und Kernel Modul
3. Starten von AVD
4. Rooting des AVDs
5. Installation von Frida-Server
6. Installation von androdeb
7. Installation von BCC innerhalb des androdebs

Die genauere Beschreibung der einzelnen Schritte wird in dem Abschnitt 4.3 gegeben.

4.3 Implementierung

In dieser Sektion werden alle Schritte von dem Bauen eines Android Virtual Devices für Sicherheitsforsher beleuchtet.

4.3.1 Erzeugen von Android System Image

Dieser Schritt ist auf der Abbildung 12 mit der Nummer 1 markiert und fängt mit dem Initialisieren eines Repository von AOSP im Ordner `~/repos/aospan`. Für Android 10 wird das Branch `android-10.0.0_r41` ausgewählt, weil das die maximale Anzahl von den unterstützten Geräten hat. Nach dem Initialisieren von Repository macht man eine Synchronisation. Um die Synchronisation maximal effektiv zu machen, findet der Skript die Anzahl von Threads des CPUs und übergibt die als Argument in `repo sync`. Die Entwickler von AOSP empfehlen Ubuntu 18.04 (Bionic Beaver) oder die letzte Version von macOS als Host-Betriebssystem für das Kompilieren und Bauen von AOSP. Um das Framework portabel und unabhängig vom Host-Betriebssystem (in der "Linux-Welt") zu machen, wird entschieden, das Kompilieren und Bauen von System Image innerhalb eines Docker-Containers zu realisieren. Der Container enthält Ubuntu 18.04 und alle notwendigen Abhängigkeiten. Als Ergebnis von diesem Schritt bekommt man ein Android System Image. Das System Image umfasst ein default-Kernel, aber wie in dem Abschnitt 4.1.4 geschrieben ist, ent-

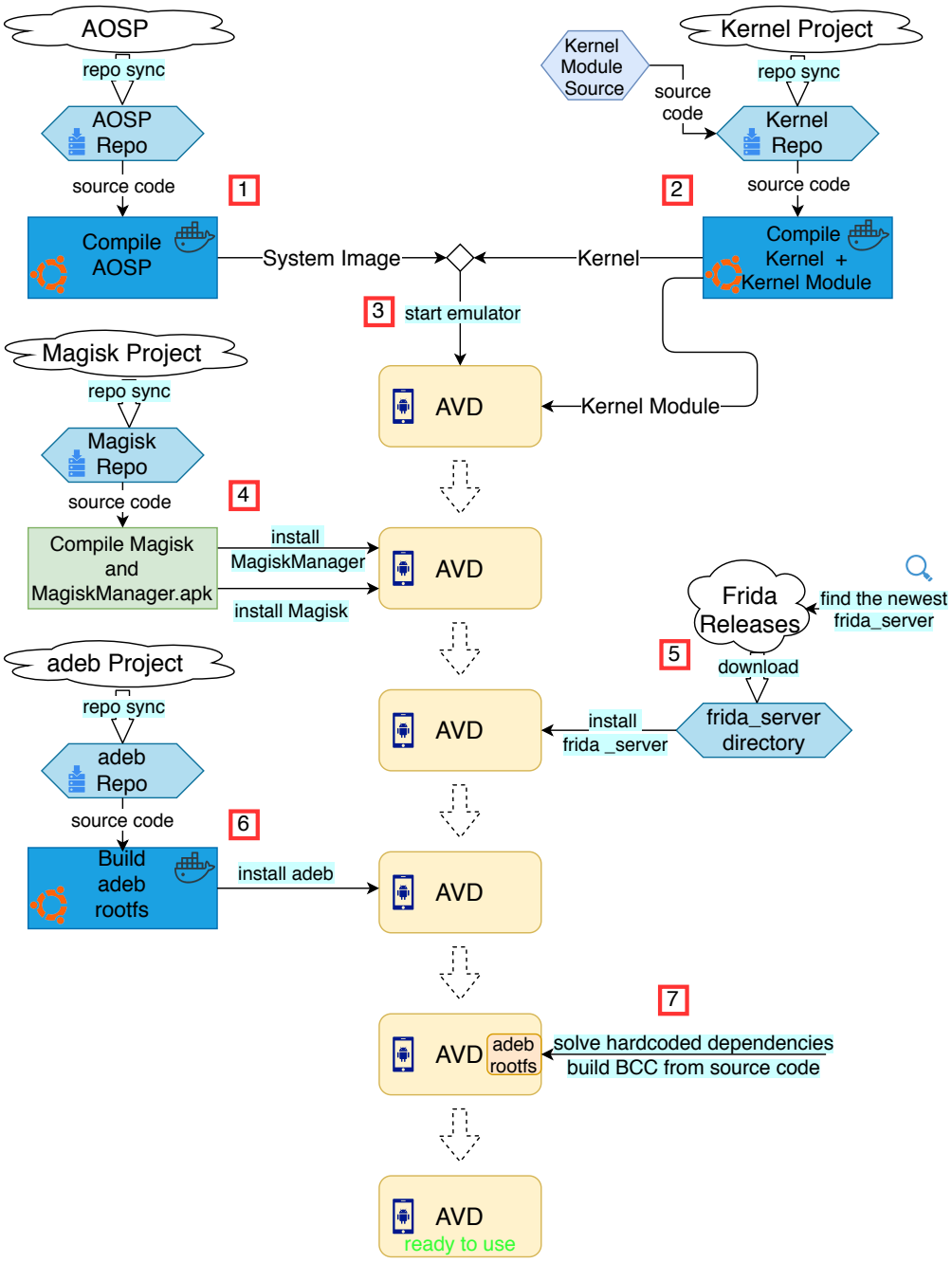

Abbildung 12: *Flussdiagramm vom Framework*

spricht das den Zielen dieser Arbeit nicht. Deswegen wird im nächsten Schritt ein Custom-Kernel gebaut.

4.3.2 Erzeugen vom Custom Kernel und Kernel Modul

Analog zum Kompilieren von AOSP initialisiert und synchronisiert das Framework ein Repository von Android Kernel Source auf dem Pfad `~/repos/kernel_source`. Der Schritt ist auf dem Diagramm 12 mit der Nummer 2 bezeichnet. Die Anwendung von Kernel-Modulen ist stark vom

Use Case abhängig. Deswegen wird für die Demonstration von Laden und Entladen von Kernel-Modulen im Rahmen dieser Bachelorarbeit der einfache "hello world" Kernel-Modul erstellt. Im Listing 4.1 ist der Quellcode von diesem Kernel-Modul dargestellt. Das "helloworld"-Kernel-Modul hat einen typischen Aufbau: Die Module müssen meistens auf GPL License eingesetzt werden (Zeile 3), eine Beschreibung der Funktionalität vom Modul ist auch verpflichtend (Zeile 4). Man braucht die Definition von Funktionen `module_init()` und `module_exit()` entsprechend für das Laden und Entladen vom Modul (Zeilen 15 und 16). Beim Laden vom Modul mit dem Befehl `insmod` wird die Funktion `hello_init(void)` aufgerufen. So wird eine Nachricht in Kernel Message Buffer geschrieben. Das Entladen vom Modul mit dem Befehl `rmmod` funktioniert analog.

```
1  #include <linux/init.h>
2  #include <linux/module.h>
3  MODULE_LICENSE("GPL");
4  MODULE_DESCRIPTION("Hello World Kernel Module for Android");
5
6  static int hello_init(void)
7  {
8      printk("Hello Kernel World....\n");
9    return 0;
10 }
11 static void hello_exit(void)
12 {
13     printk("Goodbye Kernel World\n");
14 }
15 module_init(hello_init);
16 module_exit(hello_exit);
```

Listing 4.1: *Quellcode vom Kernel Modul*

Für die Registrierung vom neuen Kernel-Modul im `KBuild` System sollen die `Kconfig` und `Makefile` Dateien im Ordner mit Modul Quellcode erzeugt werden. Der Inhalt von diesen Dateien ist entsprechend im Listing 4.2 und Listing 4.3 dargestellt.

```
1  config HELLO_MOD
2    tristate "Hello World Module"
3    default m
4    depends on MODULES
5    help
6      This module will write Hello Kernel World to the Kernel Message Buffer
```

Listing 4.2: *Kconfig file für KBuild System*

```
1  ifneq ($(KERNELRELEASE),)
2  # call from kernel build system
3  obj-m := hello.o
4  endif
```

Listing 4.3: *Makefile*

Das Framework erstellt den neuen Ordner "helloworld" im Directory `kernel/drivers`. Der Ordner enthält die früher beschriebenen Dateien `hello.c`, `Kconfig` und `Makefile`. Auf dem Diagramm 12 sind die Quelldateien von "helloworld"- Modul als "Kernel Module Source" dargestellt. Zusätzlich registriert das Framework die neuen `Kconfig` und `Makefile` in `drivers/Kconfig` und

drivers/Makefile. Die Erweiterungen von diesen Dateien sind in Listing 4.2 und Listing 4.3 dargestellt.

```
1 source "drivers/helloworld/Kconfig"
```

Listing 4.4: *Erweiterung von drivers/Kconfig*

```
1 source "obj-$(CONFIG_HELLO_MOD) += helloworld/"
```

Listing 4.5: *Erweiterung von drivers/Makefile*

In dem Abschnitt 2.9 ist schon geschrieben, dass für dynamisches Laden und Entladen von Kernel-Modulen ohne Neustart das Kernel entsprechend konfiguriert werden muss. Deswegen überprüft das Framework den Inhalt von der defconfig -Datei und erweitert die Konfiguration mit den Zeilen, die in dem Listing 4.6 dargestellt sind. Die Option CONFIG_MODULES aktiviert die Unterstützung für ladbare Module, CONFIG_MODULE_FORCE_LOAD lässt das Laden von Modulen ohne Versionsinformationen zu. Der Parameter CONFIG_MODULE_UNLOAD aktiviert die Möglichkeit für das Entladen von Kernel-Modulen. Mit der Option CONFIG_MODULE_FORCE_UNLOAD kann Kernel das Modul entfernen, ohne darauf zu warten, dass jemand das nicht mehr braucht. Zusätzlich, für das Funktionieren von BCC auf Android, erweitert das Framework die defconfig -Datei mit den Zeilen aus dem Listing 2.1 in dem Abschnitt 2.8.2. [37]

```
1 CONFIG_MODULES=y
2 CONFIG_MODULE_FORCE_LOAD=y
3 CONFIG_MODULE_UNLOAD=y
4 CONFIG_MODULE_FORCE_UNLOAD=y
```

Listing 4.6: *Kernel Konfiguration für Loadable Kernel Modulen*

Für die Portabilität und Plattform-Unabhängigkeit ist das Kompilieren vom Kernel und Kernel Modulen (Analog zum Kompilieren von AOSP) innerhalb von Docker Container mit Ubuntu 18.04 mit allen notwendigen Abhängigkeiten realisiert. Ubuntu 18.04 wird ausgewählt, weil diese Version bei AOSP empfohlen ist. Als Ergebnis von diesem Schritt bekommt man das fertige Android Kernel und "helloworld" Kernel Module. Das Kernel hat die Headers und ist für das Laden von Kernel Modulen bereit.

4.3.3 Starten von AVD

Dieser Schritt ist auf der Abbildung 12 mit der Nummer 3 gekennzeichnet. Für das Verwenden vom neu kompilierten Kernel im Emulator, soll der Emulator mit dem Parameter -kernel ~/repos/kernel_source/goldfish/arch/x86/boot/bzImage gestartet werden. Das kompilierte "helloworld"-Kernel-Modul befindet sich auf dem Pfad ~/repos/kernel_source/goldfish/drivers/helloworld/hello.ko. Nach dem Starten vom Emulator sendet das Framework diese Datei mit Hilfe von adb (Android Debug Bridge) in den Ordner des Emulators /data/. Für das Laden von diesem Modul muss der Benutzer den Befehl adb shell "insmod /data/hello.ko" eingeben.

4.3.4 Rooting des AVDs

Framework Skript für Rooting mit Magisk clont Magisk Repository in den Ordner `~/repos/Magisk`, dann wird eine Konfigurationsdatei generiert, wo man die Versionen von Magisk und Magisk Manager definiert. In dieser Arbeit sind 21.0 und 8.0.2 entsprechend. In der Zeit des Implementierens von diesem Schritt das waren die aktuellste Versionen, die noch im beta-Test waren [39]. Der Grund für diesen Auswahl ist, dass die älteste Versionen nicht vollständig funktionsfähig waren. Im nächsten Schritt ruft das Framework "build-script" vom Magisk auf, der baut Magisk und Magisk Manager .apk-Datei und generiert dann das shell-Skript für den Emulator. Anschließend kopiert das Framework das shell-Skript in den Emulator und ruft diesen an. Shell-Skript installiert Magisk auf Emulator. Am Ende installiert man Magisk Manager. Um die Root-Erkennung zu umgehen, kann der Benutzer den MagiskManager umbenennen und Magisk Hide entsprechen einstellen. Die schematische Darstellung von Rooting des AVDs ist auf dem Diagramm 12 mit der Nummer 4 markiert.

4.3.5 Installation von Frida-Server

Die Installation von Frida-Server ist mit der Nummer 5 auf dem Schema 12 bezeichnet. Skript findet die aktuellste Version von Frida-Server für die "ausgewählte Architektur" auf der offiziellen Github-Seite von dem Frida Projekt, lädt den Archiv mit dem Frida-Server herunter und extrahiert den aus dem Archiv. Dann installiert das Framework den Frida-Server mit Hilfe von adb (Android Debug Bridge) auf Emulator und startet den Frida-Server da. Beim neu Starten von Emulator muss man Frida-Server wider starten. Dafür hat das Framework einen Shell-Skript.

4.3.6 Installation von androdeb

Der Entwickler von androdeb empfiehlt Ubuntu oder Debian als Host-Betriebssystem für die Installation von androdeb in Android. Analog zum Kompilieren von AOSP im Docker-Container mit Ubuntu 18.04 wird es entschieden, die Installation von androdeb auch mit Hilfe von Docker-Container mit Ubuntu 20.04 zu realisieren. Die Version 20.04 wird ausgewählt, weil das die aktuellste stabile Version von Ubuntu ist. Im ersten Schritt clont das Framework androdeb Repository in den Ordner `~/repos/adeb`, anschließend wird `\rootfs` im Docker Container mit Ubuntu gebaut. Nach der Installation von androdeb installiert das Framework im androdeb- Filesystem die notwendigen Linux Tools wie `git`, `nano`, `python`, `clang-7` und `gcc`.

4.3.7 Installation von BCC innerhalb von androdeb

In der aktuellen Version funktioniert androdeb (21.01.2021) innerhalb eines "chrooted environment", deswegen hat man nach dessen Installation keinen Zugriff auf einige wichtige Directories. Für das Funktionieren von eBPF-Tracepoints braucht man zum Beispiel den Zugriff auf `/sys/kernel/tracing/`, aber androdeb unterstützt das "out of the box" nicht. Um das Problem zu lösen, sendet das Framework mit Hilfe vom ADB ein Shell-Skript auf das AVD und führt das Shell-Skript aus. Dann wird das Shell-Skript für das Bauen von BCC gestartet. In dem Abschnitt 2.8.2 wird schon geschrieben, dass für das Anwenden von BCC in androdeb Android Kernel eine bestimmte Konfiguration haben muss. Das default Android Kernel entspricht diesen Kriterien

nicht. Ebenfalls wird ein modifiziertes Kernel für das Laden von Kernel-Modulen benötigt. In dem Abschnitt 4.3.2 wird die Methode des Erstellens vom modifizierten Android Kernel beschrieben.

5 Evaluation

In diesem Kapitel wird das Framework, welches der Implementierung des in dieser Arbeit vorgestellten Ansatzes entspricht, evaluiert. Die Evaluation besteht aus 2 Teilen: Bewertung des Erfolgs der Implementierung von jedem Schritt des automatisierten Erstellens eines Android Virtual Devices für Sicherheitsforscher, und Vergleich mit einer Cloud-basierten Forschungsumgebung für Android von Corellium.

5.1 Evaluation vom erstellten Android Virtual Devices für Sicherheitsforscher

In diesem Abschnitt wird bewertet, wie erfolgreich die einzelnen Teile vom Android Virtual Device (AVD) für Sicherheitsforscher funktionieren. Man überprüft, ob das automatisch erstellte AVD die Eigenschaften hat, welche in dem Kapitel 1 definiert werden. Als Setup der Evaluation verwendet man Manjaro OS (Kernel 5.4), weil dies das System des Autors ist. Zusätzlich wird Docker installiert und der Docker-Daemon gestartet. In diesem Setup erstellt man automatisch mit Hilfe vom Framework das AVD mit folgenden Parametern:

- Android Version: Android 10
- Kernel Version: 4.14
- Rooting Methode: Magisk 21.0
- Frida-Server Version: die aktuellste (der Stand am 18.01.2021 - 14.2.6)
- adeb soll installiert werden
- BCC ist innerhalb von adeb installiert
- “helloworld“-Kernel-Modul wird kompiliert und ins Kernel geladen

Die Bewertung vom erstellten AVD wird laut der folgenden Kriterien erfolgen:

- AVD funktioniert
- Die Userspace Applikationen können die Root-Rechte bekommen
- Frida-Server funktioniert
- adeb ist installiert, und es ist möglich die Debian-Tools da zu verwenden
- BCC funktioniert innerhalb von adeb und man kann HelloWorld-eBPF Programm ausführen
- “helloworld“-Kernel-Modul wird kompiliert und es wird ins Kernel geladen

In den folgenden Abschnitten werden die Methodiken und die Ergebnisse von der Überprüfungen von Eigenschaften des AVDs dargestellt.

5.1.1 Root-Rechte

Um zu verifizieren, ob eine Userspace Applikation auf dem AVD die Root-Rechte bekommen kann, wird die folgende Methode ausgewählt. Man installiert auf dem Emulator die Applikation "Terminal Emulator" [26], welche analog zum Linux-Terminal funktioniert. Mit Hilfe von dem Befehl `whoami` wird überprüft, wie der default-Benutzer heißt. Das wird im Listing 5.1 gezeigt.

```
1 $ whoami
2 u0_a103
```

Listing 5.1: *User-Name vom default Android-Benutzer*

Im Anschluss wird durch den Befehl su der Benutzer auf den Root-Benutzer gewechselt. Bei dem Ausführen vom Befehl fragt der Magisk, ob diese Applikation die Root-Rechte haben darf. Bei einer positiven Antwort wechselt der "Terminal Emulator" den Benutzer auf `root`. Das wird wieder mit dem Befehl `whoami` geprüft. Das Listing 5.2 stellt die Ergebnisse dar.

```
1 $ su
2 # whoami
3 root
```

Listing 5.2: *User-Name vom Root Android-Benutzer*

Aus dem Listing 5.2 erschließt sich, dass der aktuelle Benutzer `root` ist. Zusätzlich sieht man, dass nach dem Ausführen vom Befehl su der Shell Prompt sich von $ zu # ändert. So markiert der Terminal Emulator, dass die folgenden Befehle mit den Root-Rechten ausgeführt werden [44]. Daraus ergibt sich, dass das AVD erfolgreich gerootet wird.

5.1.2 Werkzeug für das Tracing und Hooking im Userspace

Für das Kontrollieren der Funktionsfähigkeit vom ausgewählten Tracing und Hooking Tool im Userspace (Frida), macht man das Tracing der Applikation "Terminal Emulator" mit Hilfe vom `frida-trace` Tool. Das ist ein Tool zum dynamischen Tracing von Funktionsaufrufen.[28]. In dem Listing 5.3 ist dargestellt, wie Frida die Verbindung mit dem "Terminal Emulator" erstellt und eine native Funktion `write` tracet. Jedes Mal, wenn diese Funktion aufgerufen wird, sieht man die Meldung `write()` im Terminal.

```
1 # frida-trace -U -i "write" com.termoneplus
2 Instrumenting...
3 write: Loaded handler at "/home/sergey/__handlers__/libc.so/write.js"
4 Started tracing 1 function. Press Ctrl+C to stop.
5            /* TID 0x167f */
6  13067 ms  write()
7            /* TID 0x1668 */
8  13073 ms  write()
9            /* TID 0x1707 */
```

Listing 5.3: *Auszug des Userspace Tracing mit frida-trace*

Aus diesem Beispiel geht hervor, dass das Hooking/Tracing Werkzeug für Userspace korrekt funktioniert.

5.1.3 Werkzeug für das Tracing im Kernelspace

Für das Testen der Funktionsfähigkeit des Werkzeugs für das Tracing im Kernelspace wird ein einfaches "helloworld" BCC-Programm verwendet, welches "Hello World\\n"bei jedem Aufruf von sys_clone Systemcall im Terminal schreibt. Der Python-Quellcode vom diesen Programm ist in dem Listing 5.4 vorgestellt. In der ersten Zeile wird der Pfad zum Interpreter definiert. Die zweite Zeile importiert BPF-Klasse von bcc-Library [29]. Der C-Code des eBPF-Programms befindet sich auf den Zeilen 5-7. Bei dem Aufruf wird die Funktion hello eine Nachricht schreiben. In der Zeile 11 wird mit Hilfe von BPF-Klasse der BPF-Bytecode generiert und ins Kernel geladen. Die Zeile 12 wählt Kpobes als Event Target, tracet sys_clone Systemcall und setzt die Aufruf-Funktion ein. Die letzte Zeile schreibt die Ergebnisse vom Tracing ins Terminal.

```
1  #!/usr/bin/python
2  from bcc import BPF
3
4  prog = """
5  int hello(void *ctx) {
6      bpf_trace_printk("Hello world\\n");
7      return 0;
8  }
9  """
10
11 b = BPF(text=prog)
12 b.attach_kprobe(event="sys_clone", fn_name="hello")
13 b.trace_print()
```

Listing 5.4: *Quellcode vom "Hello World" eBPF Programm*

Dieses Skript wird unter AVD innerhalb des androdeb fs gestartet. Das folgende Listing 5.5 stellt die Ergebnisse dar.

```
1  root@localhost:/# python hello_world.py
2  chdir(/lib/modules/4.14.175-g6f3fc953-dirty/build): No such file or directory
3  Traceback (most recent call last):
4    File "hello_world.py", line 11, in <module>
5      b = BPF(text=prog)
6    File "/usr/lib/python2.7/dist-packages/bcc/__init__.py", line 320, in __init__
7      raise Exception("Failed to compile BPF text")
8  Exception: Failed to compile BPF text
```

Listing 5.5: *Ausgabe vom eBPF Programm*

Daraus ergibt sich, dass das Kompilieren vom eBPF-Programm nicht funktioniert. Beim Überprüfen wird ersichtlich, dass beim Bauen vom BPF Compiler Collection einige Fehler aufgetreten sind. Die Fehlermeldungen werden in dem Listing 5.6 dargestellt. Die Zeilen 1-3, 7-8, 10-11 und 14-15 vom Listing 5.6 informieren, dass für das Zwischenspeichern von temporären Dateien dem Linker von clang der extra Speicherplatz fehlt.

```
1  /usr/bin/ld: final link failed: No space left on device
2  /usr/bin/ld: final link failed: No space left on device
```

```
3  clang: error: linker command failed with exit code 1 (use -v to see invocation)
4  make[2]: *** [examples/cpp/CMakeFiles/CGroupTest.dir/build.make:154: examples/cpp/
       CGroupTest] Error 1
5  make[1]: *** [CMakeFiles/Makefile2:975: examples/cpp/CMakeFiles/CGroupTest.dir/all]
       Error 2
6  make[1]: *** Waiting for unfinished jobs....
7  clang: error: linker command failed with exit code 1 (use -v to see invocation)
8  make[2]: *** [examples/cpp/CMakeFiles/FollyRequestContextSwitch.dir/build.make:154:
       examples/cpp/FollyRequestContextSwitch] Error 1
9  make[1]: *** [CMakeFiles/Makefile2:932: examples/cpp/CMakeFiles/
       FollyRequestContextSwitch.dir/all] Error 2
10 /usr/bin/ld: final link failed: No space left on device
11 clang: error: linker command failed with exit code 1 (use -v to see invocation)
12 make[2]: *** [examples/cpp/CMakeFiles/CPUDistribution.dir/build.make:154: examples/cpp
       /CPUDistribution] Error 1
13 make[1]: *** [CMakeFiles/Makefile2:1018: examples/cpp/CMakeFiles/CPUDistribution.dir/
       all] Error 2
14 /usr/bin/ld: final link failed: No space left on device
15 clang: error: linker command failed with exit code 1 (use -v to see invocation)
16 make[2]: *** [examples/cpp/CMakeFiles/LLCStat.dir/build.make:154: examples/cpp/LLCStat
       ] Error 1
17 make[1]: *** [CMakeFiles/Makefile2:1061: examples/cpp/CMakeFiles/LLCStat.dir/all]
       Error 2
18 make: *** [Makefile:141: all] Error 2
```

Listing 5.6: *Fehlermeldung beim Bauen von BCC*

Eine Recherche nach dieser Fehlermeldung ergibt, dass auch andere Anwender von BCC diesen Fehler beim Kompilieren erhalten und der zu geringe Speicherplatz im Framework diesen Fehler verursacht [52]. Da dieser Fehler in vorherigen Versionen von BCC nicht aufgetreten ist, bleibt im Rahmen dieser Arbeit keine Zeit, um ihn zu beheben.

5.1.4 Erweiterbarkeit des Kernels

Wie in dem Abschnitt 4.3.2 geschrieben ist, kompiliert man zusammen mit dem Custom-Kernel ein "helloworld" Kernel-Modul. Dieses Kernel-Modul soll beim Laden und Entladen die Nachrichten in Kernel Message Buffer schreiben. Die Nachrichten sind entsprechend "`Hello Kernel World ....\n`" und "`Goodbye Kernel World\n`". Die Datei vom Kernel-Modul befindet sich im Ordner `/data/` im AVD. Um zu prüfen, ob das Kernel richtig konfiguriert ist und ob das "helloworld" Kernel-Modul erfolgreich kompiliert, soll man es in der Laufzeit Laden und Entladen. Das ist in dem Listing 5.7 dargestellt. Zuerst bekommt man den Zugriff auf ADB Shell und geht zum Ordner `/data/` (1. und 2. Zeile vom Listing). Mit dem Befehl `insmod hello.ko` lädt man das Modul. Durch den folgenden Befehl (Zeile 4) überprüft man, ob sich in Kernel Message Buffer die Nachricht von dem Kernel-Module befindet. Analog prüft man mit dem Befehl `rmmod hello.ko`, ob das Entladen vom Kernel-Modul funktioniert (Zeilen 6, 7).

```
1 $ adb shell
2 generic_x86_64:/ # cd data/
3 generic_x86_64:/data # insmod hello.ko
4 generic_x86_64:/data # dmesg | grep "Hello"
5 [  253.840440] Hello Kernel World....
6 generic_x86_64:/data # rmmod hello.ko
```

```
7 generic_x86_64:/data # dmesg | grep "Goodbye"
8 [  520.099916] Goodbye Kernel World
```

Listing 5.7: *Laden und Entladen vom Kernel-Modul*

Angesicht dieser Ergebnisse kann man sagen, dass die Erweiterbarkeit von Kernel erfolgreich realisiert wird.

5.1.5 Ergebnisse

Die genannten Ergebnisse (vgl. Abschnitte 5.1.1 bis 5.1.4) von den Prüfungen der Funktionsfähigkeit einzelner Teiler vom AVD zeigen, dass das AVD funktioniert, gerootet ist, das Werkzeug für Tracing und Hooking im Userspace (Frida-Server) funktioniert und das Kernel mit Hilfe von Kernel-Modulen erweiterbar ist. Das einzige Tool, welches nicht funktioniert, ist das Werkzeug für Tracing im Kernelspace (BPF Compiler Collection). Die für BCC vorbereitete Umgebung (androdeb) ist funktionsfähig und das Problem beim Kompilieren ist mit dem Linker verbunden und potenziell lösbar. Weil die Bachelorarbeit in der Zeit begrenzt ist, wird das Problem mit BCC nicht gelöst. Die folgende Tabelle 4 fasst die Ergebnisse der Evaluation des AVDs zusammen.

Tabelle 4: *Bewertung der Funktionsfähigkeit des AVDs*

Kriterien der Bewertung	**Funktionsfähigkeit**
AVD funktioniert	⊕
Root-Rechte für Userspace-Apps	⊕
Tracing/Hooking im Userspace	⊕
androdeb (adeb) funktioniert	⊕
Tracing im Kernelspace	⊖
Erweiterbarkeit des Kernels	⊕

⊕ - der Parameter ist erfolgreich implementiert
⊖ - der Parameter ist nicht implementiert

Aus der Tabelle 4 wird deutlich, dass nur ein Parameter von den bewerteten Parametern (Tracing im Kernelspace) nicht erfolgreich implementiert werden konnte. Die anderen fünf funktionieren so wie geplant. Wegen der zeitlichen Begrenzung der Bachelorarbeit ist das Framework für das Bauen des AVDs für Sicherheitsforscher kein fertiges Produkt. Es kann eher als ein Prototyp für die Entwicklung eines Build-Systems für Android Virtual Devices für Sicherheitsforscher gesehen werden.

5.2 Vergleich vom erstellten AVD mit dem Corellium AVD

Für das Vergleichen vom Corellium AVD für Sicherheitsforscher (CORSEC) und AVD, welches im Rahmen dieser Arbeit automatisch generiert wird, sind die folgende Kriterien ausgewählt: CPU Architektur vom AVD, Lokation vom AVD, Kosten, Erweiterbarkeit des Kernels, Rooting, Werkzeug für Tracing und Hooking im Userspace, Werkzeug für Tracing im Kernelspace. Die Mehrheit von modernen Smartphones basieren auf ARM-Architektur und einige Applikationen können nur unter dieser Architektur ausgeführt werden [11]. Die Mehrheit von modernen PCs basieren auf x86_64-Architektur und die Emulation von AVD mit ARM-CPU ist sehr langsam

[11]. Deswegen ist die CPU-Architektur für den Vergleich wichtig. Beim Verwenden des AVDs auf dem lokalen Computer nutzt man die Hardware (CPU, RAM, Festspeicher) vom Computer und ist durch diese Hardware begrenzt. Falls das AVD als ein Cloud-Service funktioniert, werden die Ressourcen des Servers verwendet. Aber es gibt die Möglichkeit, dass die Aktivitäten innerhalb des AVDs von dem Besitzer des Servers beobachtbar sind. Aus diesem Grund ist die Lokation des AVDs wichtig. Der Parameter "Kosten" ist von der Lokation des AVDs abhängig. Der Cloud-basierte Emulator ist ein kommerzielles Projekt und das Verwenden von den Ressourcen und Technologien der Firma ist nicht kostenlos. Im Gegensatz dazu ist das Verwenden vom AVD auf der eigenen Hardware kostenlos. Die anderen Parameter sind ebenfalls wichtig, da diese der Eingangs definierten Eigenschaften einer AVD für Sicherheitsforscher entsprechen (vgl. Kapitel 1). So kann man überprüfen, ob das CORSEC diese Kriterien auch erfüllt.
Das Projekt vom automatisierten Erstellen des AVDs ist free und Open Source. Man erzeugt den Emulator mit allen notwendigen Tools auf einem lokalen Computer. Deswegen kann man nur x86_64-Architektur vom AVD verwenden. Man hat den Zugriff auf die Root-Berechtigungen mit ADB, und die Userspace Applikationen bekommen die Root-Rechte mit Hilfe von Magisk. Der Frida-Server ist installiert.
Alle Informationen über CORSEC sind der offiziellen Support-Seite vom Corellium entnommen [19]. CORSEC ist ein Cloud-Service. Der Service verwendet die Server mit ARM64-Architektur. Abhängig von der CPU-Cores Anzahl kostet die Nutzung von CORSEC ab 99$ pro Monat (2 Cores) für Individuen. Es ist möglich ein Custom Kernel zu verwenden. Das Kernel muss zuerst lokal kompiliert werden. Man kann den Frida-Server installieren, aber der Forscher muss diesen zuerst herunterladen und manuell mit Hilfe von ADB installieren und starten. Seit Januar 2021 unterstützt CORSEC das CoreTrace. Das ist eine proprietäre hypervisor-basierte Kernel-Tracing Methode. Das hypervisor-basierte Tracing macht die Anwendung von anti-debugging Technicken viel komplizierter.[19, 17]

Die folgende Tabelle 5 fasst die Eigenschaften vom erstellten AVD und CORSEC zusammen.

Tabelle 5: *Vergleich von Frameworks AVD und CORSEC*

Vergleichskriterium	**Framework AVD**	**CORSEC**
Architektur	x86_64	arm64
Lokation von AVD	lokaler Computer	Cloud-Service
Kosten	kostenlos	ab 99$ pro Monat
Erweiterbarkeit des Kernels	⊕	das Verwenden ist möglich
Root-Rechte	⊕	⊕
Tracing im Userspace	Frida-Server	strace, das Verwenden von Frida ist möglich
Tracing im Kernelspace	⊖	⊕

⊕ - die Eigenschaft ist vorhanden
⊖ - die Eigenschaft ist nicht vorhanden

Aus der Tabelle 5 wird deutlich, dass CORSEC und AVD vom Framework 2 unterschiedliche CPU Architekturen anwenden. Die arm64-Architektur wird dabei bevorzugt, weil die Mehrheit von modernen Smartphones diese Architektur hat. CORSEC ist ein kommerzielles Projekt und man kann nach dem Abonnieren ein AVD auf dem Server von der Firma erstellen. Das Ergebnis dieser Bachelorarbeit ist kostenlos, aber in diesem Fall werden eigene Computer-Ressourcen verwendet. Das AVD vom Framework hat sofort das Custom-Kernel mit der Möglichkeit für das Laden und

Entladen von Kernel-Modulen. CORSEC lässt das Custom-Kernel mit ladbaren Kernel-Modulen anwenden, aber der Forscher muss das Kernel und die Kernel-Module zuerst lokal kompilieren. Beide Lösungen haben die Root-Berechtigungen für ADB und Userspace Applikationen. CORSEC hat `strace` für das Userspace-Tracing (basiert auf `ptrace`), man kann auch Frida-Server in CORSEC installieren, aber der Benutzer macht das selber. Das AVD vom Framework hat schon die installierte aktuellste Version vom Frida-Server. Im aktuellen Stand hat das AVD vom Framework kein Instrument für Kernelspace Tracing, CORSEC hat CoreTrace dafür.
Das im Rahmen dieser Arbeit erstellte Framework stellt eine prototypische Implementierung zur automatisierten Erstellung von AVDs für Sicherheitsforscher dar. Dies konnte erfolgreich implementiert werden. Da viele dieser Komponenten stetig aktualisiert werden war eine Anpassung an diese aktualisierten Komponenten im Rahmen dieser Arbeit nicht möglich. Über Möglichkeiten dies in künftigen Entwicklungen miteinzubeziehen wird im Kapitel 7 diskutiert . [19, 18]

6 Fazit

Ein passendes Werkzeug zu haben ist ein wichtiger Schlüssel für einen Arbeitserfolg in jedem Bereich. Das gilt auch für die Sicherheitsforschung unter Android. Die Android Virtual Devices (AVD) sind sehr wichtig für eine dynamische Analyse von Malware. Vor dem Anwenden vom AVD muss dieses noch richtig eingestellt werden. Dieser Prozess ist routinemäßig und braucht Zeit. Um die Zeit von Sicherheisforschern zu sparen, wird durch die Ausarbeitung dieser Arbeit ein automatisiertes Build-System für Android Virtual Devices für Sicherheitsforscher entwickelt. Das gebaute AVD entspricht den in der Einleitung erwähnten Kriterien .
Um sich dem Thema anzunähern, werden zu Beginn dieser Arbeit die Grundlagen von Android, Virtualisierung, Rooting von Android, Tracing im Userspace als auch im Kernelspace erklärt. Auf Basis einer Recherche nach verwandten Arbeiten wird gezeigt, wie die einzelnen Teilaspekte der Arbeit implementiert sein können. Nun wird die optimalste Technik für das Einstellen von Teilaspekten des AVDs gewählt, dafür müssen alle Techniken für jeden Teilaspekt verglichen werden. Anschließend implementiert man das Framework für das automatisierte Build-System für Android Virtual Devices für Sicherheitsforscher. In einer folgenden Überprüfung wird geschaut, ob das automatisch gebaute AVD alle in der Einleitung formulierten Kriterien erfüllt. Zusätzlich vergleicht man die Eigenschaften vom automatisch gebauten AVD mit dem neuen cloud-basierten AVD für Sicherheitsforscher von Corellium.
Die wichtigsten Erkenntnisse aus dem Überprüfen der Funktionsfähigkeit des automatisch gebauten AVDs sind, dass fünf von sechs Kriterien erfolgreich implementiert sind und ein Kriterium nicht erfüllt ist. Die realisierten Kriterien sind: Starten vom AVD, AVD ist gerootet, Werkzeug für das Tracing im Userspace ist installiert, Debian-basierte Umgebung (adeb) ist installiert und die Erweiterbarkeit vom Kernel ist funktionsfähig. Der einzige nicht realisierte Parameter ist das Werkzeug für das Tracing im Kernelspace. Wegen der zeitlichen Begrenzung der Arbeit ist das Problem nicht gelöst.
Zusätzlich wird das AVD vom Framework mit CORSEC AVD von Corellium verglichen. Diese zwei Lösungen haben unterschiedliche Implementierungen und unterscheiden sich stark in der Lokation von AVD, der CPU-Architektur, den Kosten des Anwendens und dem Werkzeug für das Tracing im Kernelspace. Die Lösung, die im Rahmen dieser Arbeit erstellt wird, hat den Vorteil, dass der Endanwender Herr der Daten ist. Dies ist insbesondere bei sensiblen Daten von besonderer Wichtigkeit. Die Tatsache, dass eine so große Firma wie Corellium ein Produkt für Sicherheitsforscher unter Android mit ähnlichen Eigenschaften auf den Markt gebracht hat, zeigt, dass spezialisierte AVDs für Sicherheitsforscher notwendig sind.

7 Ausblick

Die wachsende Popularität von spezialisierten Android Virtual Devices für Sicherheitsforscher macht das Thema attraktiv für die weitere Entwicklung. In diesem Abschnitt werden die Ideen für die zukünftige Arbeit dargestellt, welche sich auf der Basis dieser Arbeit gründen.

7.1 Lösen vom Problem mit BCC

In dem AVD, welches mit Hilfe vom Framework kompiliert wird, funktioniert BPF Compilier Collection nicht (vgl. Abschnitt 5.1.3). Deswegen können die eBPF-Applicationen nicht kompiliert werden.

7.2 Flexible Auswahl von AOSP Version und Android Kernel Version

Das Framework, welches im Rahmen dieser Arbeit implementiert wird, zeigt die Funktionsfähigkeit solcher Variante des Erstellens von AVD für Sicherheitsforscher. Aber die Versionen von AOSP und Kernel sind fest kodiert und der Endbenutzer hat keine Möglichkeit für eine genauere Einstellung vom AVD. Dafür muss ein User-Interface entwickelt werden, wo der Benutzer die notwendigen Versionen von AOSP und Kernel auswählen kann. Zusätzlich muss die Kompatibilität von Magisk Versionen mit AOSP Versionen geprüft werden.

7.3 Cross-Kompilieren von Kernel-Modulen

In der aktuellen Implementierung werden die Kernel-Module zusammen mit dem Kernel kompiliert. Falls die Konfiguration und Quelldateien vom Kernel nach dem Kompilieren des Kernels nicht geändert werden, werden beim nächsten Mal nur die neuen Kernel-Module kompiliert und das Kompilieren vom Kernel erfolgt nicht. Um den Prozess vom Erstellen der neuen Kernel-Module zu beschleunigen, kann das Cross-Kompilieren verwendet werden. In diesem Fall müssen keine Überprüfungen vom Kernel-Zustand durchgeführt werden.

8 Appendix

Unter folgendem URL https://github.com/SergShel/autobuild_avd_for_dyn_analysis befinden sich:

- Digitale Kopie dieser Bachelorarbeit
- Quellcode vom automatisierten Build-System für AVD für Sicherheitsforscher
- README-Datei mit den Informationen über allen Abhängigkeiten von diesem Projekt und Instruktionen für das Starten vom automatisierten Build-System.

9 Literatur

[1] *#SUPERSU*. URL: https://supersu.co/. (zugegriffen: 22.10.2020).

[2] *11 Best Android Developer Tool to Get You Started on Android Development*. URL: https://www.mockplus.com/blog/post/android-developer-tool. (zugegriffen: 21.01.2021).

[3] *About the Android Open Source Project*. URL: https://source.android.com/. (zugegriffen: 09.10.2020).

[4] *Android 10: Emulation of Magisk/SuperSU on an AOSP AVD*. URL: https://anthony-f-tannous.medium.com/android-10-emulation-of-magisk-supersu-on-an-aosp-avd-de93ed080fad. (zugegriffen: 22.12.2020).

[5] *Android Architecture. Kernel. Extending the Kernel with eBPF*. URL: https://source.android.com/devices/architecture/kernel/bpf. (zugegriffen: 14.10.2020).

[6] *Android Architektur. Kernel. Android Common Kernels*. URL: https://source.android.com/devices/architecture/kernel/android-common. (zugegriffen: 16.01.2021).

[7] *Android Developers. Android Studio. Android Debugger Bridge*. URL: https://developer.android.com/studio/command-line/adb. (zugegriffen: 17.01.2021).

[8] *Android Source. Security Enhancements in Android 4.3*. URL: https://source.android.com/security/enhancements/enhancements43. (zugegriffen: 28.01.2021).

[9] *Android-Architektur*. URL: https://source.android.com/devices/architecture. (zugegriffen: 09.10.2020).

[10] *Arch Linux. Capabilities*. URL: https://wiki.archlinux.org/index.php/Capabilities#Implementation. (zugegriffen: 28.01.2021).

[11] *Arm vs x86: Instruction sets, architecture, and all key differences explained*. URL: https://www.androidauthority.com/arm-vs-x86-key-differences-explained-568718/. (zugegriffen: 07.02.2021).

[12] *BPF Compiler Collection (BCC)*. URL: https://github.com/iovisor/bcc. (zugegriffen: 27.12.2020).

[13] *bpf, tracing: introduce bpf raw tracepoints*. URL: https://lwn.net/Articles/748352/. (zugegriffen: 05.11.2020).

[14] *BPFd (Berkeley Packet Filter Daemon)*. URL: https://github.com/joelagnel/bpfd. (zugegriffen: 26.12.2020).

[15] *Comparison of Different Android Root-Detection Bypass Tools*. URL: https://medium.com/secarmalabs/comparison-of-different-android-root-detection-bypass-tools-8fd477251640. (zugegriffen: 24.01.2021).

[16] *Corellium release notes*. URL: https://headwayapp.co/corellium-release-notes. (zugegriffen: 20.01.2021).

[17] *Corellium. Blog. Introduction to CoreTrace*. URL: https://corellium.com/blog/intro-to-coretrace. (zugegriffen: 18.02.2021).

[18] *Corellium. Support. Advanced Settings: Android*. URL: https://support.corellium.com/hc/en-us/articles/360053564434-Advanced-Settings-Android. (zugegriffen: 20.01.2021).

[19] *Corellium. Using Your Virtual Device*. URL: https://support.corellium.com/hc/en-us/articles/360056222614-Getting-Started. (zugegriffen: 18.01.2021).

[20] *CyanogenMod*. URL: https://en.wikipedia.org/wiki/CyanogenMod. (zugegriffen: 07.01.2021).

[21] Kyle Denney, Cengiz Kaygusuz und Julian Zuluaga. „A Survey of Malware Detection Using System Call Tracing Techniques". In: (2018).

[22] *Difference Between Static Malware Analysis and Dynamic Malware Analysis*. URL: http://www.differencebetween.net/technology/difference-between-static-malware-analysis-and-dynamic-malware-analysis/. (zugegriffen: 24.01.2021).

[23] *Docker Docs*. URL: https://docs.docker.com/. (zugegriffen: 13.10.2020).

[24] *eBPF Documentation. What is eBPF?* URL: https://ebpf.io/what-is-ebpf/. (zugegriffen: 05.11.2020).

[25] *eBPF super powers on ARM64 and Android. Powerful Linux Tracing for Android*. URL: http://www.joelfernandes.org/resources/bcc-ospm.pdf. (zugegriffen: 26.12.2020).

[26] *F-Droid. Terminal Emulator. Turn your device into a computer terminal*. URL: https://f-droid.org/en/packages/com.termoneplus/. (zugegriffen: 25.01.2021).

[27] Joel Fernandes. *Adeb. Github.com*. URL: https://github.com/joelagnel/adeb. (zugegriffen: 14.10.2020).

[28] *Frida. Docs. Frida-Trace*. URL: https://frida.re/docs/frida-trace/. (zugegriffen: 27.01.2021).

[29] *Github.Iovisor/bcc/src/python/bcc/__init__.py*. URL: https://github.com/iovisor/bcc/blob/master/src/python/bcc/__init__.py. (zugegriffen: 01.02.2021).

[30] Mohamed Hassan und Lutta Pantaleon. „An investigation into the impact of rooting android device on user data integrity". In: *2017 Seventh International Conference on Emerging Security Technologies (EST)*. IEEE. 2017, S. 32–37.

[31] *How To Root Any Android Device 2020*. URL: https://www.rootmeguide.com/how-to-root-any-android-device/. (zugegriffen: 22.10.2020).

[32] *How-to Guide: Defeating an Android Packer with FRIDA*. URL: https://www.fortinet.com/blog/threat-research/defeating-an-android-packer-with-frida. (zugegriffen: 24.01.2021).

[33] *How-to Write a Kernel Module for Android*. URL: https://abdullahyousafzaii.wordpress.com/2015/08/02/how-to-write-a-kernel-module-for-android/. (zugegriffen: 07.01.2021).

[34] Zhenghui Lee. *Exploring USDT Probes on Linux*. URL: https://leezhenghui.github.io/linux/2019/03/05/exploring-usdt-on-linux.html. (zugegriffen: 22.01.2021).

[35] *Linux Hook*. URL: https://www.cnblogs.com/pannengzhi/p/5203467.html. (zugegriffen: 25.01.2021).

[36] *Linux Kernel Module Programming - 02*. URL: https://www.youtube.com/watch?v=o768iZKtzBA&ab_channel=SolidusCode. (zugegriffen: 19.12.2020).

[37] *LKDDB main index. Index of Linux kernel configurations*. URL: https://cateee.net/lkddb/web-lkddb/. (zugegriffen: 16.01.2021).

[38] *Magisk Manager APK 8.0.0 and Magisk 21.0 ZIP*. URL: https://magisk.me/. (zugegriffen: 14.10.2020).

[39] *Magisk. Release Notes*. URL: https://topjohnwu.github.io/Magisk/releases/. (zugegriffen: 18.02.2021).

[40] *MALWARE REPORTS. IT threat evolution Q2 2020. Mobile statistics*. URL: https://securelist.com/it-threat-evolution-q2-2020-mobile-statistics/98337/. (zugegriffen: 08.10.2020).

[41] *ptrace(2) - Linux man page*. URL: https://linux.die.net/man/2/ptrace. (zugegriffen: 26.12.2020).

[42] Ole André V. Ravnås. *FRIDA*. URL: frida.re. (zugegriffen: 08.10.2020).

[43] *Seccomp Filter in Android*. URL: https://android-developers.googleblog.com/2017/07/seccomp-filter-in-android-o.html. (zugegriffen: 28.01.2021).

[44] William Shotts. *The Linux command line: a complete introduction*. No Starch Press, 2019.

[45] *Stackexchange. How Magisk works?* URL: https://android.stackexchange.com/questions/213167/how-magisk-works. (zugegriffen: 28.01.2021).

[46] *Statista. Mobile operating systems' market share worldwide from January 2012 to July 2020*. URL: https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/. (zugegriffen: 04.08.2020).

[47] *Statista. Number of smartphone users worldwide from 2016 to 2021*. URL: https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/. (zugegriffen: 04.08.2020).

[48] Joe Sylve u. a. „Acquisition and analysis of volatile memory from android devices". In: *Digital Investigation* 8.3-4 (2012), S. 175–184.

[49] *Tracing Linux: Fast, Compatible, Complete*. URL: https://www.confluera.com/post/tracing-linux-fast-compatible-complete. (zugegriffen: 05.11.2020).

[50] *Tracing on Linux. Elena Zannoni (elena.zannoni@oracle.com) Linux Engineering, Oracle America*. URL: https://events.static.linuxfound.org/images/stories/pdf/lceu2012_zannoni.pdf. (zugegriffen: 18.12.2020).

[51] *Understanding Linux Daemons*. URL: https://www.inmotionhosting.com/support/server/linux/understanding-linux-daemons/. (zugegriffen: 04.02.2021).

[52] *Unix.Stackexchange.Encountering this error /usr/bin/ld: final link failed: No space left on device*. URL: https://unix.stackexchange.com/questions/16137/encountering-this-error-usr-bin-ld-final-link-failed-no-space-left-on-device. (zugegriffen: 11.02.2021).

[53] *Wiki Arch Linux. Kernel module*. URL: https://wiki.archlinux.org/index.php/Kernel_module. (zugegriffen: 19.12.2020).

[54] *Wikikedia. Android software development*. URL: https://en.wikipedia.org/wiki/Android_software_development. (zugegriffen: 17.01.2021).

[55] *Wikipedia. Setuid*. URL: https://en.wikipedia.org/wiki/Setuid. (zugegriffen: 28.01.2021).

[56] *xda-developers.Magisk vs SuperSU*. URL: https://www.xda-developers.com/magisk-vs-supersu/. (zugegriffen: 22.10.2020).

[57] Roger Ye. *Android System Programming*. Packt Publishing Ltd, 2017.

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Selbstständigkeitserklärung

Hiermit versichere ich, die vorliegende Bachelorarbeit ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Bonn, 27. Februar 2021

Siarhei Sheludzko