Diplomarbeit

Fortgeschrittene Debugging-Tools für L4Re

Christian Prochaska

28. Februar 2011

Technische Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur
Professur Betriebssysteme

Betreuender Hochschullehrer: Prof. Dr. rer. nat. Hermann Härtig
Betreuender Mitarbeiter: Dipl.-Inf. Björn Döbel

Technische Universität Dresden
Fakultät Informatik

AUFGABENSTELLUNG FÜR DIE DIPLOMARBEIT

Name des Studenten:	Christian Prochaska
Studiengang:	Informatik
Immatrikulationsnummer:	2926954

Thema: *Fortgeschrittene Debugging-Tools für L4Re*

Mit der zunehmenden Komplexität moderner Software wachsen auch die Anforderungen an Debugging- und Programm-Analyse-Tools. Zur Analyse der aus dieser Komplexität resultierenden Programmierfehler sind herkömmliche Strategien wie die Analyse von Core Dumps und Backtraces nur bedingt geeignet. Existierende Tools wie GDB und Valgrind konzentrieren sich auf die Analyse einzelner Komponenten. Darüber hinaus sind sie für die Analyse von Userlevel-Anwendungen konzipiert, wodurch ihnen wichtige Konzepte zur Analyse von System-Software (bspw. Gerätezugriffe) fehlen.

In einer vorangegangenen Arbeit wurde das Programm-Analyse-Framework Valgrind auf den Fiasco.OC-Mikrokern und das darauf basierende L4Re-Userland portiert. Ausgehend von dieser Arbeit sollen Möglichkeiten geschaffen werden, komplexere Debugging-Tools für die Analyse von L4Re-Systemkomponenten und deren Interaktionen zu erstellen. Hierzu sollen nach Möglichkeit verfügbare Tools genutzt, erweitert und kombiniert werden. Unter anderem sind hierbei folgende Szenarien denkbar:

- *Debugging von Gerätetreibern*

L4Re verfügt mit dem Device Driver Environment (DDE) über eine Gerätetreiber-Schnittstelle, welche die Verwendung von Linux-Gerätetreibern auf L4Re ermöglicht. Die Ausführung als User-Anwendung auf L4Re ermöglicht die Analyse dieser Komponenten beispielsweise in Valgrind-Tools.

- *(Verteiltes) Record/Replay + Reverse Debugging*

Die Nutzung eines R/R-Tools für einzelne Komponenten ermöglicht das erneute Abspielen eines Anwendungslaufs und unter anderem auch die vom Fehlerzeitpunkt aus rückwärts laufende Ursachen-Analyse. Im Kontext von L4Re sind Anwendungen oft in mehrere kleine Komponenten aufgeteilt und die Ursache von Fehlern liegt ggf. in der Interaktion dieser Komponenten. Deshalb ist zu untersuchen, wie die vorhandenen Tools zur Analyse des Zusammenspiels mehrerer Komponenten genutzt werden können.

- *Debugging von Shared-Memory-Kommunikationsprotokollen*

Shared Memory ermöglicht oft eine schnellere und breitbandigere Datenübertragung zwischen Komponenten als einfache Interprozess- Kommunikation. Die Implementierung solcher Kanäle ist jedoch für Programmierfehler anfällig. Zur Analyse entsprechender Protokolle hinsichtlich Freiheit von Deadlocks und Wettlauf-Situationen ist die Analyse aller beteiligten Komponenten notwendig.

verantwortlicher Hochschullehrer:	Prof. Dr. Hermann Härtig
Betreuer:	Dipl.-Inf. Döbel
Institut:	Systemarchitektur
Beginn:	01. 06. 2010
Einzureichen:	30. 11. 2010

Unterschrift des betreuenden *Hochschullehrers*

Erklärung

Hiermit erkläre ich, dass ich diese Arbeit selbstständig erstellt und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Dresden, 28. Februar 2011

Christian Prochaska

Danksagung

Danke an Björn, Martin und Stefan.

Inhaltsverzeichnis

Abbildungsverzeichnis

Listingverzeichnis

Tabellenverzeichnis

1 Motivation

Bei der Fehlersuche in Anwendungsprogrammen kommt heute in der Entwicklungsphase meist das sogenannte „zyklische Debugging“ zum Einsatz: beim Erkennen einer Fehlerwirkung wird die Ausführung der fehlerhaften Anwendung unterbrochen und man kann mit Hilfe eines interaktiven Debugging-Tools den aktuellen Programmzustand nach Spuren untersuchen die zur Ursache des Fehlers führen. Oftmals sind diese Spuren zum Zeitpunkt des Auftretens der Fehlerwirkung jedoch bereits teilweise oder vollständig verwischt, z.B. weil lokale Variablen auf dem Stack nur während des Aufrufes der entsprechenden Funktion gültig sind und nach deren Rückkehr häufig wieder überschrieben werden. In so einem Fall muss die fehlerhafte Anwendung neu gestartet werden und man hat nun die Möglichkeit, die Ausführung zu beliebigen, aber im Voraus festzulegenden Zeitpunkten durch das Debugging-Tool unterbrechen zu lassen und anschließend jeweils eine erneute Untersuchung des Programmzustandes nach Spuren durchzuführen. Falls die gewählten Unterbrechungszeitpunkte nicht zu den gewünschten Erkenntnissen führen, sind weitere Programmneustarts mit einer neuen Auswahl von Haltepunkten notwendig.

Diese Vorgehensweise kann in vielen Fällen zum Erfolg führen, sie gelangt jedoch an ihre Grenzen, wenn das Auftreten einer Fehlerwirkung von externen Ereignissen wie z.B. über eine Netzwerkschnittstelle empfangenen Paketen mit bestimmtem Inhalt oder von bestimmten zeitlichen Empfangsreihenfolgen abhängt, die sich durch ein bloßes Rücksetzen des Programmes auf dessen Ausgangszustand nicht wiederherstellen lassen. In solchen Fällen müsste eigentlich die Umgebung des Programmes ebenfalls auf den ursprünglichen Zustand zurückgesetzt werden, um eine exakte Reproduzierbarkeit des fehlerhaften Programmablaufes zu gewährleisten. Dies ist jedoch in der Praxis meist nicht möglich und die weitere Suche nach der Fehlerursache wird somit oft zum Glücks- und Geduldsspiel.

Programmneustarts sind also nur bedingt geeignet, verloren gegangene Informationen eines bestimmten fehlerhaften Programmablaufes wieder herzustellen. Viel besser wäre es, wenn diese Informationen gar nicht erst verloren gingen. Dazu müssen sie während der Ausführung des Programmes erfasst und z.B. in einer Log-Datei gespeichert werden, die bei der anschließenden Fehlersuche zusätzlich zum aktuellen Programmzustand analysiert werden kann. In vielen Anwendungen kommt ein solches „Tracing“ bereits zum Einsatz. Zu den erfassten Informationen gehören üblicherweise aktuelle Belegungen ausgewählter Variablen oder Ereignisse wie Funktionsein- und -austritte oder das Auftreten von Fehlersituationen. Die Erfassung und Speicherung der gewünschten Informationen wird dabei durch Instrumentierung

des Programmquellcodes vorgenommen, wobei dieser mit zunehmender Anzahl von Datenerfassungsstellen auch zunehmend unübersichtlicher wird.

In den letzten Jahren wurde verstärkt an neuen Debugging-Tools geforscht, die es ermöglichen sollen, bei vertretbarem Zeit- und Speicheraufwand umfangreiche Programm-Traces zu erzeugen, ohne dafür den Quellcode des zu untersuchenden Programmes instrumentieren zu müssen. Für einige dieser Tools wurden Schnittstellen zu den bisher für das zyklische Debugging eingesetzten Werkzeugen geschaffen, die es dadurch ermöglichen, die zu untersuchende Anwendung scheinbar rückwärts ablaufen zu lassen und somit ausgehend vom Programmzustand beim Auftreten einer Fehlerwirkung Schritt für Schritt den Weg zur Fehlerursache zurückzuverfolgen, ohne das Programm erneut starten zu müssen. Diese Art des Debuggings ist weitaus zielführender und komfortabler durchzuführen als zyklisches Debugging und deshalb auch für den Einsatz in dem an der TU Dresden entwickelten L4 Runtime Environment (L4Re)[1][2] interessant. In Kapitel 6 stelle ich eine Auswahl dieser neuen Debugging-Tools genauer vor.

Neben dem interaktiven Debugging kommen heute zunehmend auch automatisierte Analysetools zum Einsatz. Valgrind[3] ist ein Framework für den Bau solcher Analysetools und wurde bereits auf L4Re portiert. Valgrind-Tools können jeweils verschiedene Analysen während der Ausführung eines Programmes durchführen, allerdings kann auf Grund der Architektur des Frameworks pro Programmlauf nur ein Valgrind-Tool aktiv sein. Dadurch tritt auch hier das Problem der eingeschränkten Reproduzierbarkeit auf Grund von notwendigen Programmneustarts auf, wenn man einen bestimmten fehlerhaften Programmablauf mit verschiedenen Valgrind-Tools untersuchen möchte. Hier ist eine einmalige Aufzeichnung des Programmablaufes für anschließende Mehrfachanalysen ebenfalls wünschenswert.

Mit Valgrind-RR[4] und Chronicle[5][6] existieren zwei Valgrind-basierte Lösungen für Linux, die eine solche Aufzeichnung eines Programmablaufes für spätere Analysen mit verschiedenen Valgrind-Tools (Valgrind-RR) bzw. Rückwärts-Debugging mit dem interaktiven Debugger GDB[7] (Chronicle) ermöglichen. Ziel dieser Arbeit ist es, diese beiden Lösungen an die speziellen Gegebenheiten von L4Re anzupassen und ihre praktische Nutzbarkeit insbesondere für das Debugging von Linux-Gerätetreibern auf L4Re zu untersuchen.

2 Gliederung der Arbeit

Die Arbeit ist wie folgt gegliedert: in Kapitel 3 stelle ich die Grundlagen meiner Arbeit dar. In Kapitel 4 beschreibe ich die Anpassungen, die ich an Valgrind-RR und Chronicle vorgenommen habe, damit sie auf L4Re eingesetzt werden können. In Kapitel 5 zeige ich, wie Valgrind-RR und Chronicle für das Debugging von Linux-Gerätetreibern auf L4Re verwendet werden können. In Kapitel 6 stelle ich verwandte Arbeiten vor und in Kapitel 7 fasse ich meine Arbeit zusammen und gebe einen Ausblick auf mögliche Weiterentwicklungen.

3 Grundlagen

3.1 Fiasco.OC

Fiasco.OC[1][8] ist ein an der TU Dresden entwickelter Mikrokern aus der L4-Familie. Ein Mikrokern zeichnet sich dadurch aus, dass er nur die jenigen Abstraktionen enthält, die für die Gewährleistung eines sicheren und effizienten Betriebes unbedingt erforderlich sind. Die wichtigsten Abstraktionen in Fiasco.OC, auf die ich im Folgenden noch genauer eingehen werde, sind:

- Tasks
- Threads
- Capabilities
- IPC-Gates[1]
- Factories

Eine Task umfasst die Ressourcen auf die eine spezifische Sammlung von Threads zugreifen kann wie z.B. einen virtuellen Adressraum, Capabilities und I/O-Ports. Ein Thread ist ein Ausführungskontext der an eine Task gebunden ist und vom Scheduler des Systems zur Ausführung gebracht wird. Tasks und Threads sind Kernobjekte, auf die mittels Capabilities seitens der Anwendungen zugegriffen werden kann. Capabilities stellen Task-lokale Namen für Kernobjekte dar. Nur wer im Besitz einer Capability ist kann auf das Kernobjekt zugreifen. Ein weiteres Kernobjekt ist das IPC-Gate, welches zur Nachrichtenübermittlung zu einem Thread dient. Dieser Thread muss nicht notwendigerweise an die selbe Task gebunden sein wie der Nutzer des IPC-Gates. Bei der Interprozesskommunikation können neben einer festen Anzahl von Bytes zusätzlich Speicherseiten und/oder Capabilities von der Quell- zur Zieltask übertragen werden. Letzteres wird auch als Mapping bezeichnet. Der Empfänger wird dadurch befähigt, diese Ressourcen ebenfalls zu nutzen. Mit Hilfe eines Factory-Kernobjekts können neue Kernobjekte wie Tasks, Threads, IPC-Gates und auch weitere Factory-Objekte erzeugt werden. Eine Factory limitiert die Nutzung des Kernspeichers zur Erzeugung von Kernobjekten.

Für die Nutzung von Kernobjekten bietet Fiasco.OC einen Systemaufruf („invoke capability“) an. Diesem übergibt man die entsprechende Capability zu dem Kernobjekt und die nötigen Daten zu der Aktion welche auf dem Kernobjekt ausgeführt werden soll. Zur Übermittlung dieser Daten zum Kern stellt der Mikrokern neben bestimmten Registern jedem Thread einen sogenannten UTCB[2] zur Verfügung. Der UTCB ist eine virtuelle Speicherregion mit fester Größe welche mit dem Kern bei der Thread-Erzeugung vereinbart wird.

1 IPC steht für „Inter Process Communication“
2 UTCB steht für „User level Thread Control Block“

3.2 L4Re

L4Re ist ein Framework welches, auf Fiasco.OC aufbauend, grundlegende Dienste und Abstraktionen zur Implementierung von Userland-Anwendungen bereitstellt. Dazu gehören:

- Capability-Verwaltung
- Speicherverwaltung
- Anwendungsinitialisierung
- Client-Server-Framework

Die Capability-Verwaltung von L4Re stellt eine Namenszuordnung für Capabilities zur Verfügung, welche der besseren Lesbarkeit dient.

Ein essenzieller Begriff in der L4Re-Speicherverwaltung ist der Dataspace[9]. Ein Dataspace ist die Abstraktion einer Menge von Speicher, auf welchen von L4Re-Anwendungen wie auf einen gewöhnlichen zusammenhängen Speicherbereich zugegriffen werden kann, unabhängig von dessen tatsächlicher Position oder Struktur. In L4Re werden Dataspaces durch Capabilities auf sogenannte Dataspace Manager repräsentiert. Diese Capabilities können von einer Task beispielsweise über eine initiale Capability auf ein Speicherallokator-Objekt bezogen werden. Um einen Dataspace nutzen zu können, muss diesem zuerst von einem sogenannten Region Manager ein virtueller Adressbereich zugeordnet werden. Jede Task besitzt eine initiale Capability auf ein Region Manager-Objekt, welches gleichzeitig der Standard-Pager der Task ist. Das bedeutet, dass alle Page-Faults die im virtuellen Adressraum der Task auftreten, mittels IPC-Aufruf vom Kern an diesen Region Manager geleitet werden. Dieser prüft dann, ob bereits eine entsprechende Zuordnung zwischen der Page-Fault-Adresse und einem Dataspace in seiner Region Map vorhanden ist. Diese Zuordnung eines Dataspaces zu einer virtuellen Adresse der Task muss zuvor von der Task selbst beim Region Manager vorgenommen werden. Ist eine solche Zuordnung vorhanden, leitet der Region Manager den Page Fault an den entsprechenden Dataspace-Manager weiter, welcher dann das Mapping von Speicher an die Page-Fault-Adresse vornimmt.

Bei der Initialisierung einer Anwendung sorgt L4Re dafür, dass diese bereits Capabilities auf grundlegende Objekte besitzt und dass diese Objekte entsprechend vorhanden sind. Dazu zählen Capabilities auf:

- einen initialen Speicherallokator
- eine Factory
- eine Debug-Konsole
- weitere anwendungsspezifische Objekte (z.B. virtuelle Gerätebusse)

L4Re-Anwendungen die Dienste für andere Anwendungen anbieten werden Server genannt. Anwendungen die diese Dienste nutzen heißen Clients. Damit ein Client die Dienste eines Servers nutzen kann, erzeugt der Server für jeden Client ein IPC-Gate und gibt diesem die dazugehörige Capability.

Gerätetreiber werden in L4Re im deprivilegierten Prozessormodus ausgeführt. Die verfügbaren I/O-Ressourcen werden von einem I/O-Server verwaltet. Dieser bietet seinen Clients virtuelle Gerätebusse mit ausgewählten Geräten an, die jeweils in einer Konfigurationsdatei festgelegt werden. Damit ein Client auf Geräte dieses virtuellen Busses zugreifen kann, bekommt er die Zugriffsrechte auf die entsprechenden I/O-Ports und I/O-Speicherbereiche vom Kern gemappt. Interrupts können mit Hilfe eines speziellen IRQ-Kernobjektes behandelt werden.

Mit dem Device Driver Environment (DDE)[10] besteht eine Möglichkeit, unveränderte Linux-Gerätetreiber in L4Re verwenden zu können. Dies wird durch eine Simulation der von den Treibern erwarteten Linux-Umgebung realisiert.

3.3 Valgrind

Valgrind ist ein Framework für die Erstellung von Debugging-Tools. Mit Valgrind-basierten Debugging-Tools können binäre Programme zur Laufzeit automatisiert analysiert und instrumentiert werden. So kann z.B. mit dem Tool `memcheck`[11] festgestellt werden, ob das untersuchte Programm auf Speicher zugreift der vorher nicht alloziert oder initialisiert wurde. Weitere Valgrind-basierte Debugging-Tools sind der Cache-Profiler `cachegrind` und der Thread-Debugger `helgrind`, welcher Synchronisationsprobleme beim gemeinsamen Zugriff auf Daten durch mehrere Threads entdecken kann.

Ein Valgrind-Tool ist ein statisch gelinktes Programm das sich aus dem Valgrind-Kern und einem Tool-Plugin zusammensetzt (siehe Abbildung 1). Der Valgrind-Kern lädt das zu untersuchende Programm vor dessen Ausführung in seinen Adressraum und übersetzt es anschließend blockweise (jeweils bis zur nächsten Verzweigungsinstruktion) mit Hilfe der VEX-Bibliothek in einen binären Zwischencode, der aus weniger komplexen Instruktionen aufgebaut ist und somit leichter analysiert werden kann. Anschließend wird eine Funktion des Tool-Plugins aufgerufen, die nun die Möglichkeit hat, den Zwischencode zu analysieren und bei Bedarf auch zu verändern. Nach dem Abschluss der Instrumentierungsphase wird der Zwischencode mit Hilfe der VEX-Bibliothek wieder in Maschinencode zurückübersetzt, welcher anschließend ausgeführt wird. Diese Vorgehensweise der Übersetzung in einen einfacheren Zwischencode mit anschließender Rückübersetzung in Maschinencode wird auch als „disassemble and resynthesize“ (D&A) bezeichnet.

Eine Besonderheit von Valgrind gegenüber anderen Frameworks für dynamische binäre Instrumentierung ist die Verwendung von sogenannten „shadow values“. Dabei wird jedem Register- und Speicherwert jeweils ein weiterer Wert zugeordnet, der Meta-Informationen über den eigentlichen Wert enthält. memcheck speichert in diesen Schattenwerten beispielsweise die Information, welche Speicherbits alloziert und initialisiert wurden. Da diese Schattenwerte atomar mit den Lese- und Schreibzugriffen auf die eigentlichen Werte geändert werden müssen, um Inkonsistenzen zu vermeiden, serialisiert Valgrind die Ausführung von Threads.

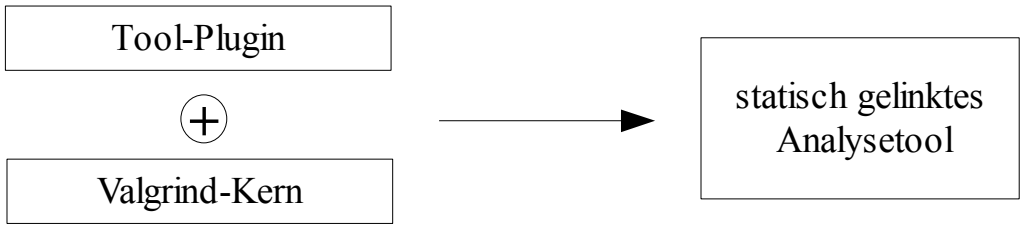

Abbildung 1: Aufbau eines Valgrind-Tools

3.3.1 Valgrind-RR

Valgrind-RR ist eine Modifikation (Patch) für den Valgrind-Kern, die es ermöglicht, den Ablauf eines Client-Programmes aufzuzeichnen und anschließend beliebig oft zu reproduzieren. Dies ermöglicht eine Analyse des aufgezeichneten Programmablaufs mit verschiedenen Valgrind-Tools. Außerdem können Analysen die im Replay-Modus durchgeführt werden beliebig zeitaufwändig sein, da sich die Analysezeit dann nicht mehr auf das Programmverhalten auswirkt.

Da Valgrind darauf ausgelegt ist, die Ausführung eines Programmes von Beginn an zu verfolgen, werden von Valgrind-RR nur die nichtdeterministischen Eingaben während der Programmausführung aufgezeichnet bzw. wiederhergestellt. Die übrigen Programmzustandsdaten ergeben sich automatisch durch die Wiederausführung des Programmcodes.

Die von Valgrind-RR aufgezeichneten Daten umfassen konkret:

- die Startadresse von Valgrinds Speicherbereich im gemeinsam genutzten Adressraum
- die Kommandozeile mit der das Client-Programm aufgerufen wurde
- die Startadresse und den Inhalt des initialen Client-Stacks
- die Rückgabewerte von Linux-Systemaufrufen
- durch Linux-Systemaufrufe veränderte Speicherinhalte
- das Resultat der RDTSC-Instruktion
- die Reihenfolge der Thread-Abarbeitung

Daten die aus Shared Memory gelesen wurden und die Ergebnisse weiterer nichtdeterministischer CPU-Instruktionen wie z.B. der IN-Instruktion zum Lesen von I/O-Ports werden in der bisher nur als „Proof-of-Concept“ veröffentlichten Version von Valgrind-RR nicht berücksichtigt. Es werden auch nicht alle Linux-Systemaufrufe unterstützt, sondern nur eine kleine Auswahl, die von den vom Autor der Modifikation ausgewählten Testprogrammen benötigt wurde. Die unterstützten Linux-Systemaufrufe sind in Tabelle 1 aufgeführt.

access	set_robust_list	futex	lstat64
gettimeofday	get_robust_list	newuname	stat64
read	set_tid_address	times	fstat64
write	clock_settime	gettid	
time	clock_gettime	socketcall	

Tabelle 1: Von Valgrind-RR unterstützte Linux-Systemaufrufe

3.3.2 Chronicle

Chronicle ist ein Valgrind-Tool das alle Register- und Schreibzugriffe des ausgeführten Client-Programms aufzeichnet. Die Aufzeichnungsdaten werden mit einem Zeitstempel (Anzahl der seit dem Programmstart ausgeführten Instruktionen) versehen, indiziert, komprimiert und schließlich in einer Log-Datei gespeichert. Die Log-Datei kann anschließend von anderen Debugging-Tools ausgewertet werden.

Für die Indizierung, Komprimierung und Speicherung der Daten ist ein separater Indexer-Prozess zuständig, der mit dem Chronicle-Tool über Shared Memory und eine Pipe kommuniziert. Dies ermöglicht eine effizientere Nutzung von Mehrkern- und Mehrprozessorsystemen, da Valgrind, wie ich in Kapitel 3.3 erläutert habe, selbst nur die Ausführung eines einzigen Threads erlaubt. Außerdem kann der Indexer-Prozess dadurch auf den vollständigen Umfang der C-Bibliothek zurückgreifen.

Durch die Indizierung der Daten und durch den großen Aufzeichnungsumfang lässt sich der Register- und Speicherzustand eines beliebigen Zeitpunktes der ursprünglichen Ausführung schnell (d.h. nur vom Umfang der Zustandsdaten abhängig und nicht vom gewählten Zeitpunkt), rekonstruieren, ohne dass das Programm dazu erneut ausgeführt werden muss. Dadurch ergeben sich vielfältige Analysemöglichkeiten, auf die ich in den folgenden Unterkapiteln näher eingehen werde.

3.3.2.1 chronicle-query

Um den Entwicklern von Auswertetools für die von Chronicle aufgezeichneten Daten den Zugang zu den gewünschten Informationen zu erleichtern, wurde vom Chronicle-Autor ein spezielles Abfrageprogramm namens chronicle-query entwickelt. Dieses Programm nimmt über seine Standardeingabe Anfragen im

JSON-Format[12] entgegen, extrahiert die gewünschten Daten aus der Log-Datei und liefert die Ergebnisse anschließend ebenfalls im JSON-Format zurück.

Es können unter Anderem folgende Informationen angefragt werden:

- zu welchem Zeitpunkt (entspricht der Anzahl der ausgeführten Instruktionen) endete die Programmausführung?
- zu welchen Zeitpunkten innerhalb eines bestimmten Zeitintervalles wurden welche Instruktionen ausgeführt ?
- zu welchen Zeitpunkten innerhalb eines bestimmten Zeitintervalles wurden welche Speicheradressen mit welche Daten beschrieben?
- zu welchen Zeitpunkten innerhalb eines bestimmten Zeitintervalles wurden welche Funktionen aufgerufen?
- zu welchen Zeitpunkten innerhalb eines bestimmten Zeitintervalles wurden welche Änderungen am Speicherbelegungsplan (Memory Map) durchgeführt?
- welchen Inhalt hatten bestimmte Register zu einem bestimmten Zeitpunkt?
- welchen Inhalt hatten bestimmte Speicheradressen zu einem bestimmten Zeitpunkt?
- welche Argumente wurden der Funktion übergeben, die zu einem bestimmten Zeitpunkt gerade ausgeführt wurde?
- wie waren die lokalen Variablen der Funktion belegt, die zu einem bestimmten Zeitpunkt gerade ausgeführt wurde?

Die Bearbeitung der Anfragen erfolgt asynchron, das heißt es können vom Analysetool mehrere Anfragen direkt hintereinander aufgegeben werden ohne erst das Ergebnis der vorherigen Anfrage abwarten zu müssen. Separate Anfragen werden von chronicle-query in separaten Threads bearbeitet, wodurch eine effiziente Nutzung von Mehrkern- bzw. Mehrprozessorsystemen möglich ist. Um die Ergebnisse anschließend wieder der entsprechenden Anfrage zuordnen zu können, wird jede Anfrage mit einer fortlaufenden Identifikationsnummer versehen, welche in der Antwortnachricht wieder mitgesendet wird. Abbildung 2 zeigt den Ablauf einer Beispielanfrage nach dem Inhalt des Programmzählers und des Stack-Pointers nach der ersten ausgeführten Instruktion.

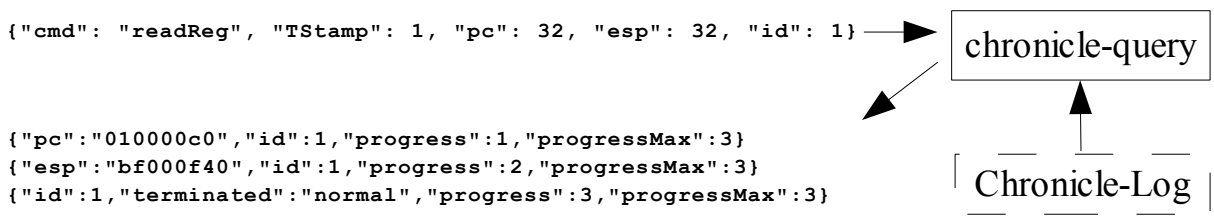

Abbildung 2: chronicle-query-Beispielanfrage

3.3.2.2 Chronomancer

Chronomancer[13] ist ein Plugin für die Eclipse-IDE[14], welches diese zu einer grafischen Benutzeroberfläche für das interaktive Debugging von Anwendungen unter Verwendung der von Chronicle aufgezeichneten Daten erweitert.

Dem Benutzer werden dabei in verschiedenen Ansichtsfenstern hilfreiche Informationen präsentiert:

- die Sourcecode-Ansicht zeigt farblich hinterlegt, welche Zeilen des Sourcecodes ausgeführt wurden
- die „Timeline“-Ansicht zeigt Funktionsaufrufe in ihrer zeitlichen Reihenfolge und mit ihren Aufrufargumenten
- die „Call Stack“-Ansicht zeigt die Funktionsaufrufhierarchie zum ausgewählten Ausführungszeitpunkt
- die „Locals“-Ansicht zeigt die Belegung lokaler Variablen zum ausgewählten Ausführungszeitpunkt
- die „Data“-Ansicht zeigt was sich hinter Zeigervariablen verbirgt

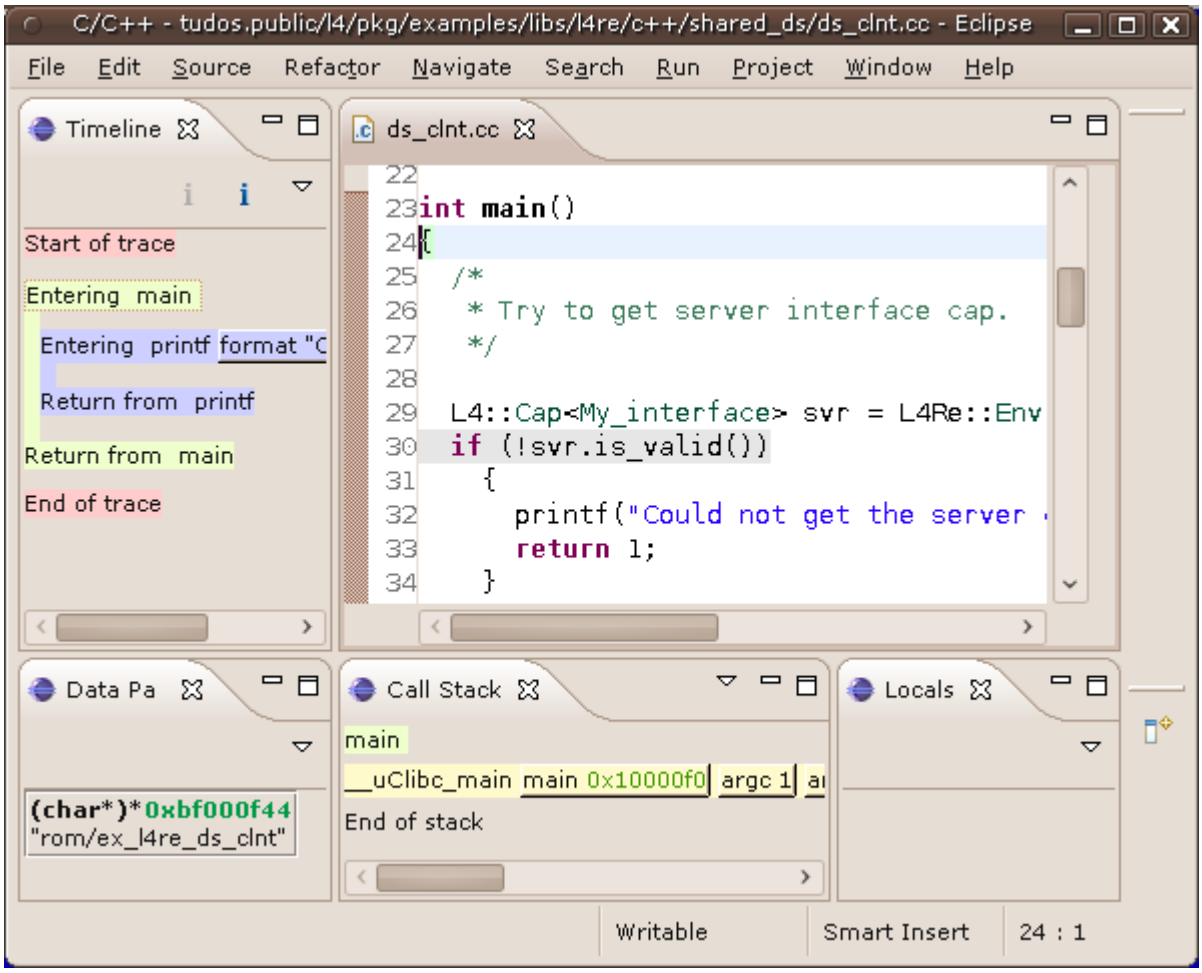

Abbildung 3: Chronomancer

3.3.2.3 chronisole

chronisole[15] ist ein Python[16]-Script, das aus den von Chronicle aufgezeichneten Daten menschenlesbare Funktionsaufruftraces erzeugen kann. Für den Zugriff auf die Chronicle-Daten kommt eine Python-Bibliothek namens chroniquery zum Einsatz. Diese bietet eine objektorientierte Schnittstelle für die Durchführung von Anfragen an chronicle-query an und kann auch von anderen Python-basierten Auswertetools verwendet werden.

chronisole sucht ausgehend von einer oder mehreren als Aufrufargument angegebenen Funktionsnamen nach allen Aufrufen dieser Funktionen und, falls gewünscht, auch rekursiv nach von diesen Funktionen aufgerufenen Unterfunktionen. Durch entsprechende Aufrufargumente und mittels einer Konfigurationsdatei können unter Anderem die Suchtiefe konfiguriert und uninteressante Funktionen von der Suche ausgenommen werden. Das Ergebnis der Suche wird anschließend formatiert und farblich angereichert auf der Standardausgabe ausgegeben. Abbildung 4 zeigt ein Beispiel einer solchen Ausgabe.

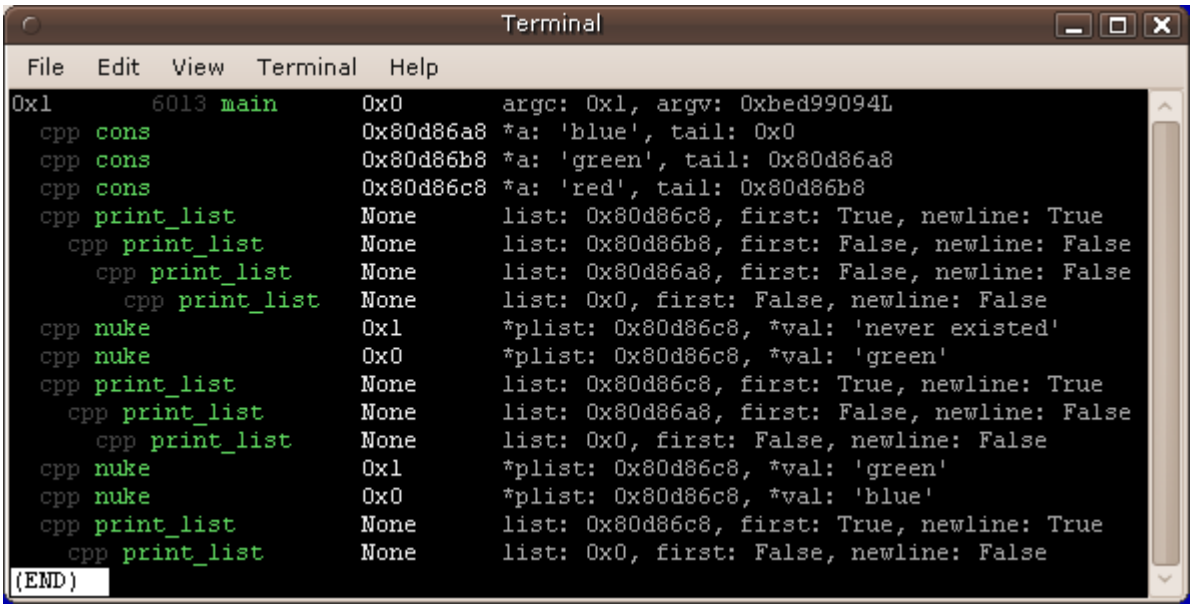

Abbildung 4: Ein von chronisole erzeugter Funktionsaufruftrace

3.3.2.4 chronicle-gdbserver

chronicle-gdbserver[17] ist ein Python-Script, welches es ermöglicht, mit Chronicle aufgezeichnete Anwendungen nach deren Ausführung („offline“) mit dem GNU Project Debugger (GDB) interaktiv zu debuggen. Dazu bietet chronicle-gdbserver eine Serverschnittstelle an, mit der sich GDB verbinden kann, und übersetzt das GDB-Remote-Protokoll in entsprechende JSON-Anfragen für chronicle-query. Für die Interaktion mit chronicle-query kommt auch hier die im vorangegangenen Kapitel erwähnte chroniquery-Bibliothek zum Einsatz.

Neben den üblichen gdb-Funktionen wie z.B. der Einzelschrittausführung oder dem Setzen von Breakpoints und Watchpoints werden von chronicle-gdbserver auch die in GDB Version 7 neu eingeführten Kommandos für die Rückwärtsausführung des untersuchten Programmes unterstützt.

Das Zusammenspiel der einzelnen Komponenten habe ich in Abbildung 5 noch einmal grafisch veranschaulicht.

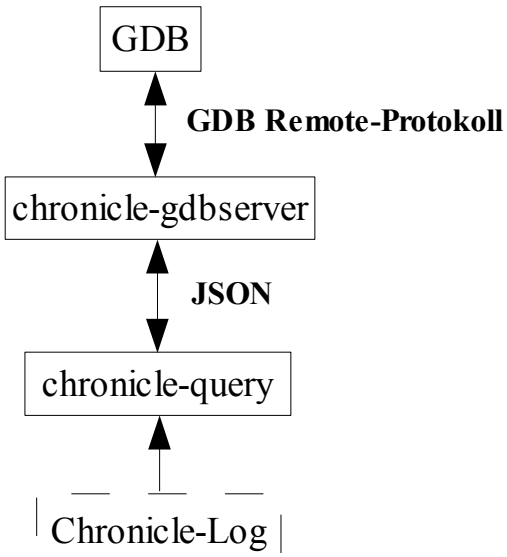

Abbildung 5: chronicle-gdbserver

3.3.3 Kombination von Valgrind-RR mit Chronicle

Da es sich bei Valgrind-RR um eine Modifikation des Valgrind-Kerns und bei Chronicle um ein Valgrind-Tool handelt, lassen sich beide auch problemlos miteinander kombinieren, wie Abbildung 6 zeigt.

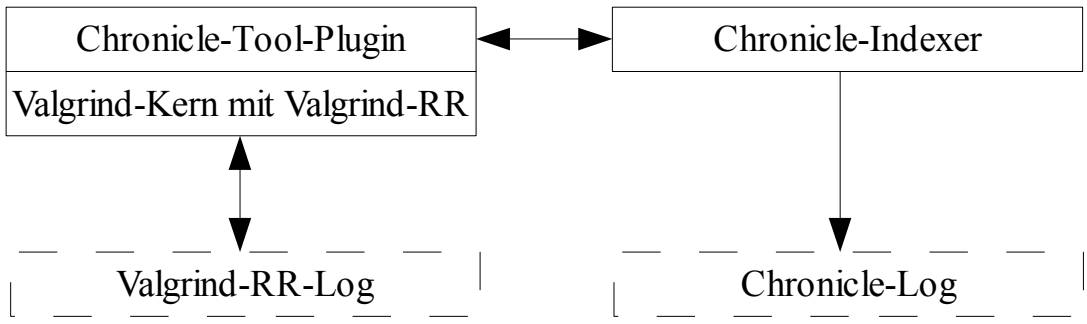

Abbildung 6: Kombination von Valgrind-RR mit Chronicle

Durch diese Kombinationsmöglichkeit kann die zeitaufwändige Erzeugung des Chronicle-Traces in die Replay-Phase von Valgrind-RR verlagert und dadurch das zu untersuchende Programm mit geringerer Verlangsamung in der Record-Phase ausgeführt werden.

4 Anpassungsarbeiten

In diesem Kapitel erläutere ich die Anpassungen die ich vorgenommen habe, um Valgrind-RR und Chronicle auf L4Re nutzen zu können. Dabei stand insbesondere auch die Anwendbarkeit für das Debugging von Gerätetreibern im Vordergrund. Zuerst musste eine Möglichkeit gefunden werden, Aufnahmen persistent speichern zu können. Des Weiteren musste Valgrind-RR zuerst einmal an die in L4Re eingesetzte Version von Valgrind angepasst werden. Anschließend waren einige Änderungen an Valgrind-RR und Chronicle an die Gegebenheiten von L4Re notwendig, auf die ich in den folgenden Unterkapiteln genauer eingehen werde.

4.1 Persistente Speicherung von Aufnahmen

Sowohl Valgrind-RR als auch Chronicle müssen ihre in der Record-Phase aufgezeichneten Daten persistent speichern, damit diese in den Replay- bzw. Analysephasen weiterhin zur Verfügung stehen. In L4Re wird derzeit jedoch keine persistente Speicherung von Daten angeboten, da noch keine Gerätetreiber für Massendatenspeicher wie Festplatten oder USB-Sticks vorhanden sind.

Um die Aufzeichnungsdaten dennoch speichern zu können, habe ich mir den Umstand zu Nutze gemacht, dass Textausgaben auf die Fiasco.OC-Log-Konsole durch Setzen einer Kernoption an die serielle Schnittstelle weitergeleitet und dadurch auf einem angeschlossenen Linux-Host gespeichert werden können. Auf diesem Wege können auch binäre Daten übertragen werden, wenn diese so codiert werden, dass sie einerseits die Ausgabe der anderen Log-Meldungen auf einem Linux-Terminal nicht stören und sie andererseits auch zuverlässig aus der Menge aller Log-Ausgaben extrahiert werden können, um sie in einer separaten Datei speichern zu können. Eine solche geeignete Codierung ist z.B. die UU-Codierung[18], die binäre Daten in „lesbare“ ASCII-Zeichen umwandelt und den Beginn und das Ende der codierten Daten mit einer speziellen Textsequenz kennzeichnet.

Valgrind-RR verwendet zum Speichern der Log-Dateien die von Valgrind angebotenen Funktionen `VG_(open)`, `VG_(write)` und `VG_(close)`. Diese Funktionen führen in der Linux-Version von Valgrind die entsprechenden Linux-Systemaufrufe aus. In L4Re dagegen werden die POSIX-Funktionen `open()`, `write()` und `close()` aufgerufen. Diese Funktionen werden auch vom Chronicle-Indexer verwendet. Sie sind in einer separaten Bibliothek (libl4re-vfs) implementiert, die ein virtuelles Dateisystem bereitstellt. Für die endgültige Speicherung der Daten sind sogenannte „Backends“ der VFS-Bibliothek zuständig. Eines dieser Backends, das sogenannte „tmpfs“-Backend, speichert die ihm übergebenen Daten temporär in einer Datenstruktur im Arbeitsspeicher. Auf Grundlage dieses Backends habe ich ein neues Backend erstellt, das die im

Arbeitsspeicher zwischengespeicherten Dateien nach Aufruf der close()-Funktion UU-codiert auf der Konsole ausgibt.

4.2 Anpassung von Valgrind-RR an die aktuelle Valgrind-Version

Valgrind-RR wurde im Oktober 2008 als "Proof-of-Concept"-Patch für die damals aktuelle Valgrind-Version 3.3.1 auf der Valgrind-Mailingliste veröffentlicht. Da seitdem keine weitere Version des Patches veröffentlicht wurde, musste dieser zuerst einmal an die in L4Re eingesetzte Valgrind-Version 3.6.0-svn angepasst werden. Die Anpassungen bestanden größtenteils aus der Übergabe zusätzlicher Argumente an Funktionen deren Signatur erweitert wurde sowie aus Änderungen beim Zugriff auf Variablen deren Typdefinition inzwischen verändert wurde.

4.3 Anpassung von Valgrind-RR an L4Re

Die ursprüngliche Version von Valgrind-RR zeichnet neben Informationen zum initialen Client-Zustand die Rückgabewerte und Speicheränderungen einiger Linux-Systemaufrufe auf sowie das Resultat der RDTSC-Instruktion und die Reihenfolge der Thread-Abarbeitung. Der Code für die Aufzeichnung des Resultates der RDTSC-Instruktion konnte ohne Änderung beibehalten werden, bei den anderen Punkten waren Anpassungen an die Gegebenheiten von L4Re notwendig. Um Valgrind-RR für die beabsichtigte Untersuchung von Linux-Gerätetreibern in L4Re einsetzen zu können, habe ich außerdem die Funktionalität von Valgrind-RR um die Berücksichtigung von Shared Memory und I/O-Instruktionen erweitert. Auf die einzelnen Änderungen werde ich in den folgenden Unterkapiteln detailierter eingehen.

4.3.1 Systemaufrufe

Valgrind kann Systemaufrufe an den entsprechenden INT- bzw. SYSENTER-Instruktionen im Maschinencode des Clients erkennen. Da die von den Systemaufrufen ausgeführten Register- und Speicheroperationen nicht Teil des Client-Codes sind, muss das Tool-Plugin gesondert darüber informiert werden, welche Register und Speicherbereiche von dem jeweiligen Systemaufruf gelesen oder geschrieben werden. Für diesen Zweck gibt es in Valgrind eine interne Tabelle, in der für jeden Systemaufruf jeweils eine Funktion eingetragen werden kann, die vor der Ausführung des Systemaufrufs aufgerufen werden soll (so genannter „PRE-Wrapper“) und eine Funktion die nach der Ausführung des Systemaufrufs aufgerufen werden soll (so genannter „POST-Wrapper“).

Üblicherweise informiert der PRE-Wrapper das Tool-Plugin darüber, welche Register und Speicherbereiche der Systemaufruf lesen wird und der POST-Wrapper informiert das Tool-Plugin darüber, welche Register und Speicherinhalte von dem Systemaufruf geschrieben wurden. Dieses Wissen stammt aus der Dokumentation oder dem Quellcode des Systemaufrufes und muss von den Valgrind-Entwicklern manuell auf dem aktuellen Stand gehalten

werden. Manche PRE-Wrapper verändern auch zusätzlich die Argumente des Aufrufes, z.B. den Einsprungspunkt eines neuen Threads bei dessen Erzeugung mit dem `clone()`-Systemaufruf, um nicht die Kontrolle über die Ausführung zu verlieren.

Valgrind-RR erweitert diese Funktionstabelle um einen Eintrag für einen sogenannten RECORDREPLAY-Wrapper. Diese Funktion wird im Record-Modus nach der Ausführung des Systemaufrufs und vor dem Aufruf des POST-Wrappers aufgerufen und ist für die Aufzeichnung der nichtdeterministischen Register- und Speicheränderungen zuständig. Im Replay-Modus wird der Systemaufruf selbst nicht mehr ausgeführt, statt dessen führt nun der RECORDREPLAY-Wrapper die im Record-Modus aufgezeichneten Änderungen des Client-Zustandes selbst durch.

Linux bietet mehr als 300 verschiedene Systemaufrufe an. Gemeinsam ist allen Linux-Systemaufrufen auf der x86-Plattform, dass alle Aufrufargumente in Registern übergeben werden und dass im EAX-Register ein Ergebniswert zurückgeliefert wird. Die Aufrufargumente können auch Zeiger auf Speicherbereiche sein, die von dem Systemaufruf gelesen oder beschrieben werden können. Die RECORDREPLAY-Funktion speichert deshalb bei Linux-Systemaufrufen grundsätzlich den Rückgabewert und bei den jenigen Systemaufrufen die Speicherbereiche verändern auch den neuen Inhalt dieser Speicherbereiche in der Log-Datei.

Fiasco.OC bietet insgesamt zwei Systemaufrufe an, wovon einer nur zur Ansteuerung des Kerndebuggers dient und keine nichtdeterministischen Effekte auf den Client-Zustand hat und deshalb in dieser Arbeit nicht weiter berücksichtigt wird.

Der für Valgrind-RR relevante Systemaufruf „invoke capability“ ermöglicht den Aufruf der Methoden von Kernobjekten. Die Aufrufargumente und auch mögliche Rückgabewerte werden dabei im UTCB abgelegt. Die Register werden für die Übertragung des Capability-Indexes des aufzurufenden Kernobjektes und für die Übertragung des sogenannten „Message Tags“ verwendet, welches unter Anderem Meta-Informationen über den aktuellen UTCB-Inhalt enthält. Der Inhalt des UTCBs und das Message Tag sind somit nach der Ausführung des Systemaufrufs als nichtdeterministische Eingaben zu betrachten und müssen entsprechend im Record-Modus in der Log-Datei gespeichert und im Replay-Modus aus der Log-Datei wiederherstellt werden. Diese Aufgabe wird von einer RECORDREPLAY-Funktion übernommen die ich für den „invoke capability“-Systemaufruf erstellt habe.

Einige Systemaufrufe müssen allerdings auch im Replay-Modus ausgeführt werden, da diese Nebenwirkungen haben die über Register- und Speicherinhaltsänderungen hinausgehen. In Linux betrifft das beispielsweise den `clone()`-Systemaufruf für die Erzeugung eines neuen Threads. Wenn ein

bestimmter Systemaufruf unabhängig von den Argumenten immer ausgeführt werden soll, braucht man einfach keine RECORDREPLAY-Funktion für diesen Systemaufruf anzugeben. Bei anderen Systemaufrufen möchte man aber vielleicht die Ausführung des Aufrufs im Replay-Modus von den Aufrufargumenten abhängig machen. Zum Beispiel ist es nützlich, den `write()`-Systemaufruf auszuführen, wenn der Dateideskriptor dem der Standardausgabe entspricht, um an Hand der Konsolenausgaben den Fortschritt der Programmwiederausführung beobachten zu können. Für diesen Fall gibt es in Valgrind-RR eine Funktion `should_not_record_replay()` die vor jeder Ausführung eines Systemaufrufs aufgerufen wird und die Aufrufargumente des Systemaufrufs auswerten kann. Wenn die Auswertung ergibt, dass dieser konkrete Systemaufruf auch im Replay-Modus ausgeführt werden soll, gibt die Funktion den Wert TRUE zurück und der Record-/Replay-Mechanismus wird anschließend bei diesem Aufruf nicht angewendet.

In L4Re gibt es ebenfalls Systemaufrufe die auch im Replay-Modus ausgeführt werden müssen oder sollen. Da diese eine Teilmenge der „invoke capability“-Systemaufrufe darstellen, habe ich entsprechende Filterbedingungen in der `should_not_record_replay()`-Funktion hinzugefügt. Die betroffenen Systemaufrufe habe ich in Tabelle 2 aufgeführt.

Aufgabe des Systemaufrufs	Erkennungsmerkmal
Dataspace-Erzeugung (Speicherallokation)	- „invoke capability“-Systemaufruf - FACTORY-Protokoll - DATASPACE-Subprotokoll
Adressraumverwaltung	- „invoke capability“-Systemaufruf - RM-Protokoll
Threadverwaltung	- „invoke capability“-Systemaufruf - THREAD-Protokoll
Textausgabe auf der Log-Konsole (aus Bequemlichkeit, zur optischen Verfolgung des Programmablaufs)	- „invoke capability“-Systemaufruf - LOG-Protokoll

Tabelle 2: Systemaufrufe die auch im Replay-Modus ausgeführt werden

4.3.2 Shared Memory

Daten die aus Shared Memory gelesen werden müssen ebenfalls als nichtdeterministische Eingaben betrachtet und deshalb aufgezeichnet und wiederhergestellt werden. In der Linux-Version von Valgrind-RR ist diese Funktionalität nicht implementiert, weshalb ich mir dafür eine neue Lösung ausgedacht habe.

Um Speicherlesezugriffe des Clients erkennen zu können, muss der Zwischencode analysiert werden. Um Analysen des Zwischencodes zum

Zwecke der Aufzeichnung zu ermöglichen, hat Valgrind-RR eine weitere Instrumentierungsphase vor der Instrumentierungsphase des Valgrind-Tools eingefügt. Da das Speichern und Wiederherstellen von Speicherlesezugriffen teuer ist, ist es wichtig, dass nach Möglichkeit wirklich nur die Lesezugriffe berücksichtigt werden die sich auf Shared Memory beziehen. Auf die Erkennung von Shared Memory-Bereichen werde ich im Folgenden noch etwas genauer eingehen, da dafür einiger Aufwand notwendig ist.

In L4Re ist jeder gültigen Speicheradresse ein Dataspace zugordnet. Dataspaces sind Objekte die zusammenhängende Speicherbereiche repräsentieren. Sie können mit Hilfe des initialen Speicherallokator-Objektes erzeugt werden. Um auf den Inhalt eines Dataspaces zugreifen zu können, muss dieser vorher vom Region Manager-Objekt in den Adressraum der Task „eingehängt“ werden. Ein Dataspace gilt dann als Shared Memory, wenn mehrere Tasks eine Capabiliy für das Dataspace-Kernobjekt besitzen - wenn also eine Task den Dataspace mit Hilfe des Factory-Kernobjektes erzeugt und anschließend die Capability für diesen Dataspace an andere Tasks weitergibt.

Mit der L4Re-Funktion `l4re_rm_find()` lässt sich für eine gegebene Speicheradresse der Capability-Index des dazugehörigen Dataspace-Kernobjektes ermitteln. Ein Dataspace-Kernobjekt gibt allerdings keine Auskunft darüber, ob es sich bei dem Dataspace um Shared Memory handelt. Aus diesem Grund versuche ich, diese Information an Hand bestimmter Ereignisse während der Ausführung des Client-Programmes selbst zu gewinnen. Für die Speicherung der Information habe ich eine Map-Datenstruktur angelegt, die dem Capability-Index eines Dataspaces einen Boolschen Wert `is_shared` zuordnet der darüber Auskunft gibt, ob der Dataspace als Shared Memory gelten soll oder nicht. Zu Beginn der Programmausführung enthält diese Datenstruktur keine Einträge. Wenn während der Ausführung des Client-Programmes der Systemaufruf für die Erzeugung eines neuen Dataspaces erkannt wird, wird der Capability-Index des neuen Dataspaces als „nicht shared“ in die Map eingetragen. Die Map enthält somit auf jeden Fall alle Dataspaces die von der Task selbst erzeugt wurden. Wenn bei einem späteren Systemaufruf erkannt wird, dass eine Capability die in der Map enthalten ist an eine andere Task weitergemappt werden soll, wird diese Capability in der Map als „shared“ markiert. Nun müssen noch die Dataspace-Capabilities behandelt werden, die von anderen Tasks empfangen wurden. Dazu wird bei jedem Systemaufruf für das Einhängen eines Dataspaces in den Adressraum der Task überprüft, ob der Capability-Index des einzuhängenden Dataspaces in der Map enthalten ist. Wenn das nicht der Fall ist, stammt die Capability offenbar von einer anderen Task und wird deshalb als „shared“ in die Map eingetragen.

Dataspaces, die von anderen Tasks erzeugt wurden, müssen darüber hinaus noch gesondert behandelt werden, da die anderen Tasks im Replay-Modus möglicherweise gar nicht mehr existieren und somit auch nicht den Dataspace

erzeugen und die Capability weitergeben können. Aus diesem Grund wird in dem Fall dass der Capability-Index noch nicht in der Map enthalten ist im Record-Modus noch die Größe des Dataspaces in der Log-Datei gespeichert und im Replay-Modus ein neuer Dataspace mit dieser Größe erzeugt.

Für die Erkennung der Speicherlesezugriffe sucht mein neu hinzugefügter Instrumentierungscode im Zwischencode nach Statements die ein temporäres Register beschreiben (`Ist_WrTmp`) und die zu schreibenden Daten dabei aus einem „Load“-Ausdruck (`Iex_Load`) beziehen. Wenn ein solches Statement gefunden wurde, wird vor dem ursprünglichen Statement ein Statement zum Aufruf einer Funktion `trace_pre_mem_load()` und nach dem ursprünglichen Statement ein Statement zum Aufruf einer Funktion `trace_post_mem_load()` bzw. `trace_post_mem_load64()` eingefügt. Die Funktion `trace_post_mem_load()` bekommt die Speicheradresse, die gelesenen Daten und die Anzahl der gelesenen Bytes (1, 2 oder 4) übergeben. Da auch 64-Bit-Lesezugriffe vorkommen können und an Instrumentierungsfunktionen aber nur 32-Bit-Argumente übergeben werden können, habe ich für diesen Fall die Funktion `trace_post_mem_load64()` erstellt, die die gelesenen Daten aufgeteilt auf zwei 32-Bit-Argumente übergeben bekommt. Beide Funktionen haben die Aufgabe, im Record-Modus zu erkennen ob die übergebene Speicheradresse zu einem Shared Memory-Bereich gehört und in diesem Fall die Adresse, die gelesenen Daten und die Anzahl der gelesenen Bytes in der Log-Datei zu speichern. Die `trace_pre_mem_load()`-Funktion bekommt die Speicheradresse und die Anzahl der zu lesenen Bytes übergeben und schreibt entsprechend im Replay-Modus die in der Log-Datei gespeicherten Daten an die übergebene Adresse, falls diese zu einem Shared Memory-Bereich gehört.

4.3.3 I/O-Instruktionen

Für den Zugriff auf Hardware-Geräte existiert bei x86-CPUs ein spezieller I/O-Adressraum. Die Adressen dieses Adressraumes werden als I/O-Ports bezeichnet und für den Zugriff auf diese sind die speziellen Instruktionen IN und OUT vorgesehen. Mit der IN-Instruktion können Daten aus Hardware-Controllern gelesen werden, mit OUT können Daten an Hardware-Controller gesendet werden. Daten die aus Hardware gelesen werden sind oft nichtdeterministisch und müssen deshalb aufgezeichnet und wiederhergestellt werden. Da die Instruktionen CPU-spezifisch sind, gibt es keine entsprechenden Zwischencode-Instruktionen. Stattdessen findet man im Valgrind-Zwischencode Statements für Aufrufe von plattformspezifischen C-Funktionen die die Instruktionen direkt ausführen. Um das Resultat der IN-Instruktion aufzeichnen und wiederherstellen zu können suche ich deshalb im Zwischencode nach Statements für die Aufrufe der für die IN-Instruktion zuständigen plattformspezifischen Funktion und ersetze den Funktionszeiger in diesen Statements durch einen Zeiger auf eine

eigene Funktion, die im Record-Modus die Instruktion aufruft und anschließend das Resultat in der Log-Datei speichert und im Replay-Modus das Resultat der Instruktion aus der Log-Datei wiederherstellt. Bei der OUT-Instruktion ersetze ich ebenfalls den Funktionszeiger durch einen Zeiger auf eine eigene Funktion, in welcher ich dafür sorge, dass die OUT-Instruktion nur im Record-Modus ausgeführt wird. Dadurch kann im Replay-Modus komplett auf die angesteuerte Hardware verzichtet werden. Darüber hinaus speichere ich in beiden Funktionen zu Kontrollzwecken noch die übergebene Port-Adresse und die Anzahl der zu lesenden bzw. schreibenden Bytes in der Log-Datei.

4.3.4 Reihenfolge der Thread-Abarbeitung

Valgrind-RR speichert im Record-Modus die Reihenfolge der ausgeführten Threads und stellt diese im Replay-Modus in der Linux-Version dadurch wieder her, dass bei jedem Thread-Wechsel die Valgrind-interne Thread-ID des vom Kern ausgewählten Threads mit der in der Log-Datei gespeicherten Thread-ID verglichen wird und bei einer Abweichung so lange den `sched_yield`-Systemaufruf ausführt, bis die Thread-IDs übereinstimmen. In der L4Re-Version rufe ich stattdessen die Funktion l4_thread_switch() mit der Thread-Capability des jenigen Threads auf, der der gespeicherten Thread-ID entspricht.

4.4 Anpassung von Chronicle an L4Re

Für die Erfassung der von Chronicle aufgezeichneten Daten waren keine Änderungen notwendig. Ich musste allerdings einige Anpassungen bei der Interaktion des Chronicle-Tools mit dem Chronicle-Indexer-Prozess vornehmen.

Der Indexer-Prozess wird in der ursprünglichen Version vom Chronicle-Tool mit der Funktion VG_(execv) gestartet, die aber für L4Re nicht implementiert ist. Um dieses Problem zu lösen, habe ich das Indexer-Programm so modifiziert, dass es beim Systemstart als L4Re-Server gestartet werden kann und anschließend auf Anfragen des Chronicle-Tools wartet.

Die Datenübertragung vom Chronicle-Tool zum Indexer findet über Shared Memory statt, welches in der Linux-Version durch das gemeinsame Mapping einer temporären Datei sowohl im Chronicle-Tool als auch im Indexer-Prozess angelegt wird. In der L4Re-Version dagegen erzeuge ich den Shared Memory-Bereich dadurch, dass der Indexer-Server einen Dataspace erzeugt und anschließend die Capability für diesen Dataspace an das Chronicle-Tool weitergibt.

Der Shared Memory-Bereich ist in mehrere aufeinanderfolgende Puffer gleicher Größe unterteilt. Zur Synchronisierung der Datenübertragung schreibt der Server in der Linux-Version für jeden freien Puffer dessen Offset im Shared Memory-Bereich in eine Pipe. Wenn das Chronicle-Tool einen neuen freien Puffer benötigt, liest es dessen Offset aus der Pipe. Falls kein Puffer frei ist, blockiert dieser Lesezugriff so lange bis der Indexer-Prozess einen neuen Offset

in die Pipe geschrieben hat. Analog schreibt das Chronicle-Tool für jeden gefüllten Puffer dessen Offset in die Pipe und der Indexer-Prozess bearbeitet diesen Puffer nachdem er den Offset aus der Pipe gelesen hat. Da sowohl vom Server als auch vom Client stets unmittelbar nach der Freigabe eines Puffers ein neuer Puffer angefordert wird, habe ich in der L4Re-Version eine Funktion `release_and_acquire_buffer()` im Indexer-Server implementiert, durch deren entfernten Aufruf das Chronicle-Tool den Server darüber informieren kann dass der zuletzt bearbeitete Puffer gefüllt ist. Die Funktion liefert anschließend den Offset des nächsten freien Puffers zurück und blockiert dabei so lange bis ein Puffer frei ist. Die Synchronisierung der Puffernutzung wird im Indexer-Server mit Hilfe je eines Semaphors für die Anzahl der freien Puffer und die Anzahl der gefüllten Puffer gewährleistet.

5 Debugging von Linux-Gerätetreibern

In diesem Abschnitt zeige ich an Hand einer Beispieluntersuchung wie Valgrind-RR und Chronicle für das Debugging von Linux-Gerätetreibern unter L4Re eingesetzt werden können. Als Testobjekt diente das L4Re-Programm dde_inputtst, welches unter Verwendung von Linux-Gerätetreibern in der DDE-Umgebung endlos alle Maus- und Tastatureingaben registriert und bei jedem Eingabeereignis eine Meldung auf der Log-Konsole ausgibt.

Die Untersuchung bestand aus vier Schritten, auf die ich in den nächsten Kapiteln genauer eingehen werde:

1.) Ausführung von dde_inputtst in Valgrind und Aufzeichnung des Programmablaufs mit Valgrind-RR

2.) Deterministische Wiederausführung von dde_inputtst in Valgrind mit Analyse durch das memcheck-Tool

3.) Deterministische Wiederausführung von dde_inputtst in Valgrind mit Erzeugung eines vollständigen Traces durch das Chronicle-Tool

4.) Erzeugung eines menschenlesbaren Funktionsaufruftraces mit chronisole

In Abbildung 7 habe ich den Ablauf der Untersuchung noch einmal grafisch veranschaulicht.

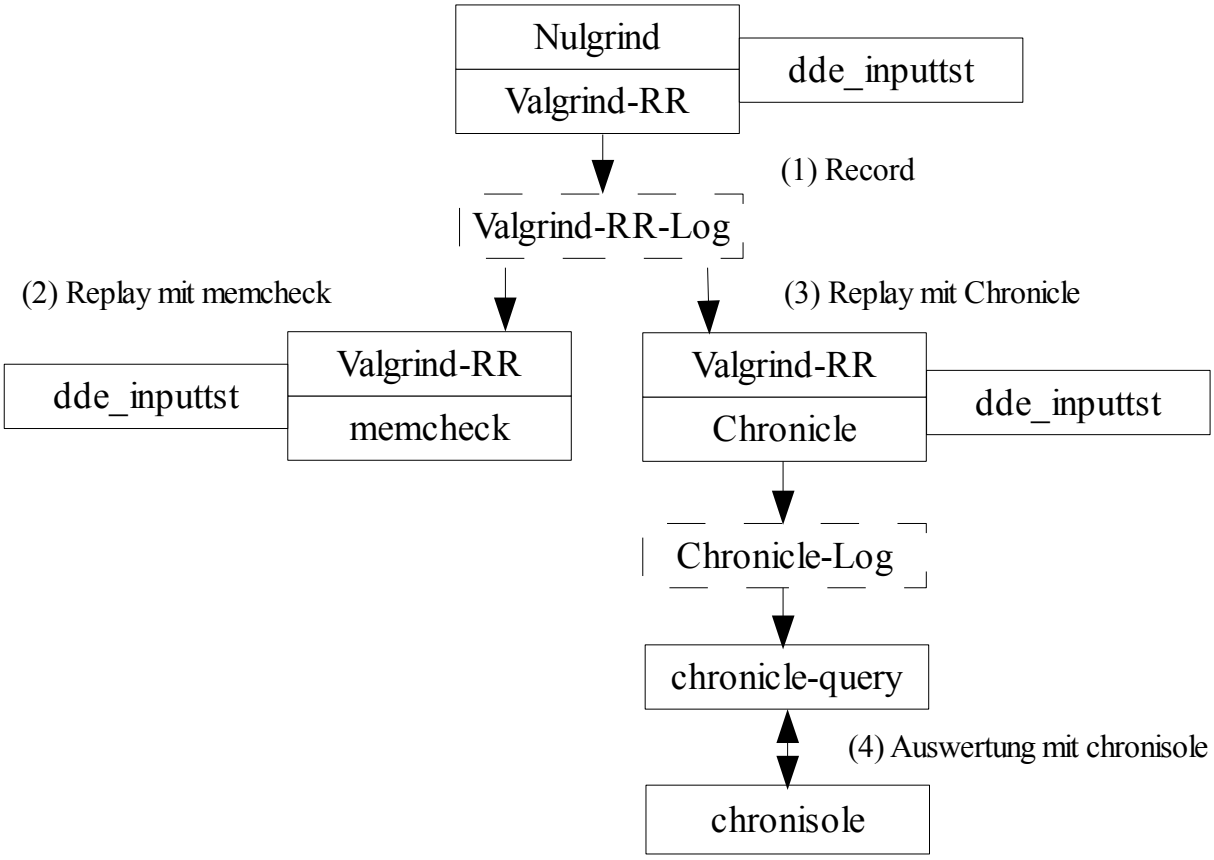

Abbildung 7: Ablauf der Beispieluntersuchung

5.1 Vorbereitungen

Um das Programm dde_inputtst mit memcheck analysieren zu können, musste vor der Ausführung eine spezielle Anpassung vorgenommen werden. Denn um die Allokation und Freigabe von Speicher mit Funktionen wie malloc() und free() analysieren zu können, lenkt Valgrind diese Funktionsaufrufe während der Ausführung auf spezielle Implementierungen dieser Funktionen im memcheck-Tool um. Dafür wird normalerweise beim Programmstart eine Bibliothek mit speziellen Wrapper-Funktionen dynamisch dazugelinkt. Da dde_inputtst jedoch ein statisch gelinktes Programm ist, musste ich eine statische Version dieser Bibliothek erzeugen und diese bereits beim Linken von dde_inputtst mit hinzufügen. Damit auch Aufrufe von kmalloc() und kfree() in die Analyse einbezogen werden können, habe ich für diese Funktionen entsprechende Wrapper-Funktionen in der Bibliothek hinzugefügt, welche momentan ebenfalls die malloc()- und free()-Implementierungen in memcheck aufrufen. Diese können allerdings keine Allokationen von physischem Speicher durchführen, so dass Gerätetreiber mit Bedarf an physischem Speicher, z.B. für DMA-Transfers, derzeit noch nicht mit memcheck analysiert werden können. Da die Wrapper-Funktionen nun statisch in dde_inputtst enthalten waren, musste ich außerdem die von diesen Funktionen aufgerufenen Ersatzfunktionen auch im Nulgrind- und im Chronicle-Tool implementieren.

5.2 Ausführung in Valgrind und Aufzeichnung des Programmablaufs mit Valgrind-RR

Für die Aufzeichnung von dde_inputtst mit Valgrind-RR habe ich die folgende L4Re-Konfigurationsdatei verwendet:

```
package.path = "rom/?.cfg";
require("libadam");
require("VG");

local io_caps  = Alib.io(Alib.config.io);
local l = L4.default_loader;

VG.none({ vbus = io_caps.gui; }, "--record-replay=1",
        "--log-file-rr=tmp/rr.log", "rom/dde_inputtst", "-cb");
```

Listing 1: Konfigurationsdatei für die Aufzeichnung

Die Lua-Funktion `VG.none()` startet Valgrind mit dem Nulgrind-Tool. Das Argument `--record-replay=1` versetzt Valgrind-RR in den Aufzeichnungsmodus und `--log-file-rr=tmp/rr.log` gibt an, wo die Log-Datei von Valgrind-RR gespeichert werden soll.

Da die Log-Datei erst nach Beendigung des ausgeführten Programmes auf der Konsole ausgegeben wird, habe ich das dde_inputtst-Programm so modifiziert, dass es sich kurze Zeit nach Ausgabe der ersten Eingabeereignisse beendet.

Die Ausführung in Qemu auf einem Linux-Host ergab folgende Ausgabe:

```
vg      | ==== Nulgrind, the minimal Valgrind tool
          ...
vg      | Initialized DDELinux 2.6
vg      | Initializing DDE page cache
vg      | ddekit_pci_init
vg      | pci bus constructor
vg      | PCI: no root bridge found, no PCI
vg      | init_wq_head
vg      | deadbeaf
vg      | <6>serio: i8042 KBD port at 0x60,0x64 irq 1
vg      | <6>serio: i8042 AUX port at 0x60,0x64 irq 12
vg      | Hello. argc = 2
vg      | Testing L4INPUT callback mode...
vg      | DDE-Input is ready.
vg      | init => 0
vg      | <6>input: AT Translated Set 2 keyboard as
          /devices/platform/i8042/serio0/input/input0
vg      | <6>input: ImExPS/2 Generic Explorer Mouse as
          /devices/platform/i8042/serio1/input/input1
vg      | Event: type 1 (Key), code 28 (Enter), value 1
vg      | Event: type 1 (Key), code 28 (Enter), value 0
vg      | ====
vg      | path() = rr.log, size() = 2066547
vg      | begin 644 rr.log.gz
vg      | M'XL(````````"`^V="7A3Q?K&3U*6`(5&0*R"$O?B6O<JB$6V@J#!!8L;14"*
          ...
vg      | end
vg      | Valgrind exitting. Status = 0
```

Listing 2: Konsolenausgabe der Aufzeichnung

In der Ausgabe von dde_inputtst (fett gedruckt) ist zu sehen, dass ich während der Programmausführung die Enter-Taste gedrückt und wieder losgelassen habe. Anschließend hat sich das Programm beendet und die Log-Datei von Valgrind-RR wurde gzip-komprimiert und in UU-Codierung ausgegeben.

Die Konsolenausgabe habe ich in der Datei qemu.stdout gespeichert. Die Log-Datei ließ sich anschließend mit folgendem Script extrahieren:

```
#!/bin/bash

grep "........| " qemu.stdout |
  sed -e 's/\x1b\[[0-9]\{1,2\}\(;[0-9]\{1,2\}\)\{0,2\}m//g' \
      -e 's/........| //' \
      -e '/^begin/,/^end/!d' |
  uudeview -i -

gunzip *.gz
```

Listing 3: Script zur Extraktion von Log-Dateien aus der Konsolenausgabe

5.3 Deterministische Wiederausführung mit Analyse durch das memcheck-Tool

Für die deterministische Wiedergabe mit memcheck wurde folgende Konfigurationsdatei verwendet:

```
package.path = "rom/?.cfg";
require("VG");

local l = L4.default_loader;
local dummy_vbus_cap = l:new_channel();

VG.memcheck({ vbus = dummy_vbus_cap; }, "--track-origins=yes",
            "--record-replay=2", "--log-file-rr=rom/rr.log",
            "rom/dde_inputtst", "-cb");
```

Listing 4: Konfigurationsdatei für die Wiederausführung mit memcheck

Die Erzeugung eines virtuellen Gerätebuses war für die Wiederausführung nicht mehr nötig, da dabei keine I/O-Operationen mehr ausgeführt werden. Es musste allerdings eine Ersatzcapability generiert werden, damit diejenigen initialen Capabilities die auch bei der Wiederausführung benötigt werden (z.B. für den Region Manager) den selben Capability-Slot wie bei der Aufzeichnung belegen. Mit dem Aufrufargument `--record-replay=2` der `VG.memcheck()`-Funktion wird Valgrind-RR in den Wiedergabemodus versetzt.

Die Ausführung ergab nun die folgende Ausgabe:

```
vg      | ==== Memcheck, a memory error detector
          ...
vg      | Initialized DDELinux 2.6
vg      | Initializing DDE page cache
vg      | ddekit_pci_init
vg      | pci bus constructor
vg      | PCI: no root bridge found, no PCI
vg      | init_wq_head
vg      | deadbeaf
vg      | ==== Use of uninitialised value of size 4
vg      | ====    at 0x1026F40: sem_wait
          (/.../l4/pkg/uclibc/lib/libpthread/src/descr.h:242)
vg      | ====    by 0x103204E: ddekit_sem_down
          (/.../l4/pkg/dde/ddekit/src/semaphore.c:50)
vg      | ====    by 0x10316BB: ddekit_del_timer
          (/.../l4/pkg/dde/ddekit/src/timer.c:149)
vg      | ====    by 0x1000FFF: del_timer
          (/.../l4/pkg/dde/linux26/lib/src/arch/l4/timer.c:84)
vg      | ====    by 0x1037C13: schedule_timeout (in rom/dde_inputtst)
vg      | ====    by 0x10385A8: wait_for_common (in rom/dde_inputtst)
vg      | ====    by 0x10398CC: i8042_probe (in rom/dde_inputtst)
vg      | ====    by 0x1008B4A: driver_probe_device
          (/.../l4/pkg/dde/linux26/contrib/drivers/base/dd.c:124)
vg      | ====    by 0x10076E1: bus_for_each_drv
          (/.../l4/pkg/dde/linux26/contrib/drivers/base/bus.c:400)
vg      | ====    by 0x1008D2B: device_attach
          (/.../l4/pkg/dde/linux26/contrib/drivers/base/dd.c:256)
vg      | ====    by 0x1007580: bus_attach_device
          (/.../l4/pkg/dde/linux26/contrib/drivers/base/bus.c:507)
vg      | ====    by 0x100836D: device_add
          (/.../l4/pkg/dde/linux26/lib/src/drivers/base/core.c:935)
vg      | ====  Uninitialised value was created by a stack allocation
vg      | ====    at 0x1031B46: ddekit_thread_schedule
          (/.../include/x86/l4f/l4/sys/ipc-l42-gcc3-nopic.h:220)
vg      | ====
vg      | <6>serio: i8042 KBD port at 0x60,0x64 irq 1
vg      | <6>serio: i8042 AUX port at 0x60,0x64 irq 12
```

```
vg       | Hello. argc = 2
vg       | Testing L4INPUT callback mode...
vg       | DDE-Input is ready.
vg       | init => 0
vg       | <6>input: AT Translated Set 2 keyboard as
           /devices/platform/i8042/serio0/input/input0
vg       | <6>input: ImExPS/2 Generic Explorer Mouse as
           /devices/platform/i8042/serio1/input/input1
vg       | Event: type 1 (Key), code 28 (Enter), value 1
vg       | Event: type 1 (Key), code 28 (Enter), value 0
vg       | ====
vg       | REPLAY -- number of guest state mismatch: 0
vg       | ==== HEAP SUMMARY:
vg       | ====     in use at exit: 23,725 bytes in 103 blocks
vg       | ====   total heap usage: 111 allocs, 8 frees, 27,193 bytes
                    allocated
vg       | ====
vg       | ==== LEAK SUMMARY:
vg       | ====    definitely lost: 0 bytes in 0 blocks
vg       | ====    indirectly lost: 0 bytes in 0 blocks
vg       | ====      possibly lost: 2,126 bytes in 4 blocks
vg       | ====    still reachable: 21,599 bytes in 99 blocks
vg       | ====         suppressed: 0 bytes in 0 blocks
vg       | ==== Rerun with --leak-check=full to see details of leaked
                  memory
vg       | ====
vg       | ==== For counts of detected and suppressed errors, rerun with:
                  -v
vg       | ==== ERROR SUMMARY: 92 errors from 1 contexts (suppressed: 0
                  from 0)
vg       | Valgrind exitting. Status = 0
```

Listing 5: Konsolenausgabe bei der Wiederausführung mit memcheck

Die Meldungen von dde_inputtst stammen alle aus der Log-Datei, während der Ausführung habe ich keine Taste gedrückt. Die „`Use of uninitialised value of size 4`"-Meldung von memcheck stellte sich als Fehlalarm heraus. Abgesehen davon hat memcheck in diesem Fall keinen kritischen Fehler in dde_inputtst entdecken können. Um den Fehlerfall zu testen, habe ich in der Funktion serio_free_event() in der Linux-Treiberdatei serio.c einen Aufruf von kfree() auskommentiert und dde_inputtst anschließend erneut mit memcheck (ohne Replay) analysiert. Wie man an der Ausgabe sieht, wurde nun erwartungsgemäß ein Speicherleck erkannt:

```
vg       | ==0== HEAP SUMMARY:
vg       | ==0==     in use at exit: 23,805 bytes in 107 blocks
vg       | ==0==   total heap usage: 111 allocs, 4 frees, 27,193 bytes
                    allocated
vg       | ==0==
vg       | ==0== LEAK SUMMARY:
vg       | ==0==    definitely lost: 60 bytes in 3 blocks
vg       | ==0==    indirectly lost: 0 bytes in 0 blocks
vg       | ==0==      possibly lost: 2,126 bytes in 4 blocks
vg       | ==0==    still reachable: 21,619 bytes in 100 blocks
vg       | ==0==         suppressed: 0 bytes in 0 blocks
vg       | ==0== Rerun with --leak-check=full to see details of leaked
                  memory
vg       | ==0==
```

```
vg      | ==0== For counts of detected and suppressed errors, rerun with:
                -v
vg      | ==0== ERROR SUMMARY: 92 errors from 1 contexts (suppressed: 0
                from 0)
```

Listing 6: Konsolenausgabe mit erkanntem Speicherleck

5.4 Deterministische Wiederausführung mit Trace-Erzeugung durch das Chronicle-Tool

Als nächstes habe ich einen vollständigen Ausführungstrace der aufgezeichneten Ausführung von dde_inputtst mit Chronicle erstellt. Die Konfigurationsdatei dafür hatte folgenden Aufbau:

```
package.path = "rom/?.cfg";
require("VG");

local l = L4.default_loader;
local dummy_vbus_cap = l:new_channel();
local chronicle_channel = l:new_channel();

l:start({ caps = { chronicle_idx = chronicle_channel:svr() },
          log = { "chronicle-indexer", "yellow" } },
        "rom/chronicle-indexer");

VG.chronicle({ vbus = dummy_vbus_cap, chronicle_idx = chronicle_channel },
             "--record-replay=2", "--log-file-rr=rom/rr.log",
             "rom/dde_inputtst", "-cb");
```

Listing 7: Konfigurationsdatei für die Wiederausführung mit Chronicle

Neben Valgrind mit dem Chronicle-Tool musste hier noch der Chronicle-Indexer als separate Task gestartet werden. Die Konsolenausgabe der Ausführung sah nun folgendermaßen aus:

```
vg      | ==== Chronicle, Chronicle tracing engine
          ...
vg      | Initialized DDELinux 2.6
vg      | Initializing DDE page cache
vg      | ddekit_pci_init
vg      | pci bus constructor
vg      | PCI: no root bridge found, no PCI
vg      | init_wq_head
vg      | deadbeaf
vg      | <6>serio: i8042 KBD port at 0x60,0x64 irq 1
vg      | <6>serio: i8042 AUX port at 0x60,0x64 irq 12
vg      | Hello. argc = 2
vg      | Testing L4INPUT callback mode...
vg      | DDE-Input is ready.
vg      | init => 0
vg      | <6>input: AT Translated Set 2 keyboard as
          /devices/platform/i8042/serio0/input/input0
vg      | <6>input: ImExPS/2 Generic Explorer Mouse as
          /devices/platform/i8042/serio1/input/input1
vg      | Event: type 1 (Key), code 28 (Enter), value 1
vg      | Event: type 1 (Key), code 28 (Enter), value 0
vg      | ====
vg      | REPLAY -- number of guest state mismatch: 0
vg      | Valgrind exitting. Status = 0

    ---------------------------------------------------------------------
```

```
    CPU 0 [380045c6]: VG_(exit)
chronicl| path() = chronicle_db, size() = 1679699
chronicl| begin 644 chronicle_db.gz
chronicl| M'XL(````````"`]S]=4!;7]<NBD(IT*)M<6^+.Q37E%*@%(?B#L6=%$_0`L4+
          ...
chronicl| end
```

Listing 8: Konsolenausgabe bei der Wiederausführung mit Chronicle

Bei seiner Beendigung ruft Valgrind standardmäßig die Kerndebugger-Konsole auf, wodurch die Ausführung aller Tasks unterbrochen wird. Nach Drücken der Taste „g“ wird die Ausführung fortgesetzt und die vom Chronicle-Indexer erzeugte Log-Datei auf der Konsole ausgegeben. Diese kann anschließend ebenfalls mit dem in Listing 3 gezeigten Script aus der gespeicherten Konsolenausgabe extrahiert werden.

5.5 Erzeugung eines menschenlesbaren Funktionsaufruftraces mit chronisole

Der letzte Schritt meiner Beispieluntersuchung bestand darin, mit Hilfe von chronisole einen menschenlesbaren Funktionsaufruftrace zu erzeugen. Funktionsaufruftraces können z.B. nützlich sein, um das Verhalten von Programmen besser verstehen zu können, aber z.B. auch um das Verhalten eines Gerätetreibers mit einem vorhandenen Modell des Gerätes vergleichen zu können[19]. Es wäre auch denkbar, aus dem Aufruftrace ein Treibermodell zu generieren und andere Treiberimplementierungen für das gleiche Gerät (z.B. für ein anderes Betriebssystem) mit diesem Modell zu vergleichen. Das generierte Treibermodell könnte auch dazu verwendet werden, abnormales Verhalten des Treibers zu erkennen, z.B. bei einer Modifikation

Um den Informationsgehalt des von chronisole generierten Funktionsaufruftraces noch etwas zu erhöhen, habe ich das chronisole-Script so modifiziert, dass Aufruf und Rückkehr jeder Funktion als separate Ereignisse dargestellt werden. Außerdem werden nun zu jedem Einzelereignis jeweils der Zeitstempel und die Thread-ID mit ausgegeben. Durch eine nachträgliche Sortierung der Ausgabe nach dem Zeitstempel ist es nun möglich, das Auftreten der Ereignisse in ihrer zeitlichen Reihenfolge nachzuvollziehen und dabei auch Threadwechsel gut erkennen zu können.

Für meine Beispieluntersuchung habe ich chronisole folgendermaßen aufgerufen:

```
./chronisole.py trace dde_inputtst -c --max-depth=2 -f i8042_init \
                -f i8042_interrupt | sort -n > dde_inputtst.trace
```

Listing 9: Aufruf von chronisole

Mit diesen Argumenten erzeugt chronisole einen Funktionsaufruftrace für das Programm `dde_inputtst`, der alle Aufrufe der Funktionen `i8042_init()` und `i8042_interrupt()` berücksichtigt sowie alle unmittelbaren Unterfunktionsaufrufe dieser beiden Funktionen. Mit der Option `-c` gibt chronisole zu

jedem Aufruf- bzw. Rückkehrereignis auch die dazugehörigen Aufrufargumente bzw. Rückgabewerte mit aus. Die Chronicle-Datenbank habe ich vorher von `chronicle_db` in `dde_inputtst.db` umbenannt, damit sie von chronisole gefunden wird. Die Ausgabe wurde nach Zeitstempel sortiert in der Datei `dde_inputtst.trace` gespeichert und ist in Listing 10 ausschnittsweise abgebildet.

```
    212070   1 -> i8042_init ()
    212076   1   -> ddekit_printf (*fmt: '%lx\n')
    214226   1   <- ddekit_printf = None
    214229   1   -> i8042_platform_init ()
    214415   1   <- i8042_platform_init = 0x0
    214419   1   -> i8042_flush ()
    214924   1   <- i8042_flush = 0x1
    214929   1   -> platform_driver_register (drv: 0x10555c0)
    216960   1   <- platform_driver_register = 0x0
    216968   1   -> platform_device_alloc (*name: 'i8042', id: 0xffffffffL)
    217737   1   <- platform_device_alloc = 0xf030
    217746   1   -> platform_device_add (pdev: 0xf030)
    272586  11 -> i8042_interrupt (irq: 0x1, dev_id: 0xf030)
    272593  11   -> __raw_local_save_flags ()
    272598  11   <- __raw_local_save_flags = 0x2
    272600  11   -> raw_local_irq_disable ()
    272605  11   <- raw_local_irq_disable = None
    272610  11   -> ddekit_lock_lock (*mtx: 0x411e44)
    272675  11   <- ddekit_lock_lock = None
    272688  11   -> ddekit_lock_unlock (*mtx: 0x411e44)
    272745  11   <- ddekit_lock_unlock = None
    272748  11   -> raw_irqs_disabled_flags (flags: 0x2)
    272757  11   <- raw_irqs_disabled_flags = 0x1
    272760  11   -> raw_local_irq_restore (flags: 0x2)
    272766  11   <- raw_local_irq_restore = None
    272779  11 <- i8042_interrupt = 0x0
    294794   1   <- platform_device_add = 0x0
    294800   1 <- i8042_init = 0x0
    306832  11 -> i8042_interrupt (irq: 0x1, dev_id: 0xf030)
                  ...
    307700  11 <- i8042_interrupt = 0x1
    311461  11 -> i8042_interrupt (irq: 0x1, dev_id: 0xf030)
                  ...
    312326  11 <- i8042_interrupt = 0x1
    313769  11 -> i8042_interrupt (irq: 0x1, dev_id: 0xf030)
                  ...
    314634  11 <- i8042_interrupt = 0x1
    318683  11 -> i8042_interrupt (irq: 0x1, dev_id: 0xf030)
                  ...
```

Listing 10: Von chronisole erzeugter Funktionsaufruftrace

Aus diesen Informationen kann man nun z.B. mit dem Programm `dot`[20] einen visuellen Aufrufgraph erzeugen (siehe Abbildung 8) und somit das Verständnis des Ablaufs erleichtern. Die Graphbeschreibungsdatei für den dargestellten Graph habe ich von Hand erstellt, es wäre aber grundsätzlich auch eine automatische Generierung der Graphbeschreibung aus der chronisole-Ausgabe möglich.

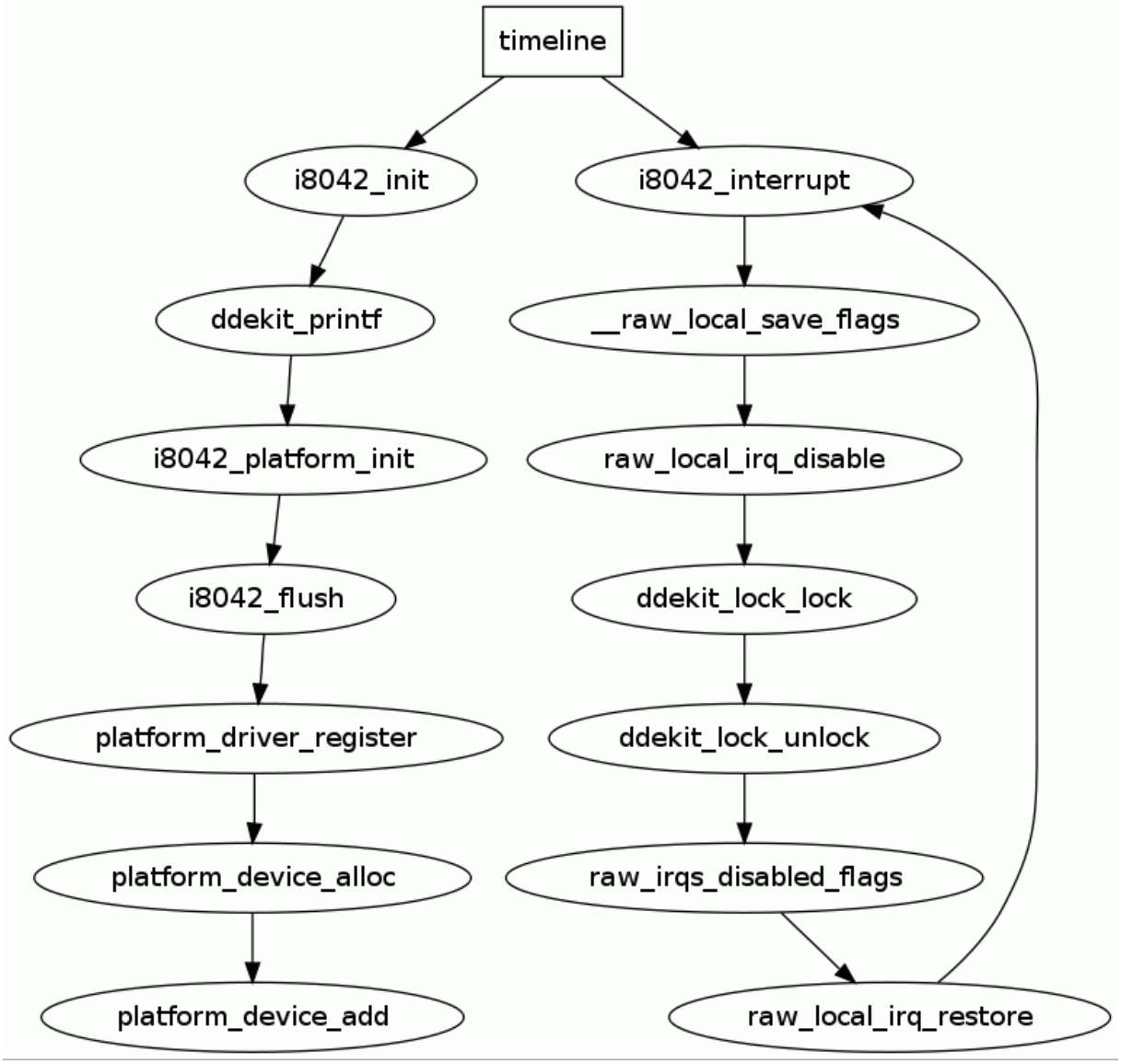

Abbildung 8: Mit dot erzeugter Aufrufgraph

6 Verwandte Arbeiten

In diesem Abschnitt stelle ich einige verwandte Arbeiten vor. Dabei habe ich eine Unterteilung in Arbeiten zu System-Level-Record/Replay und in Arbeiten zu VM-Level-Record/Replay vorgenommen. In meiner Arbeit habe ich mich mit System-Level-Record/Replay beschäftigt, d.h. mit der Aufzeichnung einzelner Tasks innerhalb eines Betriebssystems. Ein Vorteil der Aufzeichnung auf dieser Ebene besteht darin, dass sie auf echter Hardware und im Idealfall auf einfache Weise von jedem Benutzer ohne Änderungen am Betriebssystem durchgeführt werden kann. VM-Level-Record/Replay steht dagegen für die Aufzeichnung der Ausführung einer kompletten virtuellen Maschine, d.h. sowohl des Betriebssystemkerns als auch aller Tasks die in der virtuellen Maschine ausgeführt werden. Eine Aufzeichnung auf dieser Ebene ist z.B. vorteilhaft, wenn man den Betriebssystemkern oder das Zusammenspiel mehrerer Tasks untersuchen möchte, sie erfordert allerdings die Ausführung des Systems in einer virtuellen Maschine, wodurch Einschränkungen bei der Verwendung realer Hardware-Geräte entstehen können. Weitere Vor- und Nachteile der beiden Aufzeichnungsebenen ergeben sich aus der Beschreibung der einzelnen Arbeiten in den folgenden Unterkapiteln.

6.1 System-Level-Record/Replay

6.1.1 Jockey

Jockey[21] ist ein Record/Replay-Tool für Linux. Es hat den Anspruch, auf besonders einfache und sichere Weise das interaktive Debugging von Anwendungen zu unterstützen. Aus diesem Grund ist Jockey als Shared Library implementiert, die mit Hilfe des LD_PRELOAD-Mechanismus dynamisch beim Start der zu untersuchenden Anwendung in deren Adressraum geladen wird. Die Initialisierungsfunktion von Jockey durchsucht anschließend den Programmcode der Anwendung im Speicher nach Linux-Systemaufrufen und CPU-Instruktionen mit nichtdeterministischen Effekten und ersetzt diese durch Aufrufe von Wrapper-Funktionen im Jockey-Code. Diese Wrapper-Funktionen rufen im Record-Modus die ersetzten Systemaufrufe und CPU-Instruktionen auf und speichern anschließend deren nichtdeterministische Ergebnisse in einer Log-Datei. Im Replay-Modus werden die aufgezeichneten Ergebnisse aus der Log-Datei wiederhergestellt. Während der Ausführung des Programmes werden außerdem empfangene Signale sowie vom Programm vorgenommene Änderungen an Dateien die mit mmap() in den Adressraum eingeblendet wurden aufgezeichnet und wiederhergestellt. Um den Speicherbedarf für die Log-Datei zu verringern und bei lang laufenden Programmen eine schnelle Wiederherstellung eines bestimmten Programmzustandes zu ermöglichen, werden in regelmäßigen Abständen Checkpoints erstellt.

Das einmalige Ersetzen von CPU-Instruktionen ist weitaus weniger zeitaufwändig als die mehrfache Übersetzung alles ausgeführten Programmcodes wie bei Valgrind, es ermöglicht allerdings keine Unterstützung von Kernel-Level-Multithreading oder Shared Memory.

6.1.2 Flashback

Flashback[22] ist wie Jockey ein Record/Replay-Tool für die Unterstützung des interaktiven Debuggings von Linux-Anwendungen. Im Gegensatz zu Jockey ist Flashback jedoch nicht vollständig im Usermode implementiert, sondern nimmt auch Änderungen am Linux-Kernel vor. Zum Beispiel fügt es dem Linux-Kern Systemaufrufe zum Erzeugen, Wiederherstellen und Löschen von Checkpoints hinzu. Der Funktionsumfang ist ähnlich dem von Jockey, d.h. es werden neben der Erzeugung von Checkpoints die Effekte von Systemaufrufen, Signale und Änderungen an mit mmap() geöffneten Dateien gespeichert und wiederhergestellt. Durch die Änderungen am Linux-Kern ist der Einsatz von Flashback nicht so einfach und sicher wie der von Jockey, andererseits bestehen dadurch mehr Ausbaumöglichkeiten des Funktionsumfanges wie z.B. eine Unterstützung für Kernel-Level-Multithreading.

6.1.3 Rx

Rx[23] ist eine Lösung zur Erhöhung der Verfügbarkeit von Anwendungsprogrammen nach dem Auftreten von Software-Fehlern. Die Grundannahme von Rx ist, dass viele Software-Fehler durch externe Ereignisse ausgelöst werden. Die Verfügbarkeit des „behandelten“ Programmes soll nun dadurch erhöht werden, dass während der Ausführung des Programmes in regelmäßigen Abständen Checkpoints erstellt werden und nach dem Erkennen einer Fehlerwirkung der Programmzustand auf den gespeicherten Zustand zurückgesetzt wird und das Programm anschließend unter veränderten Umgebungsbedingungen weiter ausgeführt wird. Mit Hilfe sogenannter „Sensoren“ kann eine Vielzahl von verschiedenen Fehlerwirkungen erkannt werden wie z.B. Speicherzugriffsverletzungen, Divisionen durch Null oder Pufferüberläufe. Zu den Umgebungsänderungen nach dem Auftreten einer Fehlerwirkung gehören unter Anderem:

- eine Verzögerung der Wiederverwendung von mit free() freigegebenen Speicherbereichen
- die Einbettung von allozierten Puffern in Fülldaten
- eine weitestgehend räumliche Isolierung von allozierten Puffern im Adressraum
- eine Initialisierung neu allozierter Puffer mit dem Wert 0
- eine Umsortierung der Empfangsreihenfolge von Netzwerkpaketen
- eine Änderung der Größe von empfangenen Netzwerkpaketen
- eine Änderung der Scheduling-Reihenfolge
- eine Zustellung von Signalen zu anderen Zeitpunkten
- notfalls das Löschen von Netzwerkanfragen

Es werden so lange verschiedene Umgebungsänderungen ausprobiert bis das Programm eine bestimmte Zeit lang problemlos weiterläuft. Anschließend werden die Änderungen bis zur Erkennung einer weiteren Fehlerwirkung wieder eingestellt, da sie den Programmablauf verlangsamen und teilweise den Speicherbedarf erhöhen. Falls keine der Umgebungsänderungen den gewünschten Erfolg zeigt, werden die Änderungen auf einen früheren Checkpoint angewandt. Falls nach einer bestimmten Zeit immer noch kein Erfolg zu verzeichnen ist, wird das Programm komplett neu gestartet.

In jedem Fall wird von Rx zusätzlich eine Auswertung für den Programmierer erstellt in welcher dieser über das Auftreten des Fehlers und über die erfolgreichen und erfolglosen Änderungsmaßnahmen informiert wird und somit zumindest einen Anhaltspunkt für die Fehlerursache erhält.

6.1.4 First-Aid

First-Aid[24] ist eine Weiterentwicklung von Rx. Rx kann die Verfügbarkeit von Anwendungsprogrammen bereits merklich erhöhen, allerdings sind die Auswertungsinformationen von Rx für den Programmierer nicht immer hilfreich. Außerdem werden die Umgebungsänderungen bei Rx global vorgenommen (z.B. **jeder** neu allozierte Speicherpuffer wird mit Nullen aufgefüllt), wodurch der Zeit- und Speicheraufwand so hoch wird, dass die Änderungen nach einer bestimmten Zeit des fehlerfreien Betriebes wieder eingestellt werden müssen um den weiteren Ablauf des Programmes nicht übermäßig einzuschränken. Dadurch entsteht allerdings die Gefahr, dass die Fehlerwirkung kurze Zeit später erneut auftritt.

First-Aid dagegen kann die Umgebungsänderungen in Folge einer genaueren Fehlerdiagnose besser auf den konkreten Fehlerfall zuschneiden und mit Hilfe von Laufzeit-Patches auch bei zukünftiger Programmausführung das Auftreten dieses konkreten Fehlerfalles verhindern. Um eine genauere Fehlerdiagnose stellen zu können werden beim Auftreten einer Fehlerwirkung nicht nur fehlerverhindernde Änderungen wie bei Rx durchgeführt (z.B. die Einbettung von Speicherpuffern in zusätzlichen Speicher um Pufferüberläufe ungefährlich zu machen), sondern auch Änderungen die bestimmte Fehlerursachen nachweisen können (z.B. die Einbettung von Speicherpuffern in zusätzlichen Speicher mit einem bestimmten Inhalt und einem Vergleich dieses Inhaltes nach dem Auftreten der Fehlerwirkung um Pufferüberläufe nachzuweisen). Durch diese umfassendere Diagnose sind auch die Auswertungen von First-Aid für den Programmierer deutlich aussagekräftiger als die von Rx.

6.2 VM-Level-Record/Replay

6.2.1 ReVirt

ReVirt[25] zeichnet die Ausführung eines Linux-Gastsystems auf, um im Falle eines Angriffes auf dieses System die ausgenutzten Schwachstellen leichter finden und das System möglichst ohne große Datenverluste wieder in einen betriebsbereiten Zustand versetzen zu können.

Die virtuelle Maschine, auf der das Gastsystem ausgeführt wird, heißt UMLinux[26]. Dabei handelt es sich um einen modifizierten Linux-Kern mit einem speziellen Kernmodul. Das Gastsystem wird dabei als normaler Linux-Prozess ausgeführt, d.h. der im Benutzermodus verfügbare Teil des virtuellen Adressraums dieses Linux-Prozesses stellt den physischen Adressraum des Gastsystems dar.

Die im Gastsystem verfügbare Hardware wird auf dem Host-System emuliert. Massendatenspeicher wie Festplatten oder CD-ROM-Laufwerke werden beispielsweise in Form von Partitionen bzw. Dateien auf der Festplatte des Host-Systems bereitgestellt. Für den Zugriff auf die virtualisierten Hardware-Geräte verwendet der Kern des Gast-Systems Systemaufrufe des Host-Kerns wie z.B. `read()` und `write()`. Die von der virtuellen Maschine bereitgestellte Schnittstelle für den Zugriff auf Hardware entspricht also nicht der eines echten PCs und der Kern des Gast-Systems bzw. die eingesetzten Gerätetreiber wurden entsprechend an diese spezielle Schnittstelle angepasst. Interrupts und Exceptions werden bei UMLinux durch Signale simuliert.

Vor dem Start der virtuellen Maschine erstellt ReVirt einen Checkpoint durch Kopieren der virtuellen Festplatte. Während der Ausführung der virtuellen Maschine zeichnet ReVirt die Effekte der Systemaufrufe des Gastsystems auf sowie den Zeitpunkt der Zustellung und den Typ aller zugestellten Signale. Darüber hinaus werden auch die Resultate von nichtdeterministischen CPU-Instruktionen wie `rdtsc` aufgezeichnet.

6.2.2 Time-Travelling Virtual Machines (TTVM)

TTVM[27] ermöglicht das interaktive Debugging eines in einer virtuellen Maschine ausgeführten Linux-Systems. Es können sowohl der Linux-Kern als auch einzelne Anwendungen des Gastsystems untersucht werden. Dabei wird auch Rückwärts-Debugging unterstützt. Eine Besonderheit von TTVM ist zudem die Möglichkeit, mit unmodifizierten Gerätetreibern im Gastsystem reale Gerätes des Host-Systems verwenden zu können. Dadurch ist TTVM sehr gut für das interaktive Debugging von Linux-Gerätetreibern geeignet.

Als virtuelle Maschine kommt bei TTVM User-Mode-Linux (UML)[28] zum Einsatz. Dabei handelt es sich um einen modifizierten Linux-Kern, welcher es ermöglicht, ein komplettes Linux-System in zwei Benutzermodus-Prozessen auszuführen. Einer der beiden Prozesse führt dabei einen speziell angepassten Gast-Linux-Kern aus und der zweite Prozess führt genau eine Gast-Anwendung aus. Der Host-Linux-Kern wurde dabei so modifiziert, dass der virtuelle Adressraum des Gast-Anwendungsprozesses nachträglich ausgetauscht werden kann. Dadurch ist dieser eine Prozess in der Lage, mehrere Gast-Anwendungen auszuführen.

UML stellt dem Gastsystem einige simulierte Hardware-Geräte wie z.B. Festplatten oder Netzwerkadapter zur Verfügung. Festplatten werden auf dem Host-System auf Festplattenpartitionen oder Dateien abgebildet, Netzwerkadapter auf virtuelle TUN/TAP-Treiber. Interrupts werden dem Gast-Kern durch Signale übermittelt.

Für die Aufzeichnung der Ausführung des Gastsystems wird ReVirt (siehe Kapitel 6.2.1) eingesetzt, welches so modifiziert wurde, dass auch während der Ausführung regelmäßig Checkpoints erstellt werden können, um beim interaktiven Debugging nicht zu lange auf die Wiederherstellung eines bestimmten Zustandes warten zu müssen. Die Aufzeichnung der nichtdeterministischen Ereignisse wurde ebenfalls auf das Ziel einer möglichst schnellen Wiederherstellung von Zuständen hin optimiert, z.B. durch die Erstellung sowohl von Undo- als auch Redo-Logeinträgen bei der Aufzeichnung von Speicherzustandsänderungen.

Um das interaktive Debugging des Gastsystems zu ermöglichen, wurde der GNU-Debugger (GDB) so angepasst, dass er mit TTVM kommunizieren kann. Dabei wurden auch neue Kommandos für das Rückwärts-Debugging hinzugefügt wie z.B. `reverse step` und `reverse continue`, um das untersuchte Programm schrittweise oder bis zum Auslösen eines Breakpoints rückwärts ausführen zu können, oder auch das Komando `goto`, mit welchem der Zustand eines ausgewählten Ausführungszeitpunktes unmittelbar wiederhergestellt werden kann.

7 Zusammenfassung und Ausblick

In meiner Arbeit habe ich die beiden Valgrind-basierten Record/Replay-Lösungen Valgrind-RR und Chronicle für L4Re nutzbar gemacht. Dadurch ist es nun möglich, die Ausführung von L4Re-Anwendungen in Valgrind aufzuzeichnen und anschließend den aufgezeichneten Programmablauf mit mehreren verschiedenen Valgrind-Tools zu analysieren. Mit Hilfe des Chronicle-Tools können dabei vollständige Ausführungstraces generiert werden, die sich anschließend bequem unter Linux auswerten lassen. Zu den Auswertemöglichkeiten gehören unter Anderem die Erzeugung von menschenlesbaren Funktionsaufruftraces und das interaktive Debugging mit GDB, wobei auch eine simulierte Rückwärts-Ausführung möglich ist.

Ein wichtiger Bestandteil meiner Anpassungsarbeiten an L4Re war die Erweiterung von Valgrind-RR um die Aufzeichnung und Wiedergabe von Shared Memory-Lesezugriffen und I/O-Port-Leseoperationen. Dadurch ist es auch möglich, das üblicherweise von nichtdeterministischen Hardwareereignissen abhängige Verhalten von Gerätetreibern aufzeichnen und nachträglich analysieren zu können. Dies gilt insbesondere auch für Linux-Gerätetreiber im Device Driver Environment (DDE), die sich unter Linux nicht mit Valgrind-Tools analysieren lassen, da Valgrind nur Userlevel-Anwendungen unterstützt. Die Nutzbarkeit von Valgrind-RR und Chronicle für das Debugging von Linux-Gerätetreibern unter L4Re habe ich an Hand einer Beispieluntersuchung des Linux-Input-Treibers demonstriert.

Eine Einschränkung der Analysemöglichkeiten von Gerätetreibern besteht derzeit noch darin, dass Gerätetreiber, welche physischen Speicher mit kmalloc() allozieren, nicht mit dem memcheck-Tool analysiert werden können, da dieses nur virtuellen Speicher bereitstellen kann. Eine entsprechende Erweiterung von memcheck um die Unterstützung für physischen Speicher wäre wünschenswert.

Eine weitere nützliche Weiterentwicklung dieser Arbeit wäre die automatisierte Generierung von Treibermodellen oder visuellen Funktionsaufrufgraphen aus den vom chronisole-Programm erzeugten Funktionsaufruftraces.

Interessant wäre auch das Finden einer Möglichkeit, wie die von Valgrind-RR und Chronicle aufgezeichneten Informationen über Shared Memory-Lesezugriffe und Aufrufe von Locking-Funktionen für die Untersuchung des Zusammenspiels mehrerer L4Re-Anwendungen genutzt werden können, welche Daten über Shared Memory austauschen.

8 Literaturverzeichnis

[1] Adam Lackorzynski, Alexander Warg, Taming Subsystems - Capabilities as Universal Resource Access Control in L4, IIES ’09 Nuremberg, Germany, 2009
[2] L4Re-Website: http://os.inf.tu-dresden.de/L4Re/
[3] Nicholas Nethercote, Julian Seward, Valgrind: A Framework for Heavyweight Dynamic BinaryInstrumentation, PLDI’07, San Diego, California, USA, 2007
[4] Valgrind-RR Mailing List Posting: http://article.gmane.org/gmane.comp.debugging.valgrind.devel/5133
[5] Robert O'Callahan, Efficient Collection And StorageOf Indexed Program Traces, 2007
[6] Chronicle-Website: http://code.google.com/p/chronicle-recorder/
[7] GDB-Website: http://www.gnu.org/software/gdb/
[8] Fiasco.OC-Website: http://os.inf.tu-dresden.de/fiasco/
[9] Mohit Aron, Yoonho Park, Trent Jaeger, Jochen Liedtke,Kevin Elphinstone, Luke Deller, The SawMill Framework forVM Diversity. 6th Australasian Computer Systems Architecture Conference (AustCSAC'01), 2001
[10] Christian Helmuth, Generische Portierung von Linux-Gerätetreibern auf die DROPS-Architektur, 2001
[11] Julian Seward, Nicholas Nethercote, Using Valgrind to detect undefined value errors with bit-precision. Proceedings of the USENIX'05 Annual Technical Conference, Anaheim, California, USA, 2005
[12] JSON-Website: http://www.json.org/
[13] Chronomancer-Website: http://code.google.com/p/chronomancer/
[14] Eclipse-Website: http://www.eclipse.org/
[15] chronisole-Website: http://www.visophyte.org/blog/chroniquery/
[16] Python-Website: http://www.python.org/
[17] chronicle-gdbserver-Website: http://www.chiark.greenend.org.uk/~pmaydell/chronicle-gdbserver/
[18] uuencode-Spezifikation: http://pubs.opengroup.org/onlinepubs/9699919799/utilities/uuencode.html
[19] Dan Williams, Patrick Reynolds, Kevin Walsh, Emin G¨un Sirer, Fred B. Schneider, Device Driver Safety Through a Reference Validation Mechanism. 8th USENIX Symposium on Operating Systems Design and Implementation, 2008
[20] Graphviz-Website: http://www.graphviz.org/
[21] Yasushi Saito, Jockey: A UserspaceLibrary for RecordreplayDebugging. AADEBUG’05, Monterey, California, USA, 2005
[22] Sudarshan M. Srinivasan, Srikanth Kandula, Christopher R. Andrews, Yuanyuan Zhou, Flashback: A Lightweight Extension for Rollback and Deterministic Replay for Software Debugging. Proceedings of the General Track: 2004 USENIX Annual Technical Conference, Boston, MA, USA, 2004
[23] Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, Yuanyuan Zhou, Rx: Treating Bugs As Allergies - A Safe Method to SurviveSoftware Failures. SOSP’05, Brighton, United Kingdom., 2005
[24] Qi Gao, Wenbin Zhang, Yan Tang, Feng Qin, First-Aid: Surviving and PreventingMemory Management Bugsduring Production Runs. EuroSys’09, Nuremberg, Germany, 2009
[25] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, Peter M. Chen, ReVirt: Enabling Intrusion Analysis throughVirtual-Machine Logging and Replay. Proceedings of the 2002 Symposium on Operating Systems Design and Implementation (OSDI), 2002

[26] Kerstin Buchacker, Volkmar Sieh., Framework for testing the fault-toleranceof systems including OS and network aspects. Proceedings of the 2001 IEEESymposium on High Assurance SystemEngineering (HASE), 2001
[27] Samuel T. King, George W. Dunlap, and Peter M. Chen, Debugging operating systems with time-traveling virtual machines. 2005 USENIX Annual Technical Conference, 2005
[28] J. Dike, A user-mode port of the Linux kernel. Proceedingsof the 2000 Linux Showcase and Conference, 2000