# Checkpoint/Restore in Fuzzing

Malte Klaassen

17.11.2018

**Zusammenfassung**

Fuzzing hat sich in den letzten Jahren als mächtige Methode zum Auffinden möglicher Sicherheitslücken in Software etabliert. Bei der Anwendung von Fuzzing auf Applikationen mit längerem Setup oder komplexerem internen Zustandsmodell stößt Fuzzing allerdings schnell auf Performanceprobleme oder versucht höheren manuellen Testaufwand. Wir untersuchen in dieser Arbeit inwieweit Checkpoint/Restore die genannten Probleme des Fuzzing mitigieren kann. Nach einer Zusammenfassung derzeitiger Fuzzing- und C/R-Konzepte stellen wir die Struktur eines integrierten C/R-Fuzzers dar. Weiterhin präsentieren wir einen generischen Ansatz C/R innerhalb klassischer Fuzzing-Engines zu verwenden. Diesen erweitern wir für die spezifische Problemklasse des Testens von Client-Server-Systemen. Beispielhaft untersuchen wir eine Integration von CRIU als prominentes C/R-System in LLVM libFuzzer und afl. Wir präsentieren Performancemessungen für CRIU und legen auftretende Probleme versuchter Integrationen dar.

## Inhaltsverzeichnis

# 1 Einleitung

Mit zunehmender Prävalenz und Prominenz von IT-Systemen ist das Thema der Sicherheit eben dieser Systeme zunehmend in den Fokus gerückt. In der öffentlichen Wahrnehmung stehen zumeist Fälle im Fokus, die Nutzer direkt betreffen. Prominente Beispiele sind Hacks oder Leaks bei Online-Plattformen wie Facebook[4] oder Malware-Kampagnen wie WannaCry[5], zumeist ausgelöst durch Probleme in der Operational Security und verzögerte Patches. Seltener in der Öffentlichkeit stehen die tatsächlichen technischen Sicherheitslücken, die diesen zugrunde liegen - wenn dann nur in Fällen mit starker Öffentlichkeitswirkung wie beispielsweise Heartbleed[6].
Bei dem Umgang mit derartigen Sicherheitsproblemen gibt es zwei dominante Klassen von Strategien: Proaktive Strategien, welche versuchen, mögliche Probleme und Sicherheitslücken als ganzes zu verhindern, und reaktive Strategien, welche versuchen, die Effekte von Sicherheitslücken zu mindern und eine Ausnutzung zu erschweren. Über die Jahrzehnte haben sich dabei verschiedene Ansätze und Methoden zur proaktiven Erkennung, Behebung und Verhinderung von diesen Sicherheitsproblemen entwickelt, beispielsweise[7]:

- Formulierung von Sicherheitsanforderungen bereits während und vor der Designphase
- Formale Beweise, beispielsweise bei kryptografischen oder Netzwerk-Protokollen oder sogar von Programmen, insbesondere in der Funktionalen Programmierung
- Compilerchecks und -safeguards sowie statische Codeanalyse
- (Externe) Reviews der Software und Penetrationtesting
- Klassisches Sicherheitstesting

Leider sind nicht immer alle diese Methoden anwendbar und haben jeweils ihre eigenen Stärken und Schwächen. So sind formale Beweise der Sicherheit von Programmen außerhalb von einigen modernen Hochsprachen häufig nicht möglich. Klassisches Sicherheitstesting, zum Beispiel mithilfe von Runtime-Sanitizer-Checks, ist zumeist beschränkt auf die Vorstellungskraft des Testers beziehungsweise auf sein Verständnis der zu testenden Software; Grenzfälle oder komplett unerwartete, aber mögliche, Eingaben sind mit klassischem Testing schwer umfassend

abzudecken.

# 2 Fuzzing

Fuzzing ist eine Test-Methode, die teil-automatisiert versucht, genau diese Art von mit klassischen Test-Methoden schwer abzudeckenden Fällen umfassend zu erkennen, indem, (statt wie im klassischen Testing, bei dem nur von Menschen mehr oder weniger präzise beschriebene Testfälle ausgeführt werden) ein Codestück, das so genannte Fuzz-Target, wiederholt mit computergenerierten zufälligen Inputs getestet wird. Dabei wird die Ausführung vom Fuzzer überwacht, um beispielsweise Programmabstürze, das Lesen aus unitialisiertem Speicher oder auch undefiniertes Verhalten zu erkennen und zu der problematischen Codestelle zurück zu verfolgen.

## 2.1 Klassifizierung von Fuzzern

Es gibt eine ganze Reihe von Ansätzen zur Klassifizierung von Fuzzern anhand verschiedener Kriterien und entlang verschiedener Achsen. Wir haben uns für diese Arbeit für eine Klassifizerung anhand zweier Eigenschaften entschieden: Ob sich ein Fuzzer beim Testen der Struktur des Programmes und der Struktur der erwartenden Eingaben bewusst ist.

- Wenn der Fuzzer sich der Struktur des Programmes bewusst ist und dieses Wissen im Testing nutzt, um bessere Code-Coverage zu erreichen, spricht man von einem White-Box- oder Gray-Box-Fuzzer[1], je nach Technologie. Besitzt der Fuzzer keine Kenntnisse über die Programmstruktur und generiert programmunabhängige Zufallsinputs, spricht man von einem Black-Box-Fuzzer.
- Wenn der Fuzzer sich der erwarteten Struktur des Inputs, beispielsweise bei einem bestimmten Dateiformat oder Netzwerkprotokoll, bewusst ist, spricht man von einem Smart Fuzzer, ansonsten von einem Dumb Fuzzer. Häufig kann dieses Wissen und Informationen darüber, wie Inputs generiert werden sollen beim Start des Fuzzers diesem als sogenanntes Dictionary übergeben werden.

Die meisten modernen Fuzzer können sowohl als Black- als auch als Gray- beziehungsweise White-Box-Fuzzer und sowohl als Smart als auch als Dumb Fuzzer betrieben werden, diese Terme werden zumeist auch zur Beschreibung von Operationsmodi moderner Fuzzer verwendet. Neue Inputs werden anhand der verfügbaren Informationen, also Dictionaries und dem bisher bekannten Corpus (einer Sammlung von bereits bekannten, interessanten Testfällen), generiert. Diese Generierung geschieht beispielsweise durch die Mutation oder Kombination von alten Inputs. Wird ein solcher Input bei der Ausführung, beispielsweise durch die Code-Coverage-Untersuchung, als interessant für das weitere Fuzzing bewertet, so er wird dem Corpus für die weitere Generierung von neuen Inputs hinzugefügt.

[1] Diese Art des Fuzzings wird auch als Coverage-Guided-Fuzzing bezeichnet.

## 2.2 Historischer Überblick

Testing mit randomisierten Inputs ist kein sonderliche neuer Ansatz[8], jedoch ist die Effizienz von naivem Random Input Test (also Black-Box-Fuzzing) offensichtlich beschränkt, und der größte Teil der Fehler, die auf diese Art und Weise gefunden werden können, ist relativ simpel und selten in den Tiefen des Programmes versteckt. Das heißt, es handelt sich zumeist um Fehler, die vergleichsweise einfach auch durch Code-Reviews oder klassische Tests gefunden werden könnten. Entsprechend hatte Random Input Testing für recht lange Zeit nur eine relativ geringe Relevanz, sowohl in der Qualitätssicherung als auch in der Sicherheitsforschung.
Mit dem Aufkommen von effizienteren Gray- und White-Box-Fuzzern und zunehmender Relevanz von IT-Sicherheit in den späten 2000ern entwickelte sich ein breiteres Interesse an Fuzzing[9]. Es entstanden einige Open Source Fuzzern und Fuzzing-Frameworks und Fuzzing wurde Kernbestandteil des Sicherheits-Testings vieler relevanter Open Source Software Projekte, vorangetrieben insbesondere durch Projekte wie Googles OSS-Fuzz, ein Service zum kontinuierlichen Testen verschiedenster Open Source Software durch mehrere verschiedene Fuzzer.

## 2.3 Beschränkungen im Fuzzing

Leider ist natürlich auch Fuzzing nicht ohne Probleme und Beschränkungen. Da Fuzzing zu großen Teilen darauf beruht, eine große Menge an Eingaben zu testen, sinkt die Effizienz des Fuzzings deutlich, wenn die Ausführung des Fuzz-Targets zu lange dauert. Dies trägt dazu bei, dass das Fuzzen komplizierter, insbesondere interaktiver Systeme nicht ohne weiteres effizient möglich ist. Dieses Problem wird zumeist gelöst, indem man komplexe Systeme in kleinere, simple Subsysteme aufteilt und diese unabhängig voneinander fuzzt. So wird beispielsweise unnötiger Overhead im Setup des Fuzz-Targets vermieden. Dies hat aber natürlich auch Nachteile: Dem Tester muss hier nun die Struktur des zu testenden Systems bekannt sein, er muss das komplexe System in kleinere Systeme selbst zerlegen und dann für jedes der Subsysteme einzelne Fuzz-Targets definieren - dies ist zeit- und arbeitsaufwändig und, ähnlich wie beim klassischen Testing, anfällig für Fehler des Testers.

## 2.4 LLVM libFuzzer

LLVM libFuzzer[2] ist eine moderne Fuzzing-Engine, die seit 2015 unter anderem von Kostya Serebryany entwickelt wird[13]. Das primäre Ziel des libFuzzer-Projektes ist es, eine effiziente und mächtige, dabei jedoch leicht zu nutzende Fuzzing-Engine zu entwickeln, um eine breitere Nutzung des Fuzzings zum Auffinden von Sicherheitslücken, auch außerhalb von Expertenkreisen, zu ermöglichen. Dies spiegelt sich insbesondere in dem geringen benötigten Aufwand wieder, um ein einfaches Fuzz-Target fuzzen zu können:

1. Es muss ein Fuzz-Target geschrieben werden. Ein Fuzz-Target ist ein Stück Code, beispielsweise in C oder C++, welches eine Funktion `int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size)` definiert. Diese Funktion erhält den Input durch `*Data` und soll diesen dann auf den eigentlich zu testenden Code anwenden.

2. Dieses Fuzz-Target wird nun mit der clang-Compilerflag `-fsanitize=fuzzer` sowie weiteren Sanitizer-Compilerflags wie `address` für AddressSanitizer, `signed-integer-overflow` für UndefinedBehaviourSanitizer oder `memory` für MemorySanitizer kompiliert. Dies verlinkt auch automatisch das Fuzz-Target mit dem Main-Symbol von libFuzzer.

3. Die entstandene Binary enthält nun bereits das Fuzz-Target und den Fuzzer, kann also wie folgt ausgeführt werden: `./my_fuzzer CORPUS_DIRECTORY/`. Das Corpus-Verzeichnis sollte bereits einen existierenden Corpus von interessanten Testfällen enthalten, in dieses werden auch durch die hineinkompilierte Sanitizer Coverage Instrumentation[13] gefundene, als interessant bewertete Inputs als Erweiterungen des Corpus geschrieben. Weitere Optionen, insbesondere zur Kontrolle des Fuzzer-Verhaltens, werden ebenfalls beim Aufruf der Binary als Kommandozeilen-Optionen übergeben.

Neue Inputs werden dabei mittels (durch Coverage-Analyse geleitete) Mutationen des bisherigen Corpuses und, falls vom Tester angegeben, Dictionary-Regeln erzeugt. Hierbei ist es dem Nutzer auch möglich, eigene Mutatoren anzugeben um dem Fuzz-Target besser gerecht werden zu können als es generische Mutatoren können.
Da libFuzzer mit der primären Zielrichtug des Testens von APIs und einzelnen Library-Funktionen designet wurde, hat es beispielsweise zwar vollen Support für Multi-Threaded-Anwendungen, jedoch nur eingeschränkten Support für Multi-Process-Anwendungen.

## 2.5 american fuzzy lop

American fuzzy lop[3], kurz afl, ist ein 2014 von Michał Zalewski entwickelter moderner Fuzzer. Ebenso wie libFuzzer macht auch afl Gebrauch von einer Reihe von Sanitizern welche von gcc oder clang angeboten werden. afl benötigt jedoch, anders als libFuzzer, spezielle Versionen dieser Compiler (afl-gcc beziehungsweise afl-clang) und eine spezielle Fuzzing-Binary (afl-fuzz). Ein exemplarisches Fuzzing sähe mit afl wie folgt aus:

1. Schreibe ein Fuzz-Target. Ein Fuzz-Target für afl ist im Normalfall ein beliebiges Programm, welches einen Input entweder von `stdin` oder aus einer spezifizierten Datei liest und dann den zu testenden Code mit diesem Input ausführt.

2. Kompiliere dieses Fuzz-Target mit einem der afl-Compiler, beispielsweise `CC=/usr/bin/afl-gcc make fuzztarget`.

3. Führe den Fuzzer mit dem kompilierten Fuzz-Target aus: `afl-fuzz -i input_directory -o findings_directory -- ./fuzztarget`. Hierbei können weitere Optionen zum Verhalten des Fuzzers vor `--` und Optionen des Fuzztargets nach diesem angegeben werden.

Ebenso wie libFuzzer nutzt afl Code-Coverage-Analyse durch Binary-Instrumentation und erweitert den initial angegebenen Corpus um Inputs, welche zu einer Erweiterung der Code-Coverage führen. Neue Inputs werden dabei wieder aus den Corpus-Einträgen und explizit angegebenen Regeln zur Inputmutation und -generierung erzeugt.

Ein interessanter Ansatz, den afl zur Erhöhung der Performance einsetzt, ist der sogenannte Fork-Server[3]: Anstatt für jede Iteration des Fuzz-Targets ein eigenes `execve`, Linking und entsprechende Initialisierungen durchzuführen, wird das Fuzz-Target nur einmal gestartet, direkt nach dem Start eingefroren und mit copy-on-write als Iterationen des Fuzz-Targets in separate Prozesse geklont.
Auch wenn afl weniger spezialisiert auf eine bestimmte Art von zu testender Software ist als beispielsweise libFuzzer, so hat afl auch keinen vollen Support für alle möglichen Programmarten, beispielsweise besitzt afl nur eingeschränkten Netzwerk- und Multi-Process-Support.

# 3 Checkpoint/Restore

Mit Checkpoint/Restore, kurz C/R, manchmal auch Checkpoint/Restart, bezeichnet man das Speichern eines Prozesses oder einer Anwendung in einer Art und Weise, die eine spätere Wiederherstellung und Weiterausführung der Anwendung erlaubt. C/R-Systeme finden Anwendung in verschiedenen Bereichen:

- Im Debugging und in der Fault Recovery durch das Zurücksetzen eines Programmes auf einen vorherigen Zustand
- In der Beschleunigung oder Optimierung von Prozessen durch das Checkpoint-basierte Fortsetzen von Anwendungen anstelle von rechenzeitaufwändigen Komplettausführungen
- Im der Optimierung von Anwendungen durch Aussetzen der Ausführung von Programmen wenn diese gerade nicht benötigt werden
- In der Migration von Prozessen zwischen verschiedenen Maschinen

## 3.1 C/R Typen und Unterscheidungen

Für das Checkpointing/Restoring von Anwendungen gibt es nicht einen einzelnen Standard, sondern eine ganze Reihe von Modellen, die wir grob anhand von zwei Faktoren klassifizieren: Zum einen der Typ beziehungsweise die Implementierungsebene, zum anderen die Capabilities des Checkpoint/Restore-Systems, also welche Aspekte des Programms genau wiederhergestellt werden können und welche nicht.

### 3.1.1 Nach Typ

Bei der Klassifizierung nach Typ oder Implementierungsebene unterscheiden wir grob drei Kategorien:

- Das Checkpointen im Userspace, also aus dem zu checkpointenden Programm selbst
- Das Checkpointen auf Kernelebene beziehungsweise durch den Kernel
- Das Checkpointen von Virtuellen Maschinen beziehungsweise Containern

Im klassischen Userspace-Checkpointing findet der Checkpointing-Prozess größtenteils transparent gegenüber dem Kernel statt, das heißt das C/R-Tool nutzt aus

dem Userspace zugängliche Ressourcen wie `/proc/` oder Intercepts von Library- oder Syscalls zum Sammeln der notwendigen Checkpoint-Informationen und es nutzt ebenfalls nur ohne besondere Privilegien zugängige Ressourcen zur Wiederherstellung. Dieses Vorgehen führt in reinen Userspace-C/R-Systemen wie DMTCP[10] (Distributed MultiThreaded CheckPointing) unter Umständen zu einigen Fallstricken oder Problemen. So können zum Beispiel nicht alle Eigenschaften eines Programmes, beispielsweise die PID, nicht ohne weiteres korrekt wiederhergestellt werden. Die Intercepts benötigen alternative Libraries, die erstens spätestens zur Startzeit mit der Anwendung gelinkt werden müssen, zweitens häufig Beschränkungen darauf haben mit welchen Programmen (bspw. Sprachen) sie genutzt werden können und drittens unter Umständen die Funktionsweise des Programmes ändern könnten. Zudem sind Performanceeinbußen durch das Duplizieren von Kernelstrukturen im Userspace zu erwarten.

Diese C/R-Systeme nutzen den Kernel direkt, um alle nötigen Informationen während des Checkpointings zu sammeln. Sie benötigen also keine Libraries für das Sammeln dieser Informationen durch das Abfangen von Syscalls und ähnlichem und haben alle nötigen Privilegien, um alle Eigenschaften der Anwendung korrekt und vollständig wiederherzustellen. Allerdings muss der Kernel eben diese Funktionalitäten anbieten, damit Kernelebene-C/R-Tools eingesetzt werden können. Obwohl es wiederholt Versuche der Integration von C/R-Tools in den Linux Kernel gab, ist dies bisher nicht geschehen, das heißt es wird zur Nutzung ein gepatchter Kernel oder ein Kernel mit zusätzlichen Modulen benötigt.
Das Checkpointing mit diesen Kernelebenen-C/R-Tools folgt zumeist diesem Schema:

1. Einfrieren und Synchronisieren der Prozesse und Threads
2. Erfassen von globalen Informationen (das heißt Informationen, die nicht direkt Teil der zu checkpointenden Anwendung sind), zum Beispiel Namespaces und Container
3. Erfassen der Prozess- beziehungsweise Thread-Hierarchien und -Informationen wie zum Beispiel Parent/Child-Verhältnisse und PIDs
4. Erfassen des Status der einzelnen Prozesse und Threads, beispielsweise Signals, geöffnete Files und FDs
5. Beenden oder Weiterausführung der Anwendung, Schreiben der erfassten Informationen in ein Image

Dabei werden all diese Schritte natürlich nicht von der zu checkpointenden Anwendung selbst vorgenommen, sondern von einem dafür gespawnten externen Prozess. Das Wiederherstellen funktioniert generell in ähnlicher Weise:

1. Wiederherstellen der globalen Information, beispielsweise Container bei Multiprocess-Anwendungen oder des Prozesses bei Multithreaded- aber Singleprocess-Anwedungen, falls nötig
2. Erstellen einer Prozess- beziehungsweise Threadhierarchie entsprechend der erfassten Hierarchie in einem eingefrorenem Zustand

3. Wiederherstellen des State der einzelnen Prozesse und Threads

4. Fortsetzen beziehungsweise Auftauen der Prozesse/Anwendung

CRIU[1] (Checkpoint/Restore in Userspace) ist ein Hybrid aus Userspace- und Kernelebene-Systemen, das heißt, es agiert größtenteils im Userspace, nutzt jedoch Calls auf privilegierte Operationen wie das Forken mit bestimmten PIDs und Kernelfeatures wie ptrace. All diese von CRIU benötigten Kernel-Capabilities sind seit einiger Zeit im Linux Mainline-Kernel vorhanden. Das Checkpointing und Restoring in CRIU ähnelt stark dem der Kernelebene-C/R-Systeme, es benötigt insbesondere keine Intercept-Libraries wie einige Userspace-C/R-Tools. Jedoch können nicht alle benötigten Informationen durch externe Programme erfasst und wieder hergestellt werden. Dies umgeht CRIU indem es mittels ptrace sogenannten Parasite Code in die zu checkpointenden Prozesse einfügt (beziehungsweise den Restorer Blob bei der Wiederherstellung), der eben diese Informationen erfassen kann.

Das Checkpointen von Virtuellen Maschinen oder Containern nutzt zumeist einen der obigen Ansätze um die laufende Virtuelle Maschine (als Anwendung auf dem Host-System) als Ganzes zu checkpointen. So nutzt Docker beispielsweise CRIU, verwaltet durch entsprechende Aufrufe des docker Kontrollprogramms `docker checkpoint create [...]` und `docker start --checkpoint [...]`.

### 3.1.2 Nach Capabilities

Zusätzlich unterscheiden sich C/R-Tools auch danach, welche Capabilities sie Checkpointen und Restoren können. Einige Tools wie frühere BLCR-Versionen unterstützen nur einzelne multi- oder singlethreaded Prozesse, Linux-CR unterstützt dagegen das Checkpointen beliebiger Container oder Subtrees von Prozessen. Tools, die innerhalb von multithreaded Prozessen arbeiten, also einzelne Threads checkpointen und wiederherstellen können, sind uns nicht bekannt. Dies ist auch plausibel, da Threads sich eben Prozess-Ressourcen teilen und daher ein Wiederherstellen nur einiger Threads eines Prozesses zu undefiniertem Verhalten führen würde.
Weitere Unterschiede gibt es in der Frage, welche Eigenschaften von laufenden Programmen wiederhergestellt werden können. Dies sind beispielsweise Netzwerkverbindungen wie TCP- oder UDP-Sockets, Dateien und FDs oder Namespaces. C/R-Programme unterscheiden sich auch bezüglich der Frage, ob beliebige Programme gecheckpointed werden können und ob bereits zum Start der Anwendung bekannt sein muss, dass diese gecheckpointed werden soll. Im letztren Fall müssen unter Umständen entsprechende Vorbereitungen wie das Preloading von Libraries oder das Informieren des Kernels ob des bevorstehenden Checkpointings getroffen werden.

# 4 C/R-Fuzzing: Ansätze

Wie bereits erwähnt bestehen im klassischen Fuzzing einige Probleme: Komplexe Systeme oder großer Overhead im Set-Up des Fuzz-Targets führen zur Reduktion des Durchsatzes und damit zu Einschränkungen der Effizienz, mehrstufige interaktive Programme wie beispielsweise Netzwerkprotokolle sind als ganzes

nur mit großem Aufwand oder gar nicht vernünftig zu fuzzen. In dieser Arbeit wollen wir untersuchen, ob sich einige oder sogar alle diese Probleme durch die Nutzung von Checkpoint/Restore-Mechanismen beheben oder zumindest in ihren Auswirkungen reduzieren lassen. Dazu betrachten wir zunächst generell mögliche Ansätze und ihre theoretischen Vor- und Nachteile, bevor wir die Implementierbarkeit mit verfügbaren Fuzzern und Checkpoint/Restore-Systemen betrachten.
Generell sehen wir zwei Ansätze zur Integration von Checkpoint/Restore-Tools und Fuzzern:

1. Integration des C/R-Tools in den Fuzzer selbst, das heißt der Fuzzer selbst verwaltet das Checkpointing und Restoring

2. Implementierung des Checkpointing und Restoring innerhalb des Fuzztargets, transparent zum Fuzzer

Die erste Alternative, ein C/R-Fuzzer, wäre eine umfassendere Lösung als die Ad-hoc-Lösung der Implementierung im Fuzztarget und bietet eine größere Palette an Möglichkeiten als die Fuzztarget-Lösung, benötigt allerdings eben einen C/R-Fuzzer: Etwas, das nach unseren Recherchen weder in Software existiert noch in Konzepten diskutiert wurde. Ein allererster Ansatz, wenn auch nicht als C/R bezeichnet und unter anderen Aspekten erdacht ist das Fork-Server-Konzept in afl[19], siehe Kapitel 2.5.

## 4.1 C/R-Fuzzer

Klassische moderne Fuzzer wie beispielsweise libfuzzer folgen während des Fuzzings im Allgemeinen diesem Ablauf:

Listing 1: Struktur klassischer Non-C/R-Fuzzer

```
1 Bis ein Problem auftritt:
2     Generiere einen neuen Input (aus dem Seed, Corpus, Dictionary
          Regeln)
3     Führe das Fuzz-Target mit diesem Input aus
4     Analysiere das Fuzz-Target während der Ausführung und update
          den Corpus wenn nötig
```

Das Fuzz-Target wird also mit verschiedenen Inputs ausgeführt, bis ein Problem auftritt. Dabei kann das Fuzz-Target im Normalfall strukturell wie in Listing 2 beschrieben in drei Blöcke zerlegt werden.

Listing 2: Struktur klassischen Fuzztargets

```
1 Inputunabhängiges Set-Up
2 Inputabhängiges Set-Up
3 Zu testender Code oder Funktion
```

Hierbei ist das Ergebnis von Zeile 1 im Allgemeinen entweder konstant oder zumindest austauschbar (wenn es beispielsweise externe (pseudo-)zufällige Einflüsse wie Nutzung der Systemzeit oder Aufrufe von Zufallszahlgeneratoren gibt). Für die Betrachtung in dieser Arbeit könen wir das Fuzz-Target nun wie folgt aufteilen:

(a) Inputunabhängiges Set-Up (Zeile 1 in Listing 2)

(b) Inputabhängiges Set-Up und zu testende Funktionalität (Zeilen 2 und 3 in Listing 2)

Hierauf aufbauend würde ein C/R-Fuzzer als Alternative zum klassischen Fuzzer dem Aufbau in Listing 3 folgen.

Listing 3: Struktur einfacher C/R-Fuzzer

```
1 Führe den Inputunabhängigen Teil des Fuzz-Targets aus und
      checkpointe ihn
2 Bis ein Problem auftritt:
3     Generiere einen neuen Input (aus dem Seed, Corpus, Dictionary
          Regeln)
4     Restore das Fuzz-Target, übergebe diesen Input und führe den
          Inputabhängigen Teil des Fuzz-Targets aus
5     Analysiere den Inputabhängigen Teil des Fuzz-Targets während
          der Ausführung und update den Corpus wenn nötig
```

Hierbei ist folgendes zu beachten: Die Übergabe des Inputs muss während der Laufzeit (beispielsweise über `stdin`) erfolgen, eine Übergabe des Inputs durch Parameter zum Zeitpunkt des Aufrufes wie beispielsweise bei libfuzzer ist nicht ohne weiteres möglich. Das Aufteilen des Fuzz-Targets bedeutet nicht das Aufteilen in mehrere unabhängige Code-Segmente - damit Checkpoint/Restore ohne Modifikationen funktionieren kann, muss der inputabhängige Teil bereits von Anfang an im Fuzz-Target vorhanden sein. Ein solches Fuzz-Target hätte damit eine Struktur wie in Listing 4 beschrieben.

Listing 4: Struktur C/R-Fuzzer-Fuzztarget

```
1 Inputunabhängiges Setup
2 Breakpoint
3 Inputabhängiges Setup
4 Zu testender Code oder Funktion
```

Hierbei hält der Breakpoint die Ausführung des Fuzztargets an und informiert den Fuzzer darüber, dass er an dieser Stelle das Fuzz-Target checkpointen soll. Ziel eines solchen simplen C/R-Fuzzers ist es, den Overhead durch wiederholte Ausführung des Setups zu reduzieren. Im Folgenden bezeichnen wir mit $T_{Setup}$ die Laufzeit des inputunabhängigen Setups, mit $T_{Execution}$ die (mittlere) Laufzeit des inputabhängigen Setups und des zu testenden Codes, sowie mit $T_{Checkpoint}$ und $T_{Restore}$ die benötigte Zeit für eine Checkpointing- beziehungsweise Restoreoperation. Für die die Laufzeit eines normalen, nicht-C/R Fuzzers bei $n$ benötigten Operationen ergibt sich

$$T_{Non-C/R}(n) = n(T_{Setup} + T_{Execution}) \tag{1}$$

und für einen C/R-Fuzzer wie oben beschrieben

$$T_{C/R}(n) = T_{Setup} + T_{Checkpoint} + n(T_{Restore} + T_{Execute}) \tag{2}$$

Für große $n$ und $T_{Restore} < T_{Setup}$ ergibt sich hier nun also eine Zeitersparnis pro Iteration:

$$\begin{aligned} \frac{T_{Non-C/R}(n) - T_{C/R}(n)}{n} &= \frac{(n-1)T_{Setup} - T_{Checkpoint} - nT_{Restore}}{n} \\ &=_{n\to\infty} T_{Setup} - T_{Restore} \end{aligned} \tag{3}$$

$T_{Setup}$ ist dabei natürlich vom jeweilig zu testenden Programm abhängig, $T_{Restore}$ von mehreren Faktoren, unter anderem dem Programm selbst, insbesondere der Größe im Speicher und den zu checkpointenden Features wie Netzwerksockets, und natürlich dem genutzten Checkpoint/Restore-Tool. Eine genauere Betrachtung der Einflüsse dieser Faktoren findet in Kapitel 5.5 statt.

## 4.2 Non-C/R-Fuzzer mit C/R-Ansätzen

Ein anderer Ansatz zur Kombination von Fuzzing mit Checkpoint/Restore-Mechanismen ist es, statt des impliziten Checkpointing durch den Fuzzer eplizites Checkpointing und Restoring im Fuzztarget vorzunehmen. Dies hat den Vorteil einer größeren Flexibilität im Einsatz und benötigt im Idealfall keine Änderungen am Fuzzer. Dieser Ansatz könnte also unter Umständen mit bereits verfügbaren Fuzzern genutzt werden.[2]
Wir betrachten hier zwei Varianten dieses Ansatzes: zuerst in Kapitel 4.2.1 eine generische, naive Nutzung von Checkpoint/Restore im Fuzztarget eines Non-C/R-Fuzzers mit ähnlichem Ziel und einem ähnlichen Ansatz zu dem oben beschriebenen C/R-Fuzzer.
Kapitel 4.2.2 behandelt eine problemspezifische Architektur, die durch Anwendung von Checkpoint/Restore-Methoden versucht, das effiziente Fuzzen von Netzwerkprotokollen als Beispiel für komplexe mehrstufige Programme durch Exploration des Zustandsgraphen zu ermöglichen.

### 4.2.1 Naiver Non-C/R-Fuzzer

Eine konzeptuell recht einfacher Ansatz ist das Checkpointing nach der Ausführung des unabhängigen Setups, ähnlich wie bei dem oben beschriebenen C/R-Fuzzer. Dies ermöglich in späteren Ausführungen des Fuzz-Targets ein Aufsetzen an dieser Stelle. Anders als bei einem C/R-Fuzzer nach Kapitel 4.1 (bei dem diese Information durch den Fuzzer verwaltet wird) muss hier jedoch im Fuzz-Target bestimmt werden, ob es sich in der ersten Iteration oder einer späteren Iteration befindet. In der ersten Iteration muss das Setup einmalig ausgeführt und danach gecheckpointed werden; in späteren Iterationen wird das Setup nicht erneut ausgeführt, sondern der Checkpoint wiederhergestellt und die Ausführung, beginnend direkt mit dem zu testenden Code, fortgesetzt. Dies kann beispielsweise durch das Setzen von Systemvariablen oder das Erstellen eines Tokens im Dateisystem geschehen - ist die Variable bei einer Ausführung also gesetzt oder existiert der Token, so befindet sich das Fuzz-Target in einer späteren Iteration und überspringt das Setup durch ein Restore. Ein solches Fuzz-Target folgt der Struktur aus Listing 5.[3]

Listing 5: Struktur C/R-Fuzztarget für Non-C/R-Fuzzer

```
1 Falls der Token existiert:
2     Restore
3 Sonst:
4     Führe Setup aus
5     Setze Token
6     Checkpointe das Programm und setze die Ausführung fort
```

[2] Eine Diskussion dieser Umstände und der tatsächlichen Anwendbarkeit findet in Kapitel 6 statt.

[3] In Kapitel 6 stellen wir prototypische Umsetzungen mit CRIU und afl/libFuzzer dar.

```
7       Hole dir den Input
8       Führe den Test mit diesem Input aus
```

Wenn in Zeile 2 restoret wird, so wird die alte, gecheckpointete Instanz wiederhergestellt und setzt ihre Ausführung in Zeile 7 fort, holt sich also neuen Input und testet mit diesem.
Hierbei sind folgende Punkte zu beachten:

- Wie auch beim C/R-Fuzzer ist das Setup in einen inputunabhängigen und einen inputabhängigen Teil aufzubrechen. Dabei wird nur der inputunabhängige Teil vor dem Checkpoint ausgeführt, der inputabhängige Teil wird zusammen mit dem zu testenden Code in jeder Iteration in Zeile 7 ausgeführt.

- Da mit dem Restore in Zeile 2 ein neuer Prozess entsteht, muss der genutzte Fuzzer Multiprocess-Anwendungen behandeln können und über die Erstellung des (eigentlich zu testenden) restoreten Prozesses informiert werden. Inbesondere muss unerwartetes beziehungsweise unerwünschtes Verhalten im restoreten Prozess erkannt werden können. Dies kann beispielsweise durch eine restore-as-child Funktionalität vom C/R-Tool realisiert werden[4], benötigt allerdings einen Fuzzer mit Multiprocess-Fähigkeiten.

- Eine speziell zu betrachtende Multiprocess-Fähigkeit der Fuzzing-Engine ist die Messung der Code-Coverage bei White- und Grey-Box-Fuzzern. Aus Sicht der Fuzzing-Engine wird der restorenden Prozess als Fuzz-Target wahrgenommen, nicht der eigentlich zu testende Prozess. Hier wäre also speziell eine Umlenkung auf den gespawnten Prozess ein notwendiges Feature einer Fuzzing-Engine für die korrekte Messung der Code-Coverage.

- Fuzzing setzt im Allgemeinen parallele Ausführungen des selben Fuzz-Targets zur Effizienzsteigerung ein. Dies ist im beschriebenen Ansatz nicht ohne weiteres möglich, da mehrere, gleichzeitige Restores der selben Instanz zu konkurriendem Zugriff auf Ressourcen wie PID, Input und Netzwerkressourcen führen würden. Dies würde typischerweise ein erfolgreiches und korrektes Restoring verhindern. Ohne Änderung der Fuzzing-Engine könnte eine Parallelisierung nur erfolgen, indem die Fuzzing-Engine mehrfach instanziert wird und das Fuzz-Target hierbei für jede Instanz andere Werte für Token, den Imagepfad, etc. kennt.

- Der Mechanismus zur Übergabe des Inputs an das Fuzz-Target muss bestimmten Anforderungen entsprechen (dies kann je nach Fuzzer ein Problem darstellen, da diese zumeist den Input entweder beim Aufruf des Fuzztargets als Argument oder durch einen FD wie `stdin` übergeben - beides Ressourcen, auf die der wiederhergestellte, eigentlich zu testende Prozess keinen Zugriff hat). Hierfür existieren zwei Strategien:

  - Der restorete Prozess holt sich den Input direkt von der Fuzzing-Engine, beispielsweise indem die Fuzzing-Engine die Inputs in einer FIFO bereitstellt. Jeder restorete Prozess, in jeder Iteration, entnimmt seinen Input und führt mit ihm den Test aus.

[4] Zum Beispiel von CRIU unterstützt.

– Der restorende Prozess speichert vor dem Restore den erhaltenen Input in einer Art und Weise ab, dass der restorete Prozess diesen Input auslesen kann, beispielsweise durch eine Zwischenspeicherung in einer Datei. Eine Adaption von Listing 5 an diese Strategie ist in Listing 6 dargestellt.

Listing 6: Struktur C/R-Fuzztarget für Non-C/R-Fuzzer mit Zwischenspeicherung des Inputs

```
1  Hole den Input von Fuzzing-Engine
2  Schreibe den Input in eine Datei
3  Falls der Token existiert:
4      Restore
5  Sonst:
6      Führe Setup aus
7      Setze Token
8      Checkpointe das Programm
9      Hole den Input aus der Datei
10     Führe den Test mit diesem Input aus
```

Ebenso wie der oben beschriebene C/R-Fuzzer versucht dieser Ansatz nur das Setup-Overhead-Problem im Fuzzing zu lösen - mit ähnlichen Laufzeitverbesserungen. Wie auch beim C/R-Fuzzer gilt hier

$$T_{Non-C/R} = n(T_{Setup} + T_{Execution}) \tag{4}$$

$T_{C/R}$ ist etwas komplizierter, da das Fuzztarget hier einige der C/R-Fuzzer-Funktionalitäten ad-hoc implementieren muss, es ergibt sich zunächst:

$$\begin{aligned} T_{C/R}(n) &= 1(T_{Tokencheck} + T_{Setup} + T_{Tokenset} + T_{Checkpoint} + T_{Input} + T_{Execute}) \\ &\quad + (n-1)(T_{Tokencheck} + T_{Restore} + T_{Input} + T_{Execute}) \\ &\overset{(a)}{\simeq} T_{Setup} + T_{Checkpoint} + T_{Execute} + (n-1)(T_{Restore} + T_{Execute}) \\ &= T_{Setup} + T_{Checkpoint} + (n-1)T_{Restore} + nT_{Execute} \end{aligned} \tag{5}$$

Die Vereinfachung in $(a)$ setzt voraus, dass die diversen Token- und Inputoperationen verglichen mit den recht langsamen Restore- und Setup-Operationen für die letztendliche Laufzeit nicht ins Gewicht fallen.
Das $T_{C/R}$ in diesem Fall unterscheidet sich nicht relevant vom $T_{C/R}$ des C/R-Fuzzers in Gleichung (2), der Unterschied um $1T_{Restore}$ ergibt sich daraus, dass wir im C/R-Fuzzer nach dem ersten Setup nicht direkt weiter ausführen, sondern den Prozess beenden und dann später für die erste Ausführung wieder Restoren. Natürlich ließe sich im C/R-Fuzzer-Fall auch diese Optimierung vornehmen.
Auch hier ergibt sich, analog zu Gleichung (3) die Laufzeitverbesserung je Iteration als

$$\begin{aligned} \frac{T_{Non-C/R}(n) - T_{C/R}(n)}{n} &= \frac{(n-1)T_{Setup} - T_{Checkpoint} - (n-1)T_{Restore}}{n} \\ &=_{n\to\infty} T_{Setup} - T_{Restore} \end{aligned} \tag{6}$$

### 4.2.2 Exploration des Zustandsgraphen in Client/Server-Systemen

Die bisher beschriebenen Ansätze bieten nur sehr naive Ansätze zur Lösung nur eines der beschriebenen Fuzzingprobleme - Reduktion des Laufzeitover-

heads durch wiederholte Ausführung des Setups in einstufigen, simplen Anwendungen. In diesem Kapitel beschreiben wir einen Ansatz zum Testen einer der Seiten in komplexen, insbesondere mehrstufigen, interaktiven Client-Server-Anwendungen. Hierbei wird durch die Fuzzing-Inputs Schritt für Schritt ein Zustandsgraph beispielsweise des Servers aufgebaut, parallel dazu werden bereits diese Zustände jeweils mit den Inputs auf problematisches, unerwünschtes Verhalten getestet. Zur besseren Lesbarkeit werden wir in diesem Kapitel davon ausgehen, dass ein Server gefuzzt werden soll und das Fuzztarget die Client-Aufgaben übernimmt, dies lässt sich natürlich auch umkehren.
Das Fuzztarget übernimmt dabei in diesem Ansatz mehrere Aufgaben:

- Client-Server-Interaktion: Das Fuzztarget implementiert nicht einen wirklichen Client, der das erwünschte Protokoll spricht, sondern simuliert nur einen solchen Client, indem es die von der Fuzzing-Engine generierten Inputs als Client-Input an den Server übergibt, beispielsweise über eine bereits etablierte TCP-Verbindung zu dem Server.

- Verwaltung des Servers: Das Fuzztarget hält in persistenter Form, beispielsweise durch Schreiben ins Dateisystem, den bisher gefundenen Zustandsgraphen des Servers vor, also die gefundenen Zustände und die Inputs, die zu Übergängen zwischen diesen führen. Zusätzlich wird jeder Zustand, wenn er das erste Mal erreicht wird, gecheckpointet und das entsprechende Image mit dem Zustand verknüpft, das heißt das Fuzz-Target muss auch erkennen wenn einer neuer Zustand erreicht wird.

Neu generierte Inputs werden dabei hintereinander auf alle bekannten Zustände angewendet, der Zustandsgraph wird um neu gefundene Kanten und neue Zustände erweitert.
Listing 7 implementiert ein solches Fuzz-Target in Pseudocode.

Listing 7: Fuzztarget zur Exploration des Zustandsgraphen

```
1  if token not set:
2      server.start()
3      connection := connectTo(server)
4      c := checkpoint(connection)
5      state_id := server.state
6      s := cr.checkpoint(server, dump=true)
7      transition := [] // Liste der Übergänge zu anderen Zuständen
8      states := {} // Dictionary der Zustände
9      states[state_id] = {s: s, c: c, t: transition, id: state_id}
10     set token
11
12 input := fuzzer.input
13
14 for old_state in states:
15     server := cr.restore(old_state[s])
16     connection := restore(old_state[c])
17
18     connection.send(input)
19     if server.state not in states.keys:
20         new_state_c := checkpoint(connection)
21         new_state_id := server.state
22         new_state_s := cr.checkpoint(server, dump=true)
23         new_state_t := []
24         states[old_state][t].append((input, new_state_id))
```

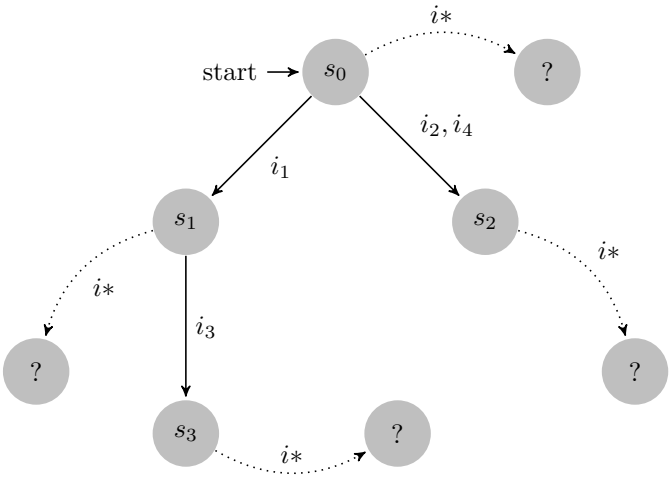


Abbildung 1: Zustandsübergangsexploration mit Input $i*$

```
25          states[new_state_id] := {s: new_state_s, c: new_state_c, t:
                new_state_t, id: new_state_id}
26      else if old_state.id ≠ server.state:
27          states[old_state][t].append(input, server.state)
```

Hierbei kann es sich bei `connection` um eine beliebige Verbindungsart zwischen dem Client und dem Server handeln, beispielsweise eine TCP-Verbindung oder UNIX-Sockets, solange das Wiederherstellen dieser Verbindung vom C/R-Tool unterstützt wird. Das clientseitige Checkpointing der Verbindung in Zeile 4 und das entsprechende Wiederherstellen wird dabei nicht durch das normale C/R-Tool durchgeführt, sondern durch das Speichern der Verbindungsinformationen. Für eine TCP-Verbindung kann dies beispielsweise durch einen Userspace TCP/IP-Stack geschehen oder durch das Nutzen der `TCP_REPAIR` Option in modernen Linux-Kerneln (seit 3.5).
Zusätzlich muss sich der Server darüber bewusst sein, in welchem Zustand er sich gerade befindet, und dies dem Fuzz-Target/Client mitteilen können (siehe: `state_id := server.state` etc.), beispielsweise durch Einfügen entsprechender Funktionalität in den Servercode und die Übergabe durch einen entsprechenden IPC-Ansatz.
Abbildung 1 zeigt die Zustandsübergangsexploration: Es liegt ein Server vor mit bereits bekannten Zuständen $s_0, s_1, s_2, s_3$ und den Zustandsübergängen für die Inputs $i_1, \ldots, i_4$. Für einen neuen Input $i*$ wird nun jeder der Zustände $s_0, \ldots, s_3$ einmal wiederhergestellt und mit $i*$ getestet. Falls Zustand $s_i$ mit Input $i*$ einen neuen Zustand erzeugt, wird dieser gecheckpointed und als Knoten $s_4$ mit der Kante $(s_i, s_4, i*)$ dem Graphen hinzugefügt.

Bei Fragestellungen, bei denen eine sinnvolle Definition des States möglich ist, ergibt dieser Ansatz zwei Vorteile gegenüber klassischem Fuzzing:

- Ein schnelleres Restore des Servers im Zustand $s_i$ gegenüber dem Setup und Versetzen des Servers in $s_i$ durch dafür benötigte Inputs.
- Der Tester muss nicht für jeden Zustand $s_i$ ein Fuzz-Target definieren, sondern das generische Fuzz-Target erzeugt dies durch den Checkpoint

des Servers im Zustand $s_i$.

Nachteile dieses Ansatzes sind:

- White-Box-Fuzzing beziehungsweise klassisches Gray-Box-Fuzzing durch Code-Coverage werden, wie schon Ansätzen in Kapitel 4.1 und 4.2.1, durch die permanente Variation des tatsächlich ausgeführten Codes erschwert beziehungsweise könnten in bisherigen Fuzzern tatsächlich gar nicht nutzbar sein, da der eigentlich ausgeführte Code, der Client, gar nicht der Code ist, den wir Fuzzen wollen - dies wäre der Server. Stattdessen stellt diese Art des Fuzzings jedoch eine eigene Art des Gray-Box-Fuzzings dar, da das Fuzztargets nach und nach die Struktur des Servers, wenn auch auf einem höheren Level als klassische Code-Coverage, herausarbeitet und darüber versucht, das Fuzzing zu optimieren.

- Selbst wenn Code-Coverage-Messungen möglich sind, so ist ihre Effektivität doch dadurch abgeschwächt, dass diese Inputs auf alle Zustände angewendet werden, nicht nur auf diejenigen Zustände, bei denen sie ob der Code-Coverage ausgewählt wurden.

- Ebenso wie bei der naiven C/R-Implemetierung im Fuzztarget muss der Fuzzer auch hier mit Multiprocess-Anwendung umgehen können, insbesondere den Server - als vom Fuzztarget gestartetes, eigenständiges Programm - in die Suche nach problematischem Verhalten einbeziehen.

- Das Formulieren von Regeln, Dictionaries, ... für die Inputs eines Smart-Fuzzers wird dadurch verkompliziert, dass diese Regeln nun nicht nur für einen Pfad oder eine bestimmte Funktion gelten sollen, sondern für das ganze zu testende Programm. Dies kann unter Umständen zu einer Vergrößerung des Suchraums oder sogar der nicht-Nutzbarkeit von Smart-Fuzzern führen, was wiederum Performanceverluste bedeutet.

- In Fällen, in denen es nur wenige gültige Übergänge auf einen Zustand mit hoher Distanz vom Ausgangsknoten gibt, kann es passieren, dass die Inputs bereits untersucht wurden oder als wenig vielversprechend von der Fuzzing-Engine verworfen wurden. Um dies zu vermeiden, sollte die Optimierung der Inputs, wenn überhaupt möglich, nicht zu zielgerichtet erfolgen.

# 5 CRIU

Für den Rest der Arbeit werden wir CRIU - Checkpoint/Restore in Userspace - näher betrachten und CRIUs Nutzbarkeit für Fuzzing, insbesondere in Kombination mit verfügbaren Fuzzing-Engines, untersuchen. CRIU ist ein Open Source C/R-Tool, welches ursprünglich als Ersatz für das OpenVZ in-kernel Checkpoint/Restore-System durch ein Team bei Virtuozzo ab 2011 entwickelt wurde[18]. Unsere Wahl von CRIU als C/R-Tool für diese Untersuchung erfolgte aus folgenden Gründen:

- Verfügbarkeit & Aktualität: CRIU ist ein vergleichsweise modernes, gut dokumentiertes Open Source C/R-Tool, das geringe Anforderungen an das System stellt - es benötigt keine Kernelpatches oder -module außerhalb des

Linux Mainline Kernels - und ist daher auf einer Vielzahl von Systemen verfügbar. Zusätzlich wird CRIU von den Entwicklern noch unterstützt und weiterentwickelt, im Gegensatz zu einigen anderen vielversprechenden C/R-Tools wie Linux-C/R.

- Unveränderte Binary: Anders als einige andere Userspace-C/R-Tools nimmt CRIU keine Veränderungen an der Binary vor und benötigt keine speziellen Schritte bei der Kompilierung - auch werden keine besondere Schritte beim Anwendungsstart benötigt, erst zum Zeitpunkt des Checkpointens wird CRIU aktiv.
- Capabilities: CRIU bietet eine breite Palette an Capabilities, die mit CRIU wiederhergestellt werden können, insbesondere TCP- und andere Netzwerkverbindungen.[5]

CRIU findet unter anderem Anwendung in einigen prominenten Container-Anwendungen wie Docker, LXC/LXD und OpenVZ, anstatt des Legacy OpenVZ C/R-Systems, zur Implementierung des Container-Checkpointings.

## 5.1 Nutzung

CRIU kann auf zwei Arten genutzten werden:

- Über CRIU-spezifische RPC-Aufrufe (Remote Procedure Call). Dies erfolgt entweder durch das CLI oder eine Programmiersprachen-API, insbesondere die C-API.
- Über das Command Line Interface ohne RPC beziehungsweise ohne CRIU Service.

Unabhängig davon, welche der beiden Optionen man nutzt, folgt das Checkpointing und Restoring in etwa den gleichen Schritten:

- Falls RPC genutzt werden soll, muss der CRIU-Service bereits aktiv sein (beispielsweise zu starten durch `criu service`), falls die C-API genutzt wird, muss diese mittels `criu_init_opts()` initialisiert werden.
- Man bestimmt, welchen laufenden Prozess man checkpointen möchte, meist über die `PID` durch Nutzung der `--tree $PID` Option beziehungsweise `criu_set_pid(pid)`. Falls sich ein Prozess über die C-API selbst checkpointen möchte, muss `criu_set_pid` nicht explizit aufgerufen werden. Zusätzlich wird angegeben, ob der Prozess nach dem Checkpointing fortgesetzt werden oder beendet werden soll.
- Man gibt einen Pfad oder einen FD für das Image-Directory an, entweder über die `--images-dir $IMGDIR` Option oder `criu_set_images_dir_fd(fd)`, bei der Nutzung der C-API muss `fd` ein bereits geöffneter FD auf das Verzeichnis sein, in das das Image geschrieben werden soll.

[5] Für eine vollständige Liste von unterstützten Features und einem Vergleich zu anderen C/R-Systemen siehe `https://criu.org/Comparison_to_other_CR_projects`.

- Spezifizierung weiterer Zusatzinformationen, beispielsweise `--shell-job` beziehungsweise `criu_set_shell_job(true)`, `--tcp-established` beziehungsweise `criu_set_tcp_established(true)`.
- Modifikation der Socket-Location des Services, des Log-Levels, Log-Files, ... falls nötig oder erwünscht

Diese Optionen werden dann mit der entsprechenden Operation, beispielsweise Checkpointing (`criu dump`, `criu_dump()`) oder Restoring (`criu restore`, `criu_restore()`), kombiniert, diese wird von CRIU ausgeführt. Wird ein Shell-Job restored, so wird dieser direkt an das laufende Terminal attached.

## 5.2 Funktionsweise

Wie bereits in der Übersicht verschiedener C/R-Systeme erwähnt ist CRIU ein Userspace-C/R-Tool, welches jedoch dabei Kernel-Features wie beispielsweise `prctl` aus dem `CONFIG_CHECKPOINT_RESTORE` Kernel-Feature oder die `TCP_REPAIR`-Flag[14] verwendet, um Anwendungen möglichst vollständig wiederherstellen zu können, ohne dabei eine komplette Kopie aller Kernelinformationen im Userspace aufbauen und permanent erhalten zu müssen, wie dies beispielsweise bei DMTCP der Fall ist. Dies bedeutet insbesondere auch, dass die zu checkpointende Anwendung weder zur Kompilierzeit noch zum Zeitpunkt des Anwendungsstartes sich des zukünftigen Checkpointings bewusst sein muss. Die von CRIU genutzten Kernel-Features sind im Mainline-Linux-Kernel enthalten und es sind alle nötigen Features aktiviert, insbesondere:

- `CONFIG_CHECKPOINT_RESTORE`, ein Feature, das eine Reihe von für C/R relevante Tools und Informationen enabled, beispielsweise `prctl` zur Manipulation von Prozessen oder eine Erweiterung der in `/proc/` verfügbaren Informationen, beispielsweise um die `/proc/<pid>/map_files`.
- `CONFIG_NAMESPACES` sowie Features für diverse spezielle Namespaces, namentlich UTS, IPC, PID und Network Namespaces.
- Weitere Features zum Monitoring verschiedener Socket-Arten.

Das Checkpointing erfolgt in folgenden Schritten:

1. Einfrieren des Prozessbaumes. Dies geschieht auf eine von zwei Arten: Entweder, falls möglich, durch die Nutzung einer freezer cgroup, oder ansonsten durch das Traversieren des PID-Baumes in `/proc/` und Anhalten der Prozesse durch die Nutzung der `PTRACE_SEIZE` Anweisung.
2. Erfassen von Informationen wie den virtuellen Speicherbereichen, Registern, ... durch das Lesen entsprechender `/proc/` Einträge und die Nutzung von ptrace.
3. Injektion eines Parasiten-Code-Blobs in die Anwendung mittels ptrace. Dieser Parasit kann nun die fehlenden Prozessinformationen extrahieren, hierfür wird die Anwendung kurzfristig wieder aufgetaut.
4. Entfernung des Parasiten mittels ptrace, Schreiben des Images und Fortsetzen der Ausführung der Anwendung.

Der Parasit ist hierbei eine PIE (Position-independent executable), welche in zwei Schritten in die Anwendung injiziert wird: Zuerst wird ein Stück Code mittels ptrace so in die Anwendung injiziert, dass dieses nach dem Auftauen der Anwendung ausgeführt wird und somit Shared Memory für den eigentlichen Parasiten zur Verfügung stellen kann. Dann wird die Anwendung kurzfristig aufgetaut, das Shared Memory zur Verfügung gestellt und die Ausführung des Parasiten angestoßen. Der Parasit wartet auf Anweisung des CRIU-Prozesses, zumeist zum Erfassen von Informationen wie Memory-Inhalten und Credentials, und führt diese aus. Zuletzt sendet der CRIU-Prozess noch eine `PARASITE_CMD_FINI` Anweisung an den Parasiten, dieser terminiert sich selbst und wird anschließend vom CRIU-Prozess durch das Unmappen des Shared Memories entfernt.
Das Restoring weicht etwas vom in Kapitel 3.1.1 beschriebenen Muster ab:

1. Lesen des Images und Zuweisung von Shared Ressources zu Prozessen, so dass diese nur einmal wiederhergestellt werden.

2. Forken der CRIU-Anwendung, um einen Prozessbaum entsprechend dem der Anwendung zu erzeugen.

3. Wiederherstellen eines Großteils der Prozessinformationen, insbesondere Speicherinhalte (jedoch nicht die exakten Mappings), Sockets, Namespaces, ...

4. Ersetzen der Forks durch einen weiteren Code-Blob, den Restorer Context. Wiederherstellen aller verbleibender Informationen, insbesondere exakte Memory Mappings, Credentials, Operationen, die von Credentials abhängen wie fork-with-pid und Threads durch den Restorer-Context, Unmapping des Restorer-Contexts und Fortsetzen der planmäßigen Ausführung der Anwendung.

Der Restorer-Blob ist, ebenso wie der Parasit, ein PIE-kompiliertes Stück Code, welches kurzfristig in Shared Memory im Anwendungsprozess lebt, um dort Aufgaben der CRIU-Anwendung zu übernehmen. Dabei agiert der Restorer-Context als Verbindungsstück zwischen dem CRIU-Fork und dem eigentlichen Anwendungsprozess. Er reorganisiert beispielsweise die bereits in Schritt 3 gefüllten Virtual Memory Areas, so dass die Mappings mit der Anwendung übereinstimmen, und stellt die korrekten PIDs wieder her.

Die beim Checkpointing entstehenden CRIU Images bestehen aus mehreren Dateien, organisiert in dem beim Checkpointing spezifizierten Ordner. Die Dateien fallen in drei Kategorien:

- Dateien im Protobuff-Format, welche einen Großteil der eigentlichen Checkpoint-Informationen enthalten, wie zum Beispiel generelle Checkpointing Informationen in `inventory.img` oder Informationen über die Credentials in `creds.img`

- CRIU-spezifische Binary-Images, welche beispielsweise die Memory-Inhalte und Pagemaps enthalten

- Dateien in einem CRIU-unspezifischen Format die allgemeine Systeminformationen wie Routing- und Netzwerkinformationen enthalten.

## 5.3 Fuzzing-Vorbereitungen

Da CRIU nicht für eine Anwendung in Fuzzing- oder ähnlichen Hoch-Frequenz-Restoring-Kontexten entwickelt wurde, bestehen bei der Nutzung von CRIU in diesen Kontexten zunächst einige Probleme:

1. Das mehrfache Wiederherstellen eines Anwendungspaars mit etablierter TCP-Verbindung scheitert, da die Verbindung bei späteren Restores resettet wird. Dies wird normalerweise, bei Checkpointing/Dumping und einfachem Restoring, durch entsprechende iptables-Regeln, gesetzt direkt nach dem Dumping, verhindert, diese Regeln werden während des Restoring wieder entfernt. Bei weiteren Restores sind diese Regeln also nicht mehr vorhanden.

2. CRIU stellt Anwendungen unter der gleichen PID wie zum Zeitpunkt des Checkpointings wieder her - wurde diese PID aber nach dem Checkpointing und vor dem Restoring wieder vergeben, so scheitert das Restoring.

3. Bei existierenden Netzwerkverbindungen ist zu beachten, dass diese zumeist eine beschränkte Lebenszeit haben, eine Zeit, nach der die Verbindung geschlossen, wird wenn keine Kommunikation statt findet. Diese Zeit wird beispielsweise für TCP-Verbindung unter Linux nicht in tatsächlicher Prozesslaufzeit gemessen, sondern als Differenz zwischen gespeicherter Zeit der letzten Kommunikation und momentaner Systemzeit berechnet - wenn nun also ein Prozess nach diesem Zeitfenster, der Linux-Default ist 7200 Sekunden, wiederhergestellt wird, so wird die Verbindung direkt wegen des Überschreitens der sogenannten TCP Keepalive Time geschlossen.

Für die genannten Probleme gibt es Lösungen, die eine Nutzung in einem Fuzzing-Umfeld ermöglichen.
Das TCP-RST-Problem (Nummer 1) lässt sich durch manuelles Setzen eben dieser Regeln direkt nach der Beendigung der Ausführung der Anwendungen lösen, beispielsweise wie in Listing 8 zu sehen.

Listing 8: Setzen der Firewallregeln

```
1     // [...]
2
3     // create the netfilter rules & iptables calls
4     char iptables1[200], iptables2[200], iptables3[200], iptables4
          [200];
5     sprintf(iptables1, "iptables -A OUTPUT -s 127.0.0.1/32 -d
          127.0.0.1/32 -p tcp -m mark ! --mark 0xc114 -m tcp --sport
          %i --dport %i -j DROP", clientport, serverport);
6     sprintf(iptables2, "iptables -A OUTPUT -s 127.0.0.1/32 -d
          127.0.0.1/32 -p tcp -m mark ! --mark 0xc114 -m tcp --sport
          %i --dport %i -j DROP", serverport, clientport);
7     sprintf(iptables3, "iptables -A INPUT -s 127.0.0.1/32 -d
          127.0.0.1/32 -p tcp -m mark ! --mark 0xc114 -m tcp --sport
          %i --dport %i -j DROP", clientport, serverport);
8     sprintf(iptables4, "iptables -A INPUT -s 127.0.0.1/32 -d
          127.0.0.1/32 -p tcp -m mark ! --mark 0xc114 -m tcp --sport
          %i --dport %i -j DROP", serverport, clientport);
9
10    // [...]
11
12    // wait for restored processes to terminate
```

```
13      waitpid(serverpid, NULL, 0);
14      waitpid(clientpid, NULL, 0);
15
16      // apply the netfilter rules
17      system(iptables1);
18      system(iptables2);
19      system(iptables3);
20      system(iptables4);
```

Die Problematik belegter PIDs (Nummer 2) lässt sich durch die Isolation der Anwendung durch beispielsweise eine Virtuelle Maschine beziehungsweise einen Container oder die Nutzung von PID Namespaces[15] lösen. Unter Umständen ließe sich das PID-Problem auch durch die Deaktivierung der `--shell-job` Option lösen, dies würde jedoch das Ausführen der Anwendungen in einer eigenen Session erfordern - etwas die betrachteten Fuzzing-Engines jedoch nicht unterstützen.
Der TCP Keepalive Timeout (Nummer 3) und verwandte Probleme lassen sich auf mehrere Arten und Weisen lösen, sollten sie ein Problem darstellen. Entweder kann die Zeit bis zum Timeout verlängert werden [6] oder der erfasste Zeitpunkt der letzten Nachricht im CRIU-Image kann angepasst werden.

## 5.4 Performance

Bei der Betrachtung der Performance von C/R-Systemen sind zwei Aspekte zu betrachten:

- Die Performanceeinflüsse auf die Laufzeit des Programms, ausgelöst beispielsweise durch das Sammeln von Informationen, die für das Checkpointing benötigt werden
- Die benötigte Zeit für die Durchführung der tatsächlichen Checkpointing- und Restore-Operationen

Ersteres, Performanceeinflüsse auf das laufenden Programm, treten bei C/R-Systemen wie CRIU nicht auf, da vor dem Checkpointing weder dem Kernel noch der Anwendung selbst bewusst ist, dass sie gecheckpointed werden wird, und nach dem Restoring zwar Reste des Restorer Context Blobs noch in Memory vorhanden sein können, diese jedoch nicht mit dem Anwendungscode interagieren und entsprechend keinen Einfluss auf die weitere Performance haben werden.
Die Geschwindingkeit, mit der Checkpointing und Restore Operationen durchgeführt werden können, spielt für klassische Zwecke wie Debugging oder Fault-Recovery zumeist eine untergeordnete Rolle. Hier wird zumeist die Anwendung nur wenige Male gecheckpointet und restoret, und die Grenzen der Akzeptabilität der Wartezeit werden zumeist durch menschliche Interaktion oder die normal erwartete Startup-Zeit bestimmt. Bei Nutzung von C/R-Systemen für Fuzzing sieht dies natürlich anders aus: Die Dauer einer Restore-Operation $T_{Restore}$ ist wie in Kapitel 4 dagelegt von hoher Relevanz für das C/R-Fuzzen, da das Fuzz-Target für jede Iteration einmal restoret werden muss.
Die benötigten Zeiten für C/R-Operationen hängen dabei von einer Reihe von verschiedenen Umständen ab, unter anderem:

- allokierter/genutzter Speicher der Anwendung

[6] `echo $new_time_in_sec > /proc/sys/net/ipv4/tcp_keepalive_time`

| Iterationen | 1byte | $10^2$byte | $10^4$byte | $10^6$byte | $10^8$byte | TCP |
|---|---|---|---|---|---|---|
| 10 | 0.292s | 0.290s | 0.292s | 0.301s | 0.627s | 0.982s |
| 50 | 1.269s | 1.269s | 1.256s | 1.298s | 2.332s | 4.615s |
| 100 | 2.459s | 2.494s | 2.477s | 2.558s | 4.482s | 9.170s |
| 500 | 12.205s | 12.228s | 12.250s | 12.648s | 21.564s | 45.541s |
| $T_{Restore}$ | 0.024s | 0.024s | 0.024s | 0.025s | 0.043s | 0.091s |

Abbildung 2: Performancemessung

- Größe des Prozessbaumes
- Auswahl zu restorender Features wie TCP-Sockets

Da natürlich auch externe Einflüsse wie Hardware, Auslastung des Systems, ... einen Einfluss auf die Performance haben und diese Zeiten zumeist, wie oben dargelegt, nicht von hoher Relevanz in der klassischen Nutzung sind, liegen in der Literatur kaum relevante Vergleiche der Performance verschieder C/R-Systeme vor.

Abbildung 2 zeigt die Ergebnisse unserer Performancemessung in Hinsicht auf Laufzeit eines Launchers, der Anwendungen startet, checkpointet, und diese wiederholt wiederherstellt, sobald die Anwendung terminiert. Diese Messung fand auf einem Lenovo Thinkpad X230 statt, die Laufzeiten sind das arithmetische Mittel jeweils mehrerer Messungen, für die gesamten Ergebnisse siehe Anhang [TODO]. Die Anwendungen sind dabei

1. Eine einzelne Anwendung, die eine bestimmte Anzahl Bytes zufälliger Daten liest, sich selbst checkpointed, eine sehr simple Operation auf den Daten ausführt, und das Ergebnis auf `stdout` ausgibt (TODO: Anhang)
2. Ein Client-Server-Paar, welches eine TCP-Verbindung aufbaut, sich selbst checkpointet sich gegenseitig jeweils ein kurzes Datenpaket zusendet und dabei Informationen über den Zustand auf `stdout` ausgeben (TODO: Code im Anhang)

Dabei setzen sich die Zeitmessungen über $n$ Iterationen $T(n)$ wie in Gleichung 7 zusammen. Durch die Kombination von Messungen mit verschiedenen Iterationen, siehe Gleichung 8, lässt sich dies dann als lineares Gleichungssystem formulieren und sowohl $T_1$ (Ausführung bis inklusive Checkpointing) als auch $T_2$ (iteriertes Restore und Execute) abschätzen.
$T_{Execute}$ wurde aus dem Programm heraus als deutlich unter $1ms$ gemessen, das heißt $T_2 = T_{Restore} + T_{Execute}$ kann bei den erhaltenen Werten als eine sehr gute Approximation für $T_{Restore}$ angesehen werden.

$$T(n) = T_1 + nT_2 := (T_{Setup} + T_{Checkpoint}) + n(T_{Restore} + T_{Execute}) \quad (7)$$

$$\begin{aligned} T(n_1) &= T_1 + n_1 T_2 \\ T(n_2) &= T_1 + n_2 T_2 \end{aligned} \quad (8)$$

Hiermit erkennt man aus Abbildung 2, dass die "Grundkosten"bei CRIU für einen einzelnen Restore bei ca. $24ms$ für ein triviales Fuzz-Target liegen würden, bei höherem Speicherverbrauch oder komplexeren Anwendungen wie einem Client-Server-Paar kann dies leicht auf $90ms$ oder mehr steigen. Mit individuell gemessenen Setup-Zeiten von maximal $10ms$ ist zu erkennen, dass mit CRIU in den betrachteten Anwendungsszenarien gemäß Gleichung 3 in Kapitel 4.1 keine Laufzeitersparnis sondern sogar ein signifikanter Laufzeitverlust erzielt würde. Ähnliche Untersuchungen simpler, insbesondere nicht vernetzter, Anwendungen mit Linux-CR[11] erreichten auf älterer Hardware Restore-Zeiten von $\leq 1ms$. Wir vermuten, dass diese Diskrepanz auf die verschiedenen Ansätze zum Restoring zurückzuführen ist, insbesondere benötigt Linux-CR nicht die Injektion der Blobs durch `ptrace` und den damit verbundenen Overhead, da Linux-CR direkt mit Kernel-Ressourcen arbeiten kann. Insbesondere heißt dies aber, dass auch wenn verfügbare mächtige C/R-Tools recht lange Checkpoint- und Restore-Zeiten beobachten lassen, es noch Optimierungspotential in dieser Richtung gibt. Wir können bei einer generellen Bewertung der Möglichkeiten von C/R-Fuzzing also noch mit einer Verbesserung dieser Werte rechnen, wenn entsprechende Entwicklungsanstrengungen unternommen werden.

# 6 C/R-Fuzzing: Implementierung

Dieses Kapitel untersucht die Nutzbarkeit der in Kapitel 2.4 und 2.5 vorgestellten Fuzzing-Engines für C/R-Fuzzing mit C/R-Tools am Beispiel von CRIU nach dem in Kapitel 4.2.1 vorgestellten Ansatz.

## 6.1 libFuzzer

Um die Nutzbarkeit libFuzzers für C/R-Fuzzing zu testen, implementierten wir ein beispielhaftes Fuzz-Target wie in Kapitel 4.2.1, Listing 6, siehe Anhang [TODO]. Das Zwischenspeichern des Inputs ist zwingend notwendig, da libFuzzer den Input immer als Argument beim Aufruf übergibt. Eine erfolgreiche Ausführung wurde unter anderem durch folgende Probleme verhindert:

- Ist AddressSanitizer[16] (ASan) aktiviert, so scheitert CRIU bereits in der ersten Ausführung während des Checkpointings: Eine Page wird als $\sim 300GB$ groß erkannt, die Allokierung entsprechendes Speichers scheitert und damit auch das Pagemapping und das Pagedumping mittels des Parasitencodes (siehe Listing 9). Die Ursache hierfür ist, dass AddressSanitizer $20TB$ virtuellen Speicher allokiert[17]. Restore-Versuche scheitern erwartungsgemäß daran, dass das Image nicht komplett erzeugt wurde (insbesondere `inventory.img`, die Top-Level-Beschreibung des Images, fehlt, siehe Listing 10). Prinzipiell lässt sich ASan abschalten, allerdings beschränkt natürlich die Aussagekraft des Fuzzings.

- CRIU setzt zwingend eine Wiederherstellung unter den gleichen PIDs vorraus. Andererseits hält libFuzzer die PID seine fuzzTargets dauerhaft belegt. Dies führt bei der ersten Ausführung des Fuzz-Targets mit Restore zu einem Konflikt um die PID und dem Scheitern des Restoreversuches, siehe Listing 11. Eine Isolierung der Fuzz-Targets in eigene PID-Namespaces wird durch libFuzzer nicht unterstützt.

Listing 9: Pagemapping/dumping Errors mit ASan

```
1 (00.010591) Error (criu/pagemap-cache.c:54): pagemap-cache:
      pmc_init: Can't allocate 30064246792 bytes
2 (00.010597) Error (criu/pagemap-cache.c:85): pagemap-cache: Failed
      to init pagemap for 28814
3 (00.010600) Error (criu/mem.c:516): Can't dump page with parasite
```

Listing 10: Restoring Error mit ASan

```
1 (00.000108) No inventory.img image
2 (00.000133) Error (criu/util.c:693): Can't read link of fd -404: No
      such file or directory
3 (00.000142) Error (criu/protobuf.c:75): Unexpected EOF on (null)
```

Listing 11: Restoring Error durch PID mismatch

```
1 (00.007567) Error (criu/cr-restore.c:1668): Pid 9177 do not match
      expected 8154
2 (00.007683) Error (criu/cr-restore.c:2309): Restoring FAILED.
```

Selbst wenn diese Probleme gelöst werden sollten, wäre eine Nutzbarkeit libFuzzers für C/R-Fuzzing (sowohl mit CRIU als auch anderen verfügbaren C/R-Systemen) schwer vorstellbar: libFuzzer kann zwar mit Multi-Threaded, aber nicht notwendiger korrekt mit Multi-Prozess-Anwendungen betrieben werden. Eine allgemeine Nutzbarkeit, unabhängig von den zu testenden Programmen, ist unwahrscheinlich.

## 6.2 afl

Ebenso wie bei der Untersuchung libFuzzers implementierten wir für den afl-Test ein simples C/R-Fuzz-Target wie in Kapitel 4.2.1 beschrieben, siehe Anhang [TODO].
Hier verhindert ein architekturelles Problem eine erfolgreiche Ausführung: Durch die Fork-Server-Optimierung in AFL wie in Kapitel 2.5 beschrieben teilt sich jede Fuzz-Target-Ausführung als Prozess Memory Areas mit dem initialen Stopped Process Image. Wenn die Fuzz-Target-Ausführung CRIU-initiiert das Checkpointing ausführen soll scheitert dies, da CRIU Shared Memory Nutzung über SysVIPC entdeckt (zu einem nicht zu dumpenden Prozess) und deswegen das Checkpointing abbricht, siehe Listing 12. Damit steht die AFL-Architektur über Cloning mit Copy-on-Write aus einem Stopped Process Image einem C/R-System wie CRIU konzeptionell entgegen.
Unabhängig von diesem Architekturproblem sind Probleme wie bei libFuzzer diskutiert absehbar:

- Fehlender voller Multi-Process-Support
- ASan-Inkompatibilität zu CRIU[7]

[7]Die Nutzung von ASan ist zwar nicht per Default aktiviert, eine Nutzung wäre jedoch im Normalfall empfohlen.

Listing 12: Checkpointing Error durch Shared Memory

```
1 (00.017584) Error (criu/cr-dump.c:431): Task 25961 with SysVIPC
      shmem map @7f24f6a8b000 doesn't live in IPC ns
2 (00.017590) Error (criu/cr-dump.c:1411): Dump mappings (pid: 25961)
       failed with -1
3 (00.017622) Unlock network
4 (00.017628) Unfreezing tasks into 1
5 (00.017641) Error (criu/cr-dump.c:1709): Dumping FAILED.
```

### 6.3 Fazit

Die Ergebnisse aus Kapitel 5 zeigen für CRIU, dass ein vollständiger Restore außerhalb des Kernels hohe Laufzeitkosten $T_{Restore}$ weit über typischen $T_{Setup}$ mit sich bringt. Es ist zu vermuten, dass ähnliche Userspace- oder Hybrid-C/R-Tools auch ähnliche Laufzeitkosten bedeuten. Dies ist im eigentlichen Einsatzzweck der C/R-Tools zumeist kein Hindernis, verhindert aber ihre Verwendbarkeit im C/R-Fuzzing außer für sehr spezifische Problemstellungen mit extrem hohen $T_{Setup}$.

Versuche einer naiven Integration von CRIU in das Fuzz-Target von libFuzzer und afl (Kapitel 6.1 und 6.2) nach dem Ansatz aus Kapitel 4.2.1 sind nicht erfolgreich: Das Prozessmanagement im Betriebssytem kann die speziellen Anforderungen des C/R-Systems und der Fuzzing-Engines nicht gleichzeitig erfüllen. Besonders deutlich wird dies bei afl, dessen Fork-Server-Ansatz bereits schon ein ähnliches Ziel wie C/R verfolgt, aber genau dadurch ein explizites C/R im Fuzz-Target verhindert. Weitere Hindernisse wie das Checkpointing einer ASan-instrumentalisierten Binary mit enorm hohen Memoryanforderungen oder die korrekte Nutzung von Code-Coverage-Instrumentalisierung von wiederhergestellten Prozessen sind abzusehen.

## 7 Ausblick

Der praktische Teil dieser Arbeit (Kapitel 5 und 6) zeigt deutlich, dass die Fuzzing-Engines der erfolgversprechendere Ansatzpunkt für eine Implementierung von C/R-Fuzzing sind. Eine reine C/R-Implementierung im Fuzz-Target, ohne Mitwirkung der Fuzzing-Engine, wird auch mit eventuellen Workarounds erfolgreich keine stabile C/R-Fuzzing-Umgebung erzielen - es ist immer mit wieder neu auftretenden Problemen zu rechnen.
In Kapitel 4 haben wir bereits auf die konzeptuelle Ähnlichkeit des Fork-Server-Ansatzes in afl zu C/R hingewiesen (faktisch ein Setup+Restore); in [19] wird auch ein Ausblick erwähnt, den Zeitpunkt des Forks frei bestimmen zu können und damit das inputunabhängige Setup im Sinne dieser Arbeit nur einmal auszuführen. Mit diesem Ansatz ließe sich die Problemklasse aus Kapitel 4.1 und 4.2.1 angehen. Die Kosten $T_{Setup}$ fielen dann nur einmalig je Fuzzing an.

Komplexere Problemklassen wie die Erforschung des Zustandsgraphen eines Client-Server-Systems erfordern zusätzliche konzeptuelle Erweiterungen der Fuzzing-Engines. Das Checkpointing eines separaten Prozesses (der Server in

Kapitel 4.2.2) müsste explizit unterstützt werden; aufgrund der Performanceergebnisse aus Kapitel 5 ist hier kernelbasiertes Checkpointing (im Stile von Linux-CR) der erfolgsversprechendere Weg. Eine denkbare Alternative wäre Proto-Checkpointing im Stile des afl-Fork-Servers, dies würde jedoch höhere Anforderungen an das Zurücksetzen des Zustandes von geteilten Ressourcen wie File Deskriptoren stellen; wie in [19] erwähnt ist dieses Setzen des Zustandes nicht für alle geteilten Ressourcen, insbesondere Sockets, möglich.

Auch wenn die beschriebenen Erweiterungen von Fuzzing-Engines einen nennenswerten Engineering-Aufwand bedeuten werden ([19] stellt deutlich einige der Herausforderungen im Umgang mit Prozessen in Unix/Linux dar), ist in Anbetracht der Bedeutung der Sicherheit von IT-Systemen und den bereits erzielten Erfolgen durch Fuzzing damit zu rechnen, dass der Bedarf an Fuzzing-Engines für komplexere oder speziellere Problemklassen erkannt und mit weiterer Verfolgung des Fuzzing-Ansatz befriedigt werden wird.

# Anhang

### Performancemessungen mit CRIU

### libFuzzer Fuzz-Target

### afl Fuzz-Target

TODO

# Literatur


[1] Eine ausführliche Dokumentation zu CRIU findet sich unter `https://criu.org/Main_Page`.

[2] Eine grundlegende Dokumentation zu libFuzzer findet sich in den LLVM Docs unter `https://llvm.org/docs/LibFuzzer.html`.

[3] Eine Dokumentation zum Aufbau und der Nutzung afls ist unter `http://lcamtuf.coredump.cx/afl/` zu finden. Eine technische Dokumentation ist insbesondere unter `http://lcamtuf.coredump.cx/afl/technical_details.txt` zu finden.

[4] Heather Kelly. *Facebook hack exposed 50 million users' info – and accounts on other sites.* CNN Business, 2018. `https://money.cnn.com/2018/09/28/technology/facebook-breach-50-million/index.html`

[5] *Cyberangriff "WannaCry Hochfahren, einloggen, hoffen* Tagesschau, 2017. `https://www.tagesschau.de/ausland/wannacry-microsoft-103.html`.

[6] Ole Reißmann. *OpenSSL-Sicherheitslücke - Warum "Heartbleed"Millionen Web-Nutzer gefährdet.* Spiegel Online, 2014. `http://www.spiegel.de/netzwelt/web/heartbleed-openssl-fehler-verraet-passwoerter-a-963381.html`

[7] B. Potter, G. McGraw. *Software Security Testing*, IEEE Security & Privacy, vol. 2, no. 5, pp. 81-85, 2004. https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=1341418

[8] Joe W. Duran, Simeon Ntafos. *A report on random testing*, ICSE '81 Proceedings of the 5th international conference on Software, pp. 179-183, 1981. https://dl.acm.org/citation.cfm?id=802530

[9] Ari Takanen. *Fuzzing: the Past, the Present and the Future*. Actes du 7ème symposium sur la sécurité des technologies de l'information et des communications (SSTIC), pp. 202-212 2009. https://codeengn.com/archive/Reverse%20Engineering/Fuzzing/Fuzzing%20the%20Past,%20the%20Present%20and%20the%20Future%20[Ari%20Takanen].pdf

[10] Jason Ansel, Kapil Arya, Gene Cooperman. *DMTCP: Transparent Checkpointing for Cluster Computations and the Desktop*. 23rd IEEE international parallel and distributed processing symposium, Rome, Italy, pp. 1–12, 2009.http://dmtcp.sourceforge.net/papers/dmtcp.pdf

[11] Oren Laadan, Serge E. Hallyn. *Linux-CR: Transparent Application Checkpoint-Restart in Linux* Proceedings of the Linux Symposium, Ottawa, Canada, pp. 159-172, 2010. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.366.6842&rep=rep1&type=pdf#page=159

[12] Jason Duell. *The design and implementation of Berkeley Lab's linux checkpoint/restart*. Lawrence Berkeley National Laboratory, 2005. https://escholarship.org/uc/item/40v987j0

[13] Kostya Serebryany. *Simple guided fuzzing for libraries using LLVM's new libFuzzer*. 2015. http://blog.llvm.org/2015/04/fuzz-all-clangs.html

[14] Jonathan Corbet. *TCP connection repair*. lwn.net, 2012. https://lwn.net/Articles/495304/

[15] *Man page: pid namespaces*. Linux Man Pages. http://man7.org/linux/man-pages/man7/pid_namespaces.7.html

[16] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, Dmitry Vyukov. *AddressSanitizer: A Fast Address Sanity Checker*. Usenix Annual Technical Conference, 2012. https://storage.googleapis.com/pub-tools-public-publication-data/pdf/37752.pdf

[17] Hanno Böck. *Fuzzing with american fuzzy lop*. lwn.net, 2015. https://lwn.net/Articles/657959/

[18] Pavel Emelyanov. *Checkpoint/restore mostly in the userspace*. 2011. https://lwn.net/Articles/451916/

[19] Michał Zalewski. *Fuzzing random programs without execve()* 2014. https://lcamtuf.blogspot.com/2014/10/fuzzing-binaries-without-execve.html