Checkpoint/Restore in Fuzzing

Malte Klaassen

tbd

Zusammenfassung

NEEDS REWORK Fuzzing von komplexen Programmen als ganzes ist meist langsam, Fuzzing komplexer Programme in vielen kleinen Einzelblöcken ist aufwändig für den Tester da für sie jeweils die Fuzz-Targets geschrieben werden müssen. In diesem Stück Papier untersuchen wir die Nutzbarkeit von Checkpoint/Restore-Mechanismen zur Lösung solcher Probleme im Fuzzing, insb. mit Blick auf Netzwerkprotokolle mit einer wohldefinierten State Machine, präsentieren eine mögliche Architektur zum Fuzzen von Implemetierungen ebensolcher Protokolle und untersuchen verschiedene Fuzzer und Checkpoint/Restore Tools auf ihre Anwendbarkeit.

Inhaltsverzeichnis

1 Einleitung

Mit zunehmender Prävalenz und Prominenz von IT-Systemen ist das Thema der Sicherheit eben dieser Systeme zunehmend in den Fokus gerückt, sowohl in Fällen häufig fehlerbehafteter Nutzung von Sofware in denen Nutzer selbst, direkt betroffen sind wie bei Hacks oder Leaks bei Online-Plattformen wie Facebook oder Malware-Kampagnen wie WannaCry, häufig ausgelöst durch technische Sicherheitslücken und verzögerte Patches, als auch vereinzelt in Fällen von rein technischen Sicherheitslücken wie beispielsweise Heartbleed.
Proaktiv vs Reaktiv
Über die Jahrzehnte haben sich dabei verschiedene Ansätze und Methoden zur proaktiven Erkennung, Behebung und Verhinderung von ebensolchen Sicherheitsproblemen entwickelt, beispielsweise:

- 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 Codeanaylse
- (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 ist zumeist beschränkt auf die Vorstellungskraft des Testers beziehungsweise auf sein Verständnis der zu testenden Software, Grenzfällen oder komplett unerwartete aber mögliche Eingaben sind mit klassischem Testing schwer umfassend abzudecken.

1.1 Fuzzing

TODO: Corpus, Seeds, Dictionaries

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 nur in von Menschen mehr oder weniger präzise beschriebenen Testfällen getestet wird, ein Codestück, das so genannte Fuzz-Target, wiederholt mit computergenerierten zufälligen Inputs getestet wird, dabei wird die Ausführung genau von Fuzzer überwacht um beispielsweise Programmabstürze, das Lesen aus unitialisiertem Speicher oder auch undefiniertes Verhalten zu Erkennen und zu der problematischen Codestelle zurück zu verfolgen.

1.2 Klassifizierung von Fuzzern

Fuzzer werden anhand mehrerer Eigenschaften unterschieden, insbesondere ob sie sich beim Testen der Struktur des Programmes und der Struktur der erwartetenden Eingaben bewusst sind. Hierbei

- Wenn der Fuzzer sich der Struktur des Programmes bewusst ist und dieses Wissen im Testing nutzt um bessere Code-Coverage zu erreichen spricht man, je nach Technologie, von einem White-Box- oder Gray-Box-Fuzzer. 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.

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 agieren, diese Terme werden zumeist auch zur Beschreibung von Operationsmodi ebensolcher moderner Fuzzer verwendet.

1.3 Historischer Überblick

Testing mit randomisierten Inputs ist kein sonderlicher neuer Ansatz, jedoch ist die Effizienz von naivem Random Input Test (also Black-Box-Fuzzing) offensichtlich beschränkt und der größte Teil der Fehler die dauf diese Art und Weise gefunden werden können relativ simpel und selten in den Tiefen des Programmes versteckt, das heißt es handelt sich zumeist um Fehler die vergleichsweise simpel auch durch Code-Reviews oder klassische Tests gefunden werden könnten, entsprechenden 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, inklusive einiger Open Source Fuzzern und Fuzzing Frameworks, hin zu Fuzzing als Kernbestandteil des Sicherheits-Testings vieler relevanter Open Source Software Projekte, vorrangetrieben insbesondere durch Projekte wie Googles OSS-Fuzz, ein Service zum kontinuierlichen Testen verschiedenster Open Source Software durch mehrere verschiedene Fuzzer.

1.4 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 Ausführug des Fuzz-Targets zu lange dauert. Dies führt dazu, 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 und so beispielsweise unnötigen Overhead im Setup des Fuzz-Targets vermeidet. Aber auch dies hat seine Nachteile: Der Tester muss hier nun mit der Struktur des zu testenden

Systems bekannt sein, muss das komplexe System in kleinere Systeme selbst zerlegen und dann für jedes der Subsysteme einzelne Fuzz-Targets schreiben - dies ist zeit- und arbeitsaufwändig und, ähnlich wie beim klassischen Testing, anfällig für Fehler des Testers.

2 Checkpoint/Restore

Mit Checkpoint/Restore, kurz C/R, manchmal auch Checkpoint/Restart, bezeichnet man das Speichern eines Prozesses oder einer Andwendung in einer Art und Weise die eine spätere Wiederherstellung und weitere Ausführung der Anwendung erlaubt. C/R-Systeme finden Anwendung in verschiedenen Bereichen, beispielsweise im Debugging und Fault Recovery durch das Zurücksetzen eines Programmes auf einen vorherigen Zustand, in der Beschleunigung oder Optimierung von Prozessen durch das Fortsetzen von Anwendungen von Checkpoints anstatt rechenzeitaufwändigen Komplettausführungen und dem Aussetzen der Ausführung von Programmen wenn diese grade nicht benötigt werden oder, je nach Fähigkeiten des zugrundeliegenden C/R-Systems, sogar die Migration von laufenden Anwendungen zwischen Maschinen.

TODO: Was alles muss gespeuchert werden?

2.1 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 nach zwei Faktoren klassifizieren: Typ bzw. Implemetierungsebene und nach den Fähigkeiten des Checkpoint/Restore-Systems, also welche Aspekte des Programms genau wiederhergestellt werden können und welche nicht.

2.1.1 Nach Typ

Bei der Klassifizierung nach Typ oder Implemetierungsebene unterscheiden wir grob nach drei Kategorien: Das Checkpointen im Userspace, aus dem zu checkpointenen Programm selbst, das Checkpointen auf Kernelebene und das Checkpointen von Virtuellen Maschinen beziehungsweise Containern.

Im klassischen Userspacecheckpointing findet das Checkpointing 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 ebenfalls nur ohne besondere Privilegien zugängige Ressourcen zur Wiederherstellung. Dieses Vorgehen führt in reinen Userspace-C/R-Systemen wie DMTCP (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 spätestens zur Startzeit mit der Anwendung gelinkt werden müssen, häufig Beschränkungen darauf haben mit welchen Programmen (bspw. Sprachen) sie genutzt werden können und unter Umständen die Funktionsweise des Programmes ändern könnten sowie mögliche Performanceeinbußen durch das duplizieren von Kernelstrukturen

im Userspace.

Dem gegenüber steht das Checkpointing auf Kernelebene, beispielsweise BLCR (Berkeley Lab's Linux Checkpoint/Restart) oder Linux-C/R. C/R-Systeme nutzen den Kernel direkt um alle nötigen Informationen wärend des Checkpointings zu sammeln, benötigen also keine Libraries zum Sammeln eben dieser Informationen durch das Abfangen von Syscalls o.ä., und haben alle nötigen Privilegien um, theoretisch, falls implementiert, alle Eigenschaften der Anwendung auch korrekt wiederherzustellen. Allerdings muss der Kernel eben diese Funktionalitäten anbieten damit Kernelebene-C/R-Tools eingesetzt werden können und obwohl es wiederholt versuche der Integration von C/R-Tools in den Mainline 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 Kernelebene-C/R-Tools folgt zumeist einem Schema änhlich diesem:

1. Einfrieren und Synchronisieren der Prozesse und Threads
2. Erfassen von globalen Informationen (d.h. Informationen die nicht direkt Teil der zu checkpointenden Anwendung sind wie Namespaces oder Container)
3. Erfassen der Prozess- bzw. Thread-Hierachien und -Informationen wie Parent/Child-Verhältnisse, PIDs, ...
4. Erfassen des Statuses der einzelnen Prozesse und Threads, bspw. Signals, geöffnete Files und FDs, ...
5. Beenden oder Weiterausführung der Anwendung, Schreiben all der erfassten Informationen

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 änhlicher Weise:

1. Wiederherstellen der globalen Information, beispielsweise Container bei Multiprocess-Anwendungen oder dem Prozess bei Multithreaded aber Singleprocess Anwedungen, wenn nötig
2. Erstellen einer Prozess- bzw. Threadhierachie entsprechend der erfassten Hierachie (in eingefrorenem Zustand)
3. Wiederherstellung des Statuses der einzelnen Prozesse und Threads
4. "Fortsetzen"beziehungsweise Auftauen der Prozesse/Anwendung

CRIU (Checkpoint/Restore in Userspace) ist ein Hybrid aus Userspace- und Kernelebene-Systemen, das heißt es agiert größtenteils in Userspace, nutzt jedoch Calls auf privilegierte Operationen wie das Forken mit bestimmten PIDs. 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, benötigt insbesodere keine Intercept-Libraries wie normale Userspace-C/R-Tools, jedoch können nicht all diese Informationen von externen Programmen erfasst oder hergestellt werden.

Dies umgeht CRIU indem es mittels ptrace sogennanten parasite code in die zu checkpointenen Prozesse einfügt (beziehungsweise den restorer blob bei der Wiederherstellung) die eben diese Informationen erfassen.

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

2.1.2 Nach Capabilities

Zusätzlich unterscheiden sich verschiedene C/R-Tools auch noch danach was genau sie Checkpointen und Restoren können. Einige Tools wie frühere BLCR-Versionen unterstützen nur einzelne multi- oder singlethread Prozesse, Linux-CR unterstützt das Checkpointen beliebiger Container oder Subtrees von Prozessen. Tools die innerhalb von multithreaded Prozessen arbeiten, beispielsweise einzelne Threads checkpointen und wiederherstellen können existieren anscheinend nicht, zumindest fanden wir keine. Dies ist auch plausibel, da solche Threads sich eben Ressourcen teilen und daher ein teilweises Wiederherstellen nur einiger Threads zu undefiniertem Verhalten führen würde.
Weitere Unterschiede gibt es in der Frage welche Eigenschaften von laufenden Programmen wiederhergestellt werden können, beispielsweise Netzwerkverbindungen wie TCP- oder UDP-Sockets, Dateien oder Namespaces, sowie bezüglich der Frage ob beliebige Programme gecheckpointed werden können oder ob bereits beim Start der Anwendung bekannt sein muss, dass diese gecheckpointed werden soll und entsprechende Vorbereitungen wie das Preloading von Libraries oder das Informieren des Kernels getroffen werden müssen.

3 Fuzzing mit C/R

Wie oben erwähnt gibt es im klassischen Fuzzing einige Probleme: Komplexe Systeme oder große Overhead im Set-Up des Fuzztargets führen zur Reduktion des Durchsatzes und damit Einschränkung der Effizienz, mehrstufige interaktive Programme wie beispielsweise Netzwerkprotokolle sind als ganzes nur mit großem Aufwand oder sogar gar nicht vernünftig zu fuzzen. In dieser Arbeit wollen wir untersuchen, ob sich einige oder sogar alle dieser Probleme durch die Nutzung von Checkpoint/Restore-Mechanismen beheben oder zumindest in ihren Auswirkungen reduzieren lassen. Dazu betrachten wir zunächst einmal generell mögliche Ansätze und ihre theoretischen Vor- und Nachteile bevor wir ihre Implementierbarkeit mit kontemporären Fuzzern und Checkpoint/Restore-Systemen betrachten.
Generell sehen wir zwei Ansätze zur Integration von Checkpoint/Restore-Tools in Fuzzern:

1. Integration des C/R-Tools in den Fuzzer selbst, das heißt der Fuzzer selbst übernimmt das Checkpointing und Restoring
2. Implementierung des Checkpointing und Restoring im Fuzztarget, transparent zum Fuzzer

Die erste Alternative, ein C/R-Fuzzer, wäre aber natürlich eine sauberere Lösung als die ad hoc Lösung der Implementierung im Fuzztarget und bietet die selben oder sogar mehr Möglichkeiten wie die Fuzztarget-Lösung, benötigt aber natürlich eben einen C/R-Fuzzer, etwas das unseren Nachforschung nach bisher nicht einmal in Ansätzen existiert.

3.1 C/R-Fuzzer

Klassische moderne Fuzzer wie libfuzzer oder afl folgen während des Fuzzings im Allgemeinen diesem Aufbau:

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 heißt das Fuzz-Target wird so oft mit verschiedenen Inputs ausgeführt bis ein Problem auftritt. Dabei hat das Fuzz-Target in Normalfall eine Form wie diese (oder kann in diese Form gebracht werden):

Listing 2: Struktur Klassisches Fuzztarget

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

Hierbei ist das Ergebnis von 1. im Allgemeinen entweder konstant oder zumindest austauschbar wenn es beispielsweise externe (pseudo-)zufällige Einflüsse wie Nutzung der Systemzeit oder Aufrufe von Zufallszahlgeneratoren gibt. Dies heißt aber nun, dass wir das Fuzz-Target aufteilen können wie folgt:

1. Inputunabhängiges Set-Up
2. Inputabhängiges Set-Up und zu testende Funktionalität

Wenn wir dies machen können wir nun als Alternative zum klassischen Fuzzer einen C/R-Fuzzer nutzen, der folgendem Aufbau folgt:

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 Input-Abhängigen Teil des Fuzz-Targets während
          der Ausführung und update den Corpus wenn nötig
```

Hierbei müssen natürlich ein paar Dinge beachtet werden: 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 heißt 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 könnte dann wie folgt aussehen:

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 sollte. Ein solcher C/R-Fuzzer kann unter Umständen das Problem großen Overheads im Setup lösen oder zumindest reduzieren. Mit T_{Setup} die Laufzeit des inputunabhängigen Setups, $T_{Execution}$ die (mittlere) Laufzeit des inputabhängigen Setups und des zu testenden Codes, $T_{Checkpoint}$ und $T_{Restore}$ die benötigte Zeit für eine Checkpointing- beziehungsweise Restoreoperation. Dann ergibt sich für die Laufzeit eines normalen, nicht-C/R Fuzzers bei n benötigten Operationen

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

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 Zeitersparniss 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 die Größe im Speicher und die zu checkpoitenen Features wie Netzwerksockets, und natürlich das genutzte Checkpoint/Restore-Tool, eine genauere Betrachtung davon findet in Kapitel CRIU.Performance statt.

3.2 Non-C/R-Fuzzer mit C/R Ansätze

Ein anderer Ansatz zur Kombination von Fuzzing mit Checkpoint/Restore-Mechanismen wäre, statt des impliziten Checkpointing durch den Fuzzer, eplizites Checkpointing und Restoring im Fuzztarget. Dies hätte den Vorteil einer größeren Flexivilität im Einsatz und würde im Idealfall keinerlei Änderungen am Fuzzer benötigen, könnte also unter Umständen bereits mit vorhandenen Fuzzern genutzt werden - darauf welche Umstände dies sind und wie es sich ob der Anwendbarkeit bei modernen Fuzzern verhält gehen wir in Kapitel CRIU+Fuzzing Engines ein. Wir betrachten hier zwei Varianten dieses Ansatzes: Zuerst eine

recht 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, danach eine Architektur die durch Anwendung von Checkpoint/Restore versucht das effiziente Fuzzen komplexer mehrstufiger Programme, beispielsweise Netzwerkprotokolle, durch Exploration des Zustandsgraphen als ganzes zu ermöglichen.

3.2.1 Naiv

Eine konzeptuell recht simple Variante der Kombination von klassischen, Non-C/R-Fuzzern mit C/R-Implementierung im Fuzztarget wäre naives Restoring nach dem inputunabhängigen Setup, änhlich wie bei dem oben beschriebenen C/R-Fuzzer. Anders als beim obigen C/R-Fuzzer muss hier jedoch im Fuzztarget entschieden beziehungsweise bestimmt werden, ob wir uns in der ersten Iteration befinden, das Setup also ausgeführt werden und danach gecheckpointed werden muss, oder ob wir uns in einer späteren Iteration befinden, wir also das Setup überspringen und direkt, mittels Restoring, zur Ausführung des zu testenden Codes springen. Dies kann beispielsweise durch das Setzen von Systemvariablen oder das Erstellen von Tokens im Dateisystem geschehen - ist die Variable bei einer Ausführung also gesetzt oder existiert die Datei so befinden wir uns in einer späteren Ausführung und sollten das Setup durch einen Restore überspringen. Konkret sollte ein solches Fuzztarget dann hierfür in etwa so aussehen:

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 (ohne Dump)
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 wieder hergestellt und setzt ihre Ausführung in Zeile 7 fort, holt sich also neuen Input und testet mit diesem.
Hierbei können jedoch einige Probleme auftreten die zu beachten sind:

- Wie beim C/R-Fuzzer muss natürlich das Setup inputunabhängig sein beziehungsweise in einen inputunabhängigen Teil, welcher vor dem Checkpoint ausgeführt wird, und einen inputabhängigen Teil, welcher mit dem Test selbst ausgeführt wird, aufgebrochen werden.

- Der Fuzzer muss mit Multiprocess-Programmen umgehen können und über das Vorhandensein des restorten Prozesses informiert sein, inbesondere muss er Exceptions, undefiniertes Verhalten und anderes unerwünschtes Verhalten im restorten Prozess erkennen können. Dies könnte beispielsweise durch eine restore-as-child Funktionalität wie beispielsweise von CRIU angeboten realisiert werden, benötigte jedoch trotzdem noch einen robusten Multiprocessfähigen Fuzzer

- Der restorte Prozess muss in irgendeiner Form an den Input kommen - dies könnte mit kontemporären Fuzzern unter Umständen ein Problem darstellen, da diese zumeist den Input entweder beim Aufruf des Fuzztargets als Argument oder durch einen FD wie `stdin` übergeben werden - beides Ressourcen auf die der wiederhergestellte Prozess im Allgemeinen keinen Zugriff hat. Dieses Problem könnte gelöste werden indem der restorte Prozess den Input direkt von Fuzzer erhält, beispielsweise indem der Fuzzer die Inputs über einen FIFO oder eine ähnliche Ressource im Filesystem, außerhalb des Prozesses selbst, übergibt oder indem der restorende Prozess vor dem Restore den Input in einer Art und Weise abspeichert, so dass der restorte Prozess diesen Input auslesen kann, beispielsweise wieder durch eine Zwischenspeicherung im Filesystem (siehe Listing WASAUCHIMMER).
- Die Messung der Codecoverage bei White- oder Gray-Box-Fuzzern dürfte wenig zuverlässig sein wenn im Fuzztarget selbst restoret wird - einerseits durch die Einführung von neuen Codeelementen durch das C/R-Tool, wobei dies natürlich vom C/R-Ansatz und dem genutzten C/R-Tool abhängt, saubere Kernelebene-C/R-Tools dürften hier beispielsweise wenig Einfluss haben, andererseits durch die massive und durch den eigentlich Programmcode nicht beschriebene Veränderung des Prozesses durch das Restore. Dies dürfte dazu führen, dass ein solcher Non-C/R-Fuzzer wenig besser als ein Black-Box-Fuzzer funktionieren dürfte.

Listing 6: Struktur C/R-Fuzztarget für Non-C/R-Fuzzer mit Inputdump

```
1  Falls der Token existiert:
2      Hole den Input
3      Schreibe den Input in eine Datei
4      Restore
5  Sonst:
6      Führe Setup aus
7      Setze Token
8      Checkpointe das Programm (ohne Dump)
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:

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

Die Vereinfachung in (a) kommt dabei aus der Annahme, dass die diversen Token- und Inputoperationen verglichen mit den recht langsamen Restore- und Setup-Operationen für die letztendliche Laufzeit nicht ins Gewicht fallen beziehungsweise im Falle der Inputoperationen auch im Non-C/R-Fall ausgeführt werden müssen, dies dort jedoch versteckt geschieht.
Das $T_{C/R}$ in diesem Fall unterscheidet sich nicht relevant vom $T_{C/R}$ 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

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

3.2.2 Exploration des Zustandsgraphen in Client/Server Systemen

Die bisher beschriebenen Ansätze bieten nur sehr naive Ansätze zur Lösung nur eines der beschriebenen Fuzzingprobleme - Laufzeitoverheadreduktion in einstufigen, simplen Anwendungen. In diesem Kapitel beschreiben wir einen Ansatz zum Testen einer Seite komplexerer, insbesondere mehrstufiger, interaktiver 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 Exceptions gestestet. 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 hat 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 indem es die von der Fuzzing-Engine generierten Inputs als Client-Input an den Server übergibt, beispielsweise über eine etablierte TCP-Verbindung zwischen dem Server und Client.

- Verwaltung des Servers: Das Fuzztarget hält in persistenter Form, beispielsweise durch Schreiben ins Dateisystem, den bisher gefundenen Zustandsgraphen des Servers, also die gefundenen Zustände und die Inputs die zu Übergängen führen, vor. Zusätzlich wird jeder Zustand wenn er das erste mal erreicht wird gecheckpointet und das entsprechende Image mit dem Zustand verknüpft, das heißt das Fuzztarget muss auch bestimmten wann neue Zustände erreicht werden.

Hierbei werden dann neu generierte Inputs aber nicht nur auf einen bestimmten Zustand angewendet sondern entweder hintereinander auf alle bisher bekannten Zustände oder auf zufällig gewählte Zustände, so dass der Zustandsgraph weiter ausgebaut wird.
Ein solches Fuzztarget könnte in Pseudocode wie folgt aussehen:

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 = [] // [(input, new_state)]
8      states <- {state_id = {s = s, c = c, t = transition, id =
           state_id}}
9  for state in states:
10     server <- cr.restore(state[s])
11     connection <- restore(state[c])
12
13     input <- fuzzer.input
14     connection.send(input)
15     if state.id ≠ server.state and server.state not in states.keys:
16         new_c <- checkpoint(connection)
17         new_state_id <- server.state
18         new_s <- cr.checkpoint(server, dump=true)
19         new_t = []
20         states[state][t].append((input, new_state_id))
21         states[new_state_id] = {s = new_s, c = new_c, t = new_t, id
               = new_state_id}
22     else if state.id ≠ server.state:
23         states[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 (beispielsweise Zeile 4) wird dabei nicht durch das normale C/R-Tool durchegeführt, da nicht der ganze Client gecheckpointed wird, sondern durch andere, der Art von `connection` angemessene Ansätze. 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 Fuzztarget/Client mitteilen können (siehe: `state_id <- server.state` etc.), beispielsweise durch Einfügen entsprechender Funktionalität in den Servercode und die Übergabe durch einen entsprechenden IPC-Ansätze oder direkt durch das Auslesen des States aus den C/R-Dumps.
Für einen Server mit bereits bekannten Zuständen s_0, s_1, s_2, s_3 mit den Zustandsübergängen wie gezeigt und entsprechenden Inputs für die Übergänge $i_1, \ldots, i_4$ würde nun also für einen neuen Input $i*$ wie in Abbildung 1 gezeigt jeder der Zustände $s_0, \ldots, s_3$ einmal wiederhergestellt, mit $i*$ getestet und falls Zustand s_i mit Input $i*$ einen neuen Zustand erzeugt dieser gecheckpointed und als Knoten s_4 mit der Kante $(s_i, s_4, i*)$ dem Graphen hinzugefügt.

Durch diese schrittweise Erforschung des Zustandsgraphen kann der Fuzzer nun an den verschiedenen Zuständen ansetzen und solte so durch Formulierung nur eines Fuzztargets in der Lage sein die ganze Funktionalität dieses Servers zu testen - anstatt wie im klassischen Fuzzing einzelne Fuzztargets für jeden Pfad in diesem Zustandsgraphen zu schreiben oder sogar nur einzelne Funktionen in Isolation zu testen.
Hierbei gilt natürlich zu beachten, dass auch dieser Ansatz einige Probleme mit

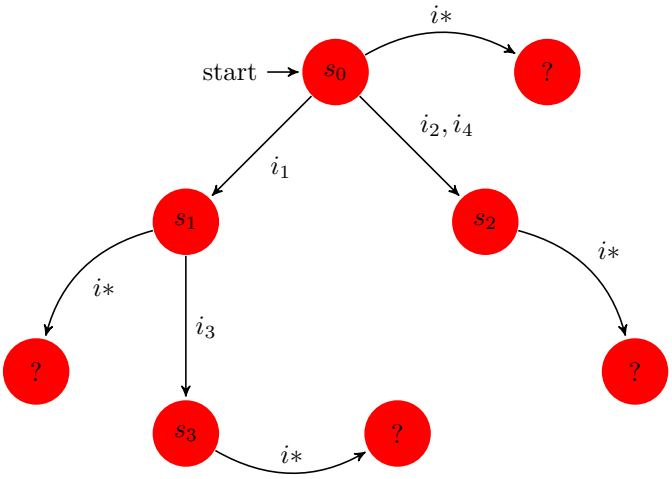

Abbildung 1: Zustandsübergangsexploration mit Input $i*$

sich bringen dürfte:

- White-Box-Fuzzing beziehungsweise klassisches Gray-Box-Fuzzing durch Code-Coverage werden, wie schon bei den naiven Ansätzen, 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 von Exceptions einbeziehen.

- Das Formulieren von Regeln, Dictionaries, ... für den Inputs eines Smart-Fuzzers wird verkompliziert, da diese Regeln nun nichtnur für einen Pfad oder eine bestimmte Funktion gelten sollten 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 zu Performanceeinbußen führen kann

- In Fällen in denen es nur wenige gültige Übergänge auf einen Zustand mit hoher Distanz vom Ausgangsknoten gibt könnte 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.

4 CRIU

CRIU is probably the only C/R tools that could make sense right now, two reasons why: 1. It's actually still getting updates and is actually easily useable on modern systems unlike many of the other things 2. It's capabilities exceed any other modern C/R tool

4.1 Funktionsweise

Detailed description of internal functionality of CRIU.

4.2 Capabilities

What can we restore with CRIU? Pretty much any program, independent of language etc, does not require linking with some interception library, marking at start, ... CRIU also can restore pretty much any feature we could find, especially established TCP connection.

4.3 Probleme

It does not work out of the box in a fuzzing context, due to: 1. Firewall rules and resets, we can fix that by adding those manually after each iteration. 2. PIDs are already in use again. There could be two fixes here: Either putting it in it's own pid namespace or possibly by running it in a context that does not require exact PID matches, e.g. detached from the shell, ...

4.4 Performance

Here are estimates for performance, as in "$T_{Restore} = x$. There are also comparisons to benchmarks for other fuzzing engines we found, especially Linux-CR which seems to have way better performance (though we were not able to get Linux-CR working on our machine so we have to rely on data from their paper). Short discussion what makes CRIU so much slower than Linux-CR and what performance depends on anyway.

5 CRIU + Fuzzing Engines

We have a quick look at modern fuzzing engines.
Now, this is where it gets annoying: CRIU - or any other CR system - doesn't work with pretty much any modern fuzzing engines, most notably as they restore processes, not threads. To fix this we would either need 1. In-Process Thread-Restoring (and there are a few reasons why that probably isn't the best of all ideas - most notably shared ressources in different stats, shared memory where both code lives, ...) 2. Multi-Process Fuzzing Engines - yeah, we don't have those yet apparently.

6 Fazit und Ausblick

There still is room for research in this area, especially if higher performance C/R-Systems like Linux-C/R are a thing and once Multi-Process-Fuzzer exist.