Evolutionäres unstrukturiertes Fuzzing von L4-Mikrokernen

Daniel Loebenberger[1] · Steffen Liebergeld[2]

[1]genua GmbH
daniel_loebenberger@genua.de
[2]Kernkonzept GmbH
steffen.liebergeld@kernkonzept.com

Zusammenfassung

Fuzzing ist eine erfolgversprechende Technik, um auf semi-automatische Weise Schwachstellen in sicherheitsrelevanten Systemkomponenten aufzudecken. Dabei ist es neuerdings erstmals möglich, Schnittstellen eines Betriebssystemkerns direkt im laufenden Betrieb mittels evolutionärem Fuzzing zu analysieren. Wir übertragen diese Technik im Rahmen dieser Arbeit auf besonders sicherheitskritische Kerne, spezifisch den L4Re-Mikrokern, und beschreiben neuartige Ansätze, welche eine auf Fuzzing basierende Analyse dieses Kerns ermöglichen.

1 Einführung

L4-Mikrokerne, abgeleitet aus den Arbeiten von Liedtke [Lied95], gehören mittlerweile zum Standardrepertoire in der Entwicklung sicherheitskritischer IT-Systeme. Zu den bekanntesten Anwendungen gehört der Sicherheits-Coprozessor auf Apple iPhones, der in der heutigen digitalen Gesellschaft weite Verbreitung findet [Appl16]. Durch die geringe Komplexität sind L4-Mikrokerne gut analysierbar. Die Hauptfunktionalität dieser Kerne ist jedoch die Etablierung einer Separationsfunktionalität von Betriebssystemkomponenten und Anwendungen während des laufenden Betriebs. Dabei realisieren sie abgeschottete Bereiche, in denen die einzelnen Komponenten des Systems strikt getrennt voneinander laufen. Der L4-Mikrokern kontrolliert die Zugriffe auf alle Ressourcen und stellt sicher, dass eine Kommunikation zwischen den Bereichen nur über wohldefinierte Schnittstellen erfolgen kann. So lassen sich die erlaubten Informationsflüsse beschränken und weniger vertrauenswürdige Komponenten von den restlichen Teilen des Systems separieren.

Bedingt durch die zentrale Sicherheitsrolle des L4-Mikrokerns, ist es unabdingbar, diesen ausführlich auf Schwachstellen zu überprüfen. Dies geschieht einerseits durch manuelle Inspektion der Implementierung [BaBi13], andererseits durch (semi-)automatisierte Penetrationstests [Myer79]. Eine spezielle Klasse solcher Tests sind sogenannte *Fuzzer* [Mill08]. Traditionellerweise wurden diese zur Analyse kleinerer Software-Programme genutzt, bei denen Eingaben potenziell ungültiger Werte Fehler im Programmablauf provozieren sollen, um so Schwachstellen in der Implementierung ausmachen zu können. In einem weiteren manuellen Schritt werden diese dann auf ihre Relevanz überprüft und gegebenenfalls behoben.

In dieser Arbeit erläutern wir wie man Fuzzer nutzen kann, um relevante Systemschnittstellen von L4-Mikrokernen auf Schwachstellen zu testen. Dabei bauen wir auf dem bekannten Fuzzer

P. Schartner · A. Baumann (Hrsg.) · D•A•CH Security 2017 · syssec (2017) 50-57.

AFL [Zale16], sowie der für Linux-Kerne erweiterten Implementierung TriforceAFL [HeNe16] auf und präsentieren die Realisierung eines für L4-Mikrokerne angepassten Fuzzers.

2 Stand der Technik

2.1 Penetrationstests mittels Fuzzing

Fuzzing ist eine relativ alte Technik zum Testen von Implementierungen, die schon Ende der 1980er Jahre von Miller [Mill08] vorgeschlagen wurde. Die Idee dabei ist, ein Programm mit zufälligen Eingaben so lange laufen zu lassen, bis dieses abstürzt. Sobald ein solcher Absturz gefunden wurde, wird in einem weiteren Schritt die Abarbeitung der fraglichen Eingabe manuell studiert, um so etwaige Schwachstellen in der Implementierung zu finden. Das Problem hierbei ist, dass der Suchraum der möglichen Eingaben teils enorm groß ist, sodass die Erfolgswahrscheinlichkeit eines solch „blinden“ Vorgehens sehr gering ist. Daher wurden *geleitete* Techniken entwickelt, welche die Erfolgswahrscheinlichkeit des Fuzzers substanziell erhöhen, indem sie die Wahl der zufälligen Eingaben abhängig von der Abarbeitung vorhergehender Eingaben treffen.

Ein vielversprechender Ansatz ist hierbei der Einsatz *evolutionärer Algorithmen* [FoOW66], welche die Menge der untersuchten Eingaben als *Individuen* betrachtet, welchen jeweils eine *Fitness* zugeordnet werden kann. Dabei werden im Lauf des evolutionären Algorithmus nur diejenigen Individuen weiter betrachtet, die eine hohe Fitness aufweisen, und diese verwendet, um neue Eingaben zu generieren.

Eine bekannte Implementierung dieser Idee ist der Fuzzer *american fuzzy lop* (AFL) [Zale16]. Dort wird als Fitnessfunktion die Code-Abdeckung des zu testenden Programms genutzt. Damit werden solche Eingaben als besonders vielversprechend betrachtet, welche eine besonders Tiefe Bearbeitung der Eingabe erfordern. So wird das untersuchte Programm möglichst umfassend getestet und besonders Grenzfälle der Bearbeitung in die Untersuchung mit einbezogen. Man beachte, dass dabei die Struktur der untersuchten Schnittstelle nicht genutzt wird. Der Fuzzing-Prozess ist daher *unstrukturiert*.

Aufbauend auf AFL wurde der Linux Systemaufruf-Fuzzer *TriforceAFL* entwickelt [HeNe16]. Die Besonderheit hierbei ist die Nutzung der AFL Methodik für hoch komplexe Implementierungen, in diesem Fall einen vollständigen Linux-Kern. Zu deren Realisierung wurde der Linux-Kern in der Emulationsumgebung *QEMU* [QEMU16] gestartet und ein AFL-„Treiber“ als Nutzerprozess realisiert, welcher die AFL-Tests an die relevanten Systemschnittstellen des Linux-Kerns weiterleitet.

Die Übertragung der Erkenntnisse des TriforceAFL Projekts auf L4-Mikrokerne (spezifisch den *L4Re-Mikrokern* [L4Re16]), die ein völlig anderes Systemschnittstellen-Design als Linux verfolgen, war bisher ein nicht untersuchtes Problem, welches wir in dieser Arbeit systematisch analysieren.

Wir sind hier in guter Gesellschaft: Die Möglichkeit, nun erstmals Betriebssystemkerne im laufenden Betrieb zu analysieren, führt momentan zu einer regen Aufmerksamkeit in diesem Gebiet. Für eine Übersicht siehe [John17].

2.2 L4-Mikrokerne

L4-Mikrokerne laufen im privilegierten Modus des Prozessors und stellen den Anwendungen nur die absolut notwendigen Mechanismen bereit, nämlich Abstraktionen für Adressräume und

Prozesse, *Interprozesskommunikation* (IPC), sowie Scheduling. Andere Komponenten, die in monolithischen Systemen – wie Linux – zum Kern gehören, also beispielsweise Gerätetreiber, Dateisysteme und Netzwerkstack, sowie alle Anwendungen, laufen bei L4 im unprivilegierten Modus und müssen für privilegierte Operationen die vom Kern bereitgestellten Schnittstellen nutzen. Der L4Re-Mikrokern implementiert die L4-Mikrokern-API [Lied95] und ergänzt diese um eine feingranulare Berechtigungsvergabe. Dabei werden jeder aktiven Komponente nur genau diejenigen Rechte zugeteilt, die sie benötigt, um ihre Aufgabe zu erledigen.

Grundlegender Schutzbereich im L4Re-Mikrokern ist der *Task*. Jedem Task zugeordnet sind ein Adressraum, eine Menge von I/O-Ports und eine Tabelle von Berechtigungen. In einem Task können mehrere Prozesse ausgeführt werden. Sie teilen sich die Ressourcen des Tasks und kommunizieren ausschließlich über IPC. Über diese Schnittstelle werden alle Systemaufrufe realisiert und Daten an den Mikrokern übergeben.

IPC stellt also genau die Systemschnittstelle dar, an welcher der im Rahmen dieser Arbeit beschriebene L4-TriforceAFL-Fuzzer ansetzt. Eine detailliertere Beschreibung dieser Schnittstelle findet sich im kommenden Abschnitt.

Man beachte, dass wir lediglich Systeme mit einem einzelnen Prozessor behandeln und Interrupts (und damit Kernunterbrechungen) ausschalten. Dadurch werden bestimmte Kernmechanismen inaktiv und können im Rahmen der Penetrationstests zu einem Einfrieren des Kerns führen, welches im Real-Betrieb nicht auftreten würde. Ferner schränken wir damit eine ganze Klasse von Fehlern, die beispielsweise Nebenläufigkeit und Seiteneffekte betreffen, in dieser Arbeit aus.

2.3 Die IPC-Systemschnittstelle

Da der L4Re Mikrokern lediglich über eine einzelne Systemschnittstelle, die schon angesprochene IPC-Systemschnittstelle verfügt, stellt diese einen natürlicher Kandidat für die vorliegenden Untersuchungen dar.

Die Schnittstelle erwartet eine Reihe von Parametern, welche die verschiedenen Funktionalitäten steuern. Da der Fuzzing-Prozess in dieser Arbeit unstrukturiert realisiert wird (das heißt syntaktische und semantische Eigenheiten der IPC-Systemschnittstelle werden nicht explizit betrachtet), ist es für die Anwendung von AFL auf die Schnittstelle eigentlich nicht notwendig, auf die konkrete Bedeutung der einzelnen Parameter einzugehen.

Es stellte sich aber im Laufe der Analysen heraus, dass es auch im Rahmen des unstrukturierten Fuzzings sinnvoll ist, einige (wenige) Bytes mit festen Werten zu versehen, um den evolutionären Prozess von AFL in die richtige Richtung zu leiten: So gibt es einen Parameter `timeout`, welcher im Rahmen eines IPC Aufrufs gesetzt werden kann. Falls `timeout` Null ist, blockiert der IPC Aufruf so lange, bis er eine entsprechende Antwort bekommt. Im Fuzzing-Prozess gibt es aber keine korrespondierenden Kommunikationspartner, sodass bei Null gesetztem `timeout` inkorrekter Weise ein Hängen des Systems von AFL registriert wird und AFL solche Eingaben künftig bevorzugt auswählen würde. Um dies zu vermeiden wurde im gesamten Fuzzing-Prozess der Wert von `timeout` stets auf einen positiven festen Wert gesetzt.

Der resultierende von AFL zu traversierende Suchraum korrespondiert zu den 1056 Bytes, welche der IPC-Schnittstelle als Parameter übergeben werden können. Dieser Raum ist offensichtlich zu groß, um vollständig traversiert zu werden, jedoch konnte die evolutionäre Heuristik von AFL schon oft unter Beweis stellen, dass diese auch in sehr großen Suchräumen relevante Eingaben finden kann [Zale16].

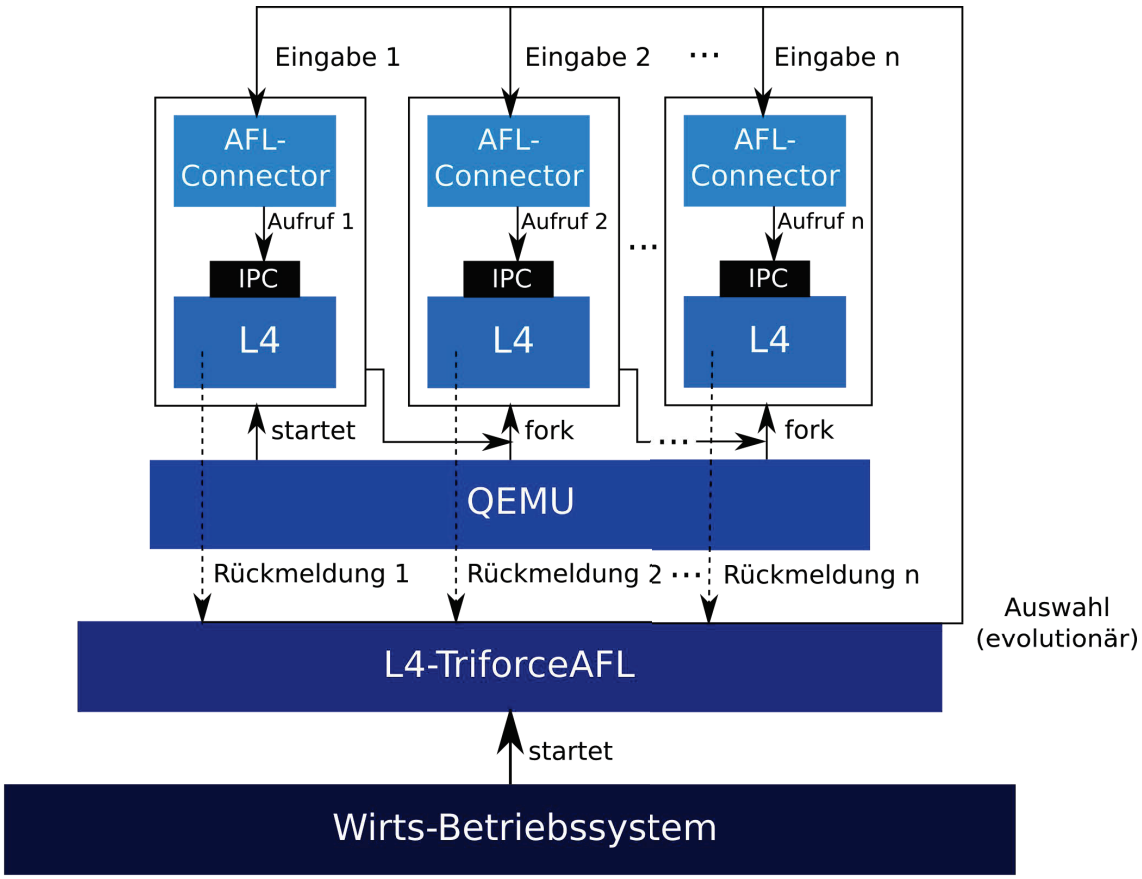

Abb. 1: Architektur des L4-Systemaufruf Fuzzers

3 Fuzzing von Systemaufrufen

3.1 Ablauf des Analyse-Prozesses

Um die IPC Schnittstelle des L4Re-Mikrokerns mit den in AFL realisierten Mechanismen zu verbinden, ist es einerseits notwendig, die Eingaben von AFL zu interpretieren und daraus IPC-Aufrufe zu erzeugen, und andererseits Mechanismen zu realisieren, welche die Abarbeitung der Eingabe zurück an AFL melden.

Die grundlegende Architektur des im Rahmen dieser Arbeit entwickelten L4-TriforceAFL-Fuzzers ist in Abbildung 1 dargestellt.

Auf einem Wirtsbetriebssystem wird AFL initialisiert. Dieses startet seinerseits die Emulationsumgebung QEMU, welche wiederum den L4Re-Mikrokern bootet. Innerhalb des Kerns wird nach dem Start des Systems der Prozess zur Verbindung mit AFL – der sogenannte *AFL-Connector* – zur Ausführung gebracht, welcher über die im TriforceAFL Projekt realisierten Mechanismen die Ansteuerung von AFL innerhalb von L4 realisiert.

Im Detail hat der AFL-Connector die in Abbildung 2 angegebene Struktur. Wesentlich für den Fuzzing-Prozess sind vier Routinen (`startForkserver`, `getWork`, `startWork` und `doneWork`), welche direkt mit dem darunterliegenden QEMU kommunizieren. Die zu untersuchende Schnittstelle selbst ist die im vorherigen Abschnitt beschriebene IPC-Systemschnittstelle in Form der aufrufbaren Funktion `l4_ipc`. Diese wird allerdings nicht

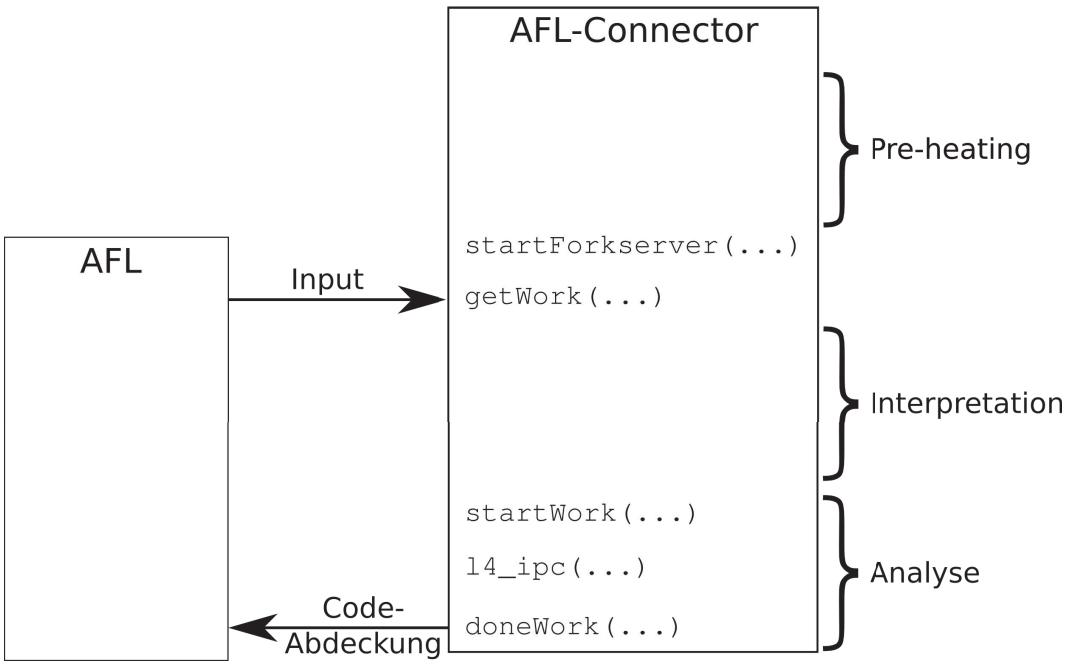

Abb. 2: Struktur des AFL-Connectors

sofort, sondern nach einer Reihe von notwendigen Vorarbeiten aufgerufen, welche die Verbindung von AFL zu L4Re realisieren.

So führt aus Effizienzgründen der AFL-Connector zunächst verschiedene Funktionen des L4Re Kerns aus, um kostenintensive Initialisierungen nicht ständig wiederholen zu müssen. Sobald dieses sogenannte *Pre-heating* abgeschlossen ist, wird im AFL-Connector die Funktion `startForkserver` ausgeführt, welche dem im Rahmen des TriforceAFL-Projekts modifiziertem QEMU mitteilt, welches die definierte Ausgangsposition des Kerns darstellt, die im Folgenden untersucht werden soll. Ausgehend von diesem Zustand werden bei jedem Lauf des L4-TriforceAFL-Fuzzers anschließend separate Kopien des L4Re-Mikrokerns erzeugt und mit Eingaben von AFL versehen. Man beachte, dass dieses Vorgehen völlig zustandslos ist, da der Zustand des zu untersuchenden Kerns bei jedem Fuzzing-Aufruf identisch ist.

Die eigentliche (unstrukturierte) AFL-Eingabe in Form einer festen Anzahl an Bytes erfolgt über den Aufruf der Funktion `getWork`. Dabei liegt es vollständig außerhalb der Kontrolle des AFL-Connectors welche Eingaben hier in das System einfließen, die Auswahl wird ausschließlich von AFL übernommen.

Nun folgt die Interpretation der von AFL übergebenen Bytes. Vereinfacht gesprochen werden diese schlicht als sukzessive Eingabe der IPC-Schnittstelle interpretiert. Falls die Anzahl der von AFL übergebenen Bytes kein Vielfaches der Schnittstellengröße ist, werden die übrigen Eingaben mit Nullen aufgefüllt. In der Realität ist der Interpretationsprozess etwas komplizierter, aus Darstellungsgründen verzichten wir an dieser Stelle jedoch auf Details.

Aus dem Analyseprozess resultiert nun eine Folge von IPC-Aufrufen, welche über die L4Re-Funktion `l4_ipc` an den L4Re-Kern übergeben werden. Bevor dies allerdings erfolgt, instruiert die Funktion `startWork` das TriforceAFL-QEMU, die Nachverfolgung der Code-Abdeckung zu starten.

Anschließend werden die L4Re-Systemaufrufe ausgeführt. QEMU sammelt dabei die notwendige Laufzeitinformation in Form von Tabellen, welche zu der Code-Abdeckung des Systemaufrufs im L4Re-Kern korrespondieren, solange bis die Funktion `doneWork` vom AFL-Connector aufgerufen wird. Die Laufzeitinformationen werden anschließend an AFL weitergegeben, um die evolutionäre Auswahl neuer Eingaben (über `getWork`) entsprechend zu beeinflussen.

Mit dieser Feedback-Schleife ist es möglich, AFL direkt auf die IPC-Schnittstelle des L4Re-Mikrokerns anzuwenden.

3.2 Erzeugung initialer Testeingaben

Die Qualität der Testtiefe des L4-TriforceAFL-Fuzzers hängt stark von den initialen Testeingaben ab, auf deren Basis AFL den Testablauf gestaltet. Dazu ist es notwendig, gültige Aufrufe der IPC-Schnittstelle im laufenden Betrieb des L4Re-Mikrokerns zu erzeugen und diese als Eingabe für AFL abzulegen. Bedingt durch die hierarchische Struktur der Architektur ist es daher notwendig, einen Mechanismus im L4-Userland zu realisieren, welcher die Byte-Folgen der IPC-Schnittstellenaufrufe über QEMU an das Wirtsbetriebssystem weiterleitet. Hierzu wurde der soeben skizzierte Connector-Prozess durch einen Analyse-Prozess ersetzt, welcher im L4Re-Mikrokern eine eigens realisierte Funktionalität aktiviert, die über einen speziellen Kommunikationsport Daten an das Wirtsbetriebssystem weiterzuleiten vermag. Dadurch ist es möglich die IPC Aufrufe einer laufenden L4-Applikation auf dem Wirtsbetriebssystem abzulegen.

Um realistische Aufrufsequenzen zu erzeugen, wurden zwei verschiedene Ansätze verfolgt: Zum einen wurde in der Aufzeichnungsphase ein L4-Linux vom L4Re-Mikrokern gestartet und die dadurch im Mikrokern ankommenden Aufrufe gezielt abgegriffen und gespeichert. Dies erlaubt, eine solide Eingabe-Basis für die IPC-Schnittstelle als Grundlage für die AFL-Analyse zu erzeugen. Zum anderen wurden die IPC-Aufrufe künstlich erzeugt und die dabei verwendeten Werte gespeichert. Dabei wurde der IPC-Aufruf durch eine Funktion ersetzt, welche die Rohdaten in dem passenden, vordefinierten Format ausgab.

4 Durchführung der Penetrationstests

4.1 Testaufbau

Die in Abschnitt 3.1 skizzierte Architektur wurde mit unterschiedlichen Ausführungsstrategien auf einem Virtualisierungssystem mit 10 Intel® Xeon® CPUs 2.20GHz mit jeweils 4 Kernen auf Ubuntu 4.4.0-21-generic aufgesetzt.

Dabei wurden in mehreren unabhängigen Experimenten die im vorherigen Abschnitt beschriebenen Strategien zur Erzeugung initialer Testeingaben genutzt, um AFL sinnvolle Startpunkte im Suchraum der möglichen Eingaben der IPC-Schnittstelle zur Verfügung zu stellen. Anschließend wurde der Fuzzing-Prozess mit 80 parallel laufenden AFL-Instanzen jeweils einige Tage ausgeführt. Im Ganzen wurde so für jedes Experiment jeweils ungefähr ein CPU-Jahr die IPC-Schnittstelle des L4Re-Mikrokerns untersucht und der L4Re Mikrokern dabei grob eine Milliarden mal in QEMU Laufen gelassen. Man beachte, dass es zwar möglich war, die von AFL bereitgestellte Funktionalität zur Synchronisation der parallel laufenden Instanzen zu nutzen, jedoch musste eine umfangreiche Logik realisiert werden, welche eine präzise Steuerung, Überwachung und Interpretation des jeweiligen Experiments ermöglichte. Konkret wurden hierzu verschiedene Start-Skripte realisiert, welche zunächst die initialen Eingaben erzeug-

ten und dann in einer von AFL verarbeitbaren Ablagestruktur speicherten. Anschließend wurde L4-Triforce-AFL parallel ausgeführt.

Die Aufrufe von AFL erwarten dabei insbesondere einen Parameter, welcher angibt wann AFL eine interessante Eingabe gefunden hat, beispielsweise die Position der `panic` Routine im L4Re-Mikrokern. Ein weiter Parameter ist der Speicherbereich über den QEMU die Code-Abdeckung verfolgt. Die Strategie der vorliegenden Untersuchungen war daher, die Parameterwahl so vielfältig wie möglich zu gestalten, d.h. verschiedene Eingabewerte, Abbruch-Kriterien und Speicherbereiche auszuwählen, um somit L4Re möglichst umfangreich zu analysieren.

4.2 Resultate

Im Laufe der Analyse des L4Re-Mikrokerns fand der L4-TriforceAFL-Fuzzer keine Eingaben, die den L4Re-Mikrokern abstürzen ließen. Unsere Untersuchungen zeigten ferner, dass der Fuzzer – wie erwartet – viele der möglichen Abarbeitungspfade abdeckte.

Dabei verhielt sich AFL bei jedem der oben beschriebenen Experimente ähnlich.

Die Ergebnisse lassen zweierlei Schlüsse ziehen: Einerseits bestätigten sie die Intuition, dass kleine dedizierte Mikrokerne mit reduziertem Funktionsumfang ob ihrer Übersichtlichkeit weniger fehleranfällig sind als große Kerne, wie etwa der Linux-Kern. Andererseits stellt dies natürlich keinen Beweis dar, dass der L4Re-Mikrokern keine Schwachstellen aufweist, sondern vielmehr, dass der Ansatz, mit geleitetem evolutionärem (aber unstrukturiertem) Fuzzing Schwachstellen finden zu wollen, im Kontext der IPC-Schnittstelle des L4Re-Mikrokerns nicht erfolgversprechend zu sein scheint.

5 Fazit und Ausblick

In dieser Arbeit wurde erstmals der evolutionäre Fuzzer AFL, sowie das für Linux-Systemaufrufe konzipierte TriforceAFL auf Systemschnittstellen eines L4-Mikrokerns angewandt und eine Architektur realisiert, welche es erstmals erlaubt, einen L4-Kern im laufenden Betrieb mittels evolutionärem Fuzzing zu analysieren. Dies stellt die Grundlage für weitere Untersuchungen dar, um die Sicherheit von L4-Mikrokernen im Detail zu evaluieren.

Eine Möglichkeit die im vorherigen Abschnitt angegebenen Resultat weiter zu verfeinern, wäre die Nutzung eines *strukturierten* Fuzzing-Ansatzes. Statt – wie im Rahmen dieser Arbeit angenommen – L4Re als *grey box* zu betrachten, d.h. abgesehen von der Laufzeitinformation keine weiteren Informationen (wie etwa den Quelltext oder sonstige strukturelle Informationen) für den Fuzzer zu nutzen, ist es denkbar, die Struktur der IPC-Schnittstelle im Detail zu analysieren. Anschließend werden mittels einer formalen Grammatik von AFL nur solche Eingaben erzeugt, welche strukturell genau zu der zu analysierenden Schnittstelle passen. Hierdurch würde einerseits der Suchraum wesentlich verringert und Fuzzing-Barrieren, d.h. Bitfolgen, welche vom Fuzzer zufällig korrekt geraten werden, müssen um Fortschritte zu erzielen, umgangen werden.

Die hier vorgestellte Architektur eignet sich ebenfalls für Tests der IPC-Schnittstelle zwischen zwei L4-Prozessen, welche als Benutzerprozesse des L4Re-Mikrokerns laufen. Ungelöst ist hierbei jedoch die Problematik, dass der Fuzzing-Prozess momentan auf nur einer Seite des Kommunikationskanals sitzt und es nicht klar ist, wie man den entsprechenden Kommunikationspartner sinnvoll realisiert.

Des Weiteren ist ein stark limitierender Faktor die Zustandslosigkeit des zu analysierenden Kerns. In kommenden Arbeiten wäre es denkbar, den zu untersuchenden Ausgangszustand

selbst während des Fuzzing-Prozesses zu modifizieren. Dies ist mit der vorliegenden Architektur jedoch nicht ohne weiteres möglich.

Literatur

[Appl16] Apple: iOS Sicherheit – iOS 9.3 (oder neuer) (2016).

[BaBi13] A. Bacchelli, C. Bird: Expectations, outcomes, and challenges of modern code review. *In: Proceedings of the 35th IEEE/ACM International Conference On Software Engineering (ICSE 2013)* (2013).

[FoOW66] L. J. Fogel, A. J. Owens, M. J. Walsh: Artificial Intelligence through Simulated Evolution. John Wiley & Sons Inc, New York (1966).

[HeNe16] J. Hertz, T. Newsham: TriforceAFL (2016), https://www.nccgroup.trust/uk/about-us/newsroom-and-events/blogs/2016/june/project-triforce-run-afl-on-everything/.

[John17] R. Johnson: Evolutionary Kernel Fuzzing (2017), https://www.blackhat.com/us-17/briefings/schedule/index.html#evolutionary-kernel-fuzzing-7720.

[L4Re16] L4Re: A 3rd generation microkernel (2016), https://os.inf.tu-dresden.de/fiasco/.

[Lied95] J. Liedtke: On μ-Kernel Construction. *In: Proc. 15th ACM Symposium on Operating Systems Principles (SOSP)* (1995), 237?250.

[Mill08] B. Miller: Foreword. *In: A. Takanen, J. DeMott, C. Miller (Hrsg.), Fuzzing for Software Security Testing and Quality Assurance*, Artech House (2008), xv–xvi.

[Myer79] G. J. Myers: The Art of Software Testing. John Wiley & Sons Inc, New York (1979).

[QEMU16] QEMU: Open source processor emulator (2016), http://wiki.qemu.org/Main_Page.

[Zale16] M. Zalewski: american fuzzy lop (2016), http://lcamtuf.coredump.cx/afl/.