



Bundesamt
für Sicherheit in der
Informationstechnik

Eine Studie im Auftrag des
Bundesamtes für Sicherheit in der Informationstechnik (BSI)

Dokumentation und Analyse des Linux- Pseudozufallszahlengenerators

Version 3.8 / 2013-12-02



Zusammenfassung

Die Beurteilung der Eignung und Qualität kryptographischer Systeme ist in Deutschland Aufgabe des Bundesamtes für Sicherheit in der Informationstechnik (BSI), das daher auch diese Studie des Zufallszahlengenerators von Linux in Auftrag gegeben hat. Linux wird in vielen Servern, Desktopsystemen und mobilen, eingebetten IT-Geräten verwendet, die in sensiblen Bereichen von Wirtschaft und Verwaltung einschließlich der kritischen Infrastrukturen zum Einsatz kommen. Gute Zufallszahlen des LRNG sind somit eine Voraussetzung für die Sicherheit der Daten in Behörden und Unternehmen wie auch in den Endgeräten, die in den Haushalten der Bürger zum Einsatz kommen.

Der Betriebssystem-Kernel Linux stellt über die Gerätedateien `/dev/random` und `/dev/urandom` User-Space-Programmen eine Schnittstelle zu seinem Zufallszahlengenerator (LRNG) bereit. Dessen Funktionen, Eigenschaften und Nutzung werden im Rahmen dieser Analyse untersucht. Dazu gehören u. a. die Entropiegewinnung (d. h. Entropiequellen und deren Gerätetreiber), die Durchmischungs- und Ausgabefunktion des Linux-eigenen Entropiepools sowie die Übergabe an den User-Space (z. B. zu Anwendungsprogrammen).

Von besonderem Interesse ist bei dieser Untersuchung neben der Untersuchung der algorithmischen Anteile des LRNG die Abschätzung der Entropie der in den LRNG einfließenden Rohdaten: Es soll die Frage geklärt werden, ob der LRNG in der Lage ist, 100 Bit Entropie zeitnah, d.h. auch nach einem Systemstart, bereitzustellen.

Autoren

Stephan Müller, atsec information security GmbH

Gerald Krummeck, atsec information security GmbH

Mario Romsy, atsec information security GmbH

Copyright

Das Werk einschließlich aller seiner Teile ist urheberrechtlich geschützt. Jede Verwertung außerhalb der engen Grenzen des Urhebergesetzes ist ohne Zustimmung des BSI unzulässig und strafbar. Dies gilt insbesondere für Vervielfältigung, Übersetzung, Mikroverfilmungen und die Einspeicherung und Verarbeitung in elektronischen Systemen.

Dokumentation und Analyse des Linux-Pseudozufallszahlengenerators

Linux Kernel Version 3.8

BSI-Referenz

BSI-Titel: Analyse des Linux-RNG

BSI-Projektnummer: 966

Inhaltsverzeichnis

1	Einleitung.....	11
1.1	Motivation und Zielsetzung.....	11
1.2	Methodik.....	11
1.3	Struktur des Dokuments.....	12
2	Das Design des Linux-RNG.....	13
2.1	Überblick.....	13
2.1.1	Bemerkungen.....	13
2.2	Entropie-Pools.....	14
2.3	Verwaltung der Pools.....	15
2.3.1	Verwaltung der Zeitvarianzen pro Rauschquelle.....	17
2.4	Implementierung der Gerätedateien.....	18
2.4.1	random_poll.....	19
2.4.2	Lesen und Schreiben.....	19
2.4.3	IOCTLs.....	19
2.4.4	fasync.....	20
2.5	Entropie.....	20
2.5.1	Sammlung von Entropie.....	20
2.5.2	Hinzufügen von Daten zu Entropie-Pools.....	31
2.5.3	Abschätzung der Entropie.....	35
2.6	Extraktion von Zufallszahlen.....	39
2.6.1	Extraktion von Daten via Gerätedateien.....	42
2.6.2	Extraktion von Daten innerhalb des Kerns.....	42
2.7	Hardware-Zufallszahlengeneratoren.....	43
2.7.1	Intel RDRAND-Instruktion.....	43
3	Bekannte Analysen des Linux-RNG.....	45
3.1	Angriffe von Gutterman et al. und deren aktuelle Relevanz.....	45
3.1.1	Denial-of-Service-Angriffe.....	45
3.1.2	Verwendung in Diskless-Systems.....	45
3.1.3	(Enhanced-)Backward-Secrecy.....	45
3.2	Analyse von Lacharme et al.....	46
3.2.1	LRNG ohne Eingabe in die Entropie-Pools.....	46
3.2.2	(Enhanced-)Forward-Secrecy.....	46
3.2.3	Eingabebasierte Angriffe.....	46
3.2.4	Bewertung der Entropieabschätzung.....	46
3.3	Schlussfolgerungen aus [GPR06] und [LRSV12].....	46
4	Abdeckung der Anforderungen an NTG.1.....	48
4.1	NTG.1.1.....	48
4.2	NTG.1.2.....	48
4.3	NTG.1.3.....	49
4.4	NTG.1.4.....	51
4.4.1	/dev/random.....	51
4.4.2	/dev/urandom.....	53
4.5	NTG.1.5.....	53
4.6	NTG.1.6.....	53
4.7	Struktur des LRNG.....	55
4.7.1	/dev/random.....	56
4.7.2	/dev/urandom.....	56
4.8	DRG.3 Anforderungen.....	56
4.8.1	DRG.3.1.....	57
4.8.2	DRG.3.2.....	58
4.8.3	DRG.3.3.....	58
4.9	Vorhandensein einer Entropieschätzung.....	59
4.9.1	/dev/random.....	59
4.9.2	/dev/urandom.....	60
5	Testreihen I: Untersuchung der Entropie-Pools.....	61
5.1	Motivation und Erklärung der verschiedenen Tests.....	61
5.2	Test 1 - Gesetzte Bits innerhalb des Entropie-Pools.....	63
5.2.1	Test 1 im laufenden Betrieb - Testansatz.....	63

5.2.2	Test 1 - Gesetzte Bits im Entropie-Pool <code>input_pool</code>	64
5.2.3	Test 1 - Gesetzte Bits im Entropie-Pool <code>nonblocking_pool</code>	65
5.2.4	Test 1 - Gesetzte Bits im Entropie-Pool <code>blocking_pool</code>	67
5.2.5	Test 1 zum Systemstart - Vorüberlegungen und Testansatz.....	68
5.2.6	Test 1 - Gesetzte Bits im Entropie-Pool <code>input_pool</code> zum Systemstart.....	70
5.2.7	Test 1 - Gesetzte Bits im Entropie-Pool <code>nonblocking_pool</code> zum Systemstart.....	76
5.2.8	Test 1 - Gesetzte Bits im Entropie-Pool <code>blocking_pool</code> zum Systemstart.....	82
5.3	Test 2 - Gesetzte Bits für jede Bitposition.....	84
5.3.1	Test 2 im laufenden Betrieb - Testansatz.....	84
5.3.2	Test 2 - Gesetzte Bits für jede Bitposition im Entropie-Pool <code>input_pool</code>	85
5.3.3	Test 2 - Gesetzte Bits für jede Bitposition im Entropie-Pool <code>nonblocking_pool</code>	86
5.3.4	Test 2 - Gesetzte Bits für jede Bitposition im Entropie-Pool <code>blocking_pool</code>	88
5.3.5	Test 2 zum Systemstart - Vorüberlegungen und Testansatz.....	89
5.3.6	Test 2 - Gesetzte Bits für jede Bitposition im Entropie-Pool <code>input_pool</code> zum Systemstart.....	89
5.3.7	Test 2 - Gesetzte Bits für jede Bitposition im Entropie-Pool <code>nonblocking_pool</code> zum Systemstart.....	94
5.3.8	Test 2 - Gesetzte Bits für jede Bitposition im Entropie-Pool <code>blocking_pool</code> zum Systemstart.....	99
5.4	Test 3 - Anzahl an Änderungen jeder einzelnen Bitposition.....	102
5.4.1	Test 3 im laufenden Betrieb - erwartete Wahrscheinlichkeiten.....	102
5.4.2	Test 3 im laufenden Betrieb - Testansatz.....	104
5.4.3	Test 3 - Veränderungen pro Bitposition im Entropie-Pool <code>input_pool</code>	104
5.4.4	Test 3 - Veränderungen pro Bitposition im Entropie-Pool <code>nonblocking_pool</code>	106
5.4.5	Test 3 - Veränderungen pro Bitposition im Entropie-Pool <code>blocking_pool</code>	108
5.4.6	Test 3 zum Systemstart - Vorüberlegungen und Testansatz.....	109
5.4.7	Test 3 - Veränderungen pro Bitposition im Entropie-Pool <code>input_pool</code> zum Systemstart.....	109
5.4.8	Test 3 - Veränderungen pro Bitposition im Entropie-Pool <code>nonblocking_pool</code> zum Systemstart.....	115
5.4.9	Test 3 - Veränderungen pro Bitposition im Entropie-Pool <code>blocking_pool</code> zum Systemstart.....	120
6	Testreihen II - Untersuchung der Entropie.....	124
6.1	Entropieschätzung für <code>input_pool</code>	124
6.1.1	Zeitverhalten von Trickle-Threshold.....	124
6.1.2	Zeitlicher Entropieverlauf.....	125
6.2	Analyse des Entropiemaßes für Ereignisse.....	128
6.2.1	Entropieschätzung des LRNG pro Hardware-Ereignis.....	128
6.2.2	Übersicht: Analyse des Entropiemaßes.....	131
6.2.3	Häufigkeitsverteilung und Entropie der Ereigniswerte.....	132
6.2.4	Häufigkeitsverteilung und Entropie der Prozessorzyklen.....	135
6.2.5	Häufigkeitsverteilung und Entropie der Jiffies.....	147
6.2.6	Untersuchung auf stochastische Unabhängigkeit.....	158
6.2.7	Vergleich theoretischer Entropie mit LRNG Abschätzungen.....	164
7	Weitere Tests.....	168
7.1	Verwendung der Zufallsdaten.....	168
7.1.1	Testdurchführung.....	168
7.1.2	Testresultate im laufenden System.....	168
7.1.3	Interpretation der Ergebnisse.....	169
7.2	Größe des Seeds.....	169
7.3	Test Procedure A.....	171
7.3.1	Testdurchführung.....	171
7.3.2	Testresultate.....	172
7.4	BSI Test Suite A.....	172
7.4.1	Testdurchführung.....	172
7.4.2	Testresultate.....	172
7.5	LRNG Ausgabe und „dieharder“-Test.....	172
7.5.1	Testdurchführung.....	172
7.5.2	Testresultate.....	172
7.6	Test des Referenzsystems.....	175

7.6.1 Test entsprechend Abschnitt 7.3.....	176
7.6.2 Test entsprechend Abschnitt 7.4.....	176
7.6.3 Test entsprechend Abschnitt 7.5.....	176
8 Zusammenfassung und Ausblick.....	179
8.1 Zusammenfassung.....	179
8.2 Ausblick und offene Fragen.....	180
Appendix A Technische Aspekte und Implementierung.....	181
A.1 SystemTap.....	181
A.1.1 Messfehler mit SystemTap.....	182
A.1.2 Voraussetzungen für den Einsatz.....	182
A.1.3 Einfluss der Messungen auf die Resultate.....	183
A.2 Testdurchführung.....	184
A.2.1 Vorbetrachtungen.....	184
A.2.2 Aufruf von SystemTap-Skripten.....	184
A.2.3 Aufruf von R-Project Programmen.....	184
A.2.4 Verwendete Hard- und Software.....	184
A.2.5 Nachbildung der Einsatzumgebung.....	185
Appendix B Neuerungen im LRNG.....	186
B.1 Linux Kern 3.5.....	186
B.1.1 Änderung von in Kapitel 1 genannten Dateien.....	186
B.1.1.1 drivers/char/random.c.....	186
B.1.1.2 include/linux/random.h.....	187
B.1.1.3 arch/x86/include/asm/archrandom.h.....	187
B.1.2 Änderungen in Aufrufen der Entropiesammelfunktionen.....	187
B.1.2.1 add_input_randomness.....	187
B.1.2.2 add_interrupt_randomness.....	188
B.1.2.3 add_disk_randomness.....	188
B.1.3 Definition und Verwendung von neuen Schnittstellen.....	188
B.2 Linux Kern 3.6.....	188
B.2.1 Änderung von in Kapitel 1 genannten Dateien.....	188
B.2.1.1 drivers/char/random.c.....	188
B.2.1.2 include/linux/random.h.....	189
B.2.1.3 arch/x86/include/asm/archrandom.h.....	190
B.2.2 Änderungen in Aufrufen der Entropiesammelfunktionen.....	190
B.2.2.1 add_input_randomness.....	190
B.2.2.2 add_interrupt_randomness.....	190
B.2.2.3 add_disk_randomness.....	190
B.2.3 Definition und Verwendung von neuen Schnittstellen.....	190
B.3 Linux-Kern 3.7.....	190
B.3.1 Änderung von in Kapitel 1 genannten Dateien.....	190
B.3.1.1 drivers/char/random.c.....	190
B.3.1.2 include/linux/random.h.....	190
B.3.1.3 include/uapi/linux/random.h.....	190
B.3.1.4 arch/x86/include/asm/archrandom.h.....	191
B.3.2 Änderungen in Aufrufen der Entropiesammelfunktionen.....	191
B.3.2.1 add_input_randomness.....	191
B.3.2.2 add_interrupt_randomness.....	191
B.3.2.3 add_disk_randomness.....	191
B.3.3 Definition und Verwendung von neuen Schnittstellen.....	191
B.4 Linux-Kern 3.8.....	191
B.4.1 Änderung von in Kapitel 1 genannten Dateien.....	191
B.4.1.1 drivers/char/random.c.....	191
B.4.1.2 include/linux/random.h.....	191
B.4.1.3 include/uapi/linux/random.h.....	192
B.4.1.4 arch/x86/include/asm/archrandom.h.....	192
B.4.2 Änderungen in Aufrufen der Entropiesammelfunktionen.....	192
B.4.2.1 add_input_randomness.....	192
B.4.2.2 add_interrupt_randomness.....	192
B.4.2.3 add_disk_randomness.....	192
B.4.3 Definition und Verwendung von neuen Schnittstellen.....	192

B.5 Linux-Kern 3.9.....	192
B.5.1 Änderung von in Kapitel 1 genannten Dateien.....	192
B.5.1.1 drivers/char/random.c.....	192
B.5.1.2 include/linux/random.h.....	192
B.5.1.3 include/uapi/linux/random.h.....	193
B.5.1.4 arch/x86/include/asm/archrandom.h.....	193
B.5.2 Änderungen in Aufrufen der Entropiesammelfunktionen.....	193
B.5.2.1 add_input_randomness.....	193
B.5.2.2 add_interrupt_randomness.....	193
B.5.2.3 add_disk_randomness.....	193
B.5.2.4 add_device_randomness.....	193
B.5.3 Definition und Verwendung von neuen Schnittstellen.....	193
B.6 Linux-Kern 3.10.....	193
B.6.1 Änderung von in Kapitel 1 genannten Dateien.....	193
B.6.1.1 drivers/char/random.c.....	193
B.6.1.2 include/linux/random.h.....	194
B.6.1.3 include/uapi/linux/random.h.....	194
B.6.1.4 arch/x86/include/asm/archrandom.h.....	194
B.6.2 Änderungen in Aufrufen der Entropiesammelfunktionen.....	194
B.6.2.1 add_input_randomness.....	194
B.6.2.2 add_interrupt_randomness.....	194
B.6.2.3 add_disk_randomness.....	194
B.6.2.4 add_device_randomness.....	194
B.6.3 Definition und Verwendung von neuen Schnittstellen.....	194
B.7 Linux-Kern 3.11.....	194
B.7.1 Änderung von in Kapitel 1 genannten Dateien.....	194
B.7.1.1 drivers/char/random.c.....	194
B.7.1.2 include/linux/random.h.....	195
B.7.1.3 include/uapi/linux/random.h.....	195
B.7.1.4 arch/x86/include/asm/archrandom.h.....	195
B.7.2 Änderungen in Aufrufen der Entropiesammelfunktionen.....	195
B.7.2.1 add_input_randomness.....	195
B.7.2.2 add_interrupt_randomness.....	195
B.7.2.3 add_disk_randomness.....	195
B.7.2.4 add_device_randomness.....	195
B.7.3 Definition und Verwendung von neuen Schnittstellen.....	195
B.8 Linux-Kern 3.12.....	195
B.8.1 Änderung von in Kapitel 1 genannten Dateien.....	195
B.8.1.1 drivers/char/random.c.....	195
B.8.1.2 include/linux/random.h.....	196
B.8.1.3 include/uapi/linux/random.h.....	196
B.8.1.4 arch/x86/include/asm/archrandom.h.....	196
B.8.2 Änderungen in Aufrufen der Entropiesammelfunktionen.....	196
B.8.2.1 add_input_randomness.....	196
B.8.2.2 add_interrupt_randomness.....	196
B.8.2.3 add_disk_randomness.....	196
B.8.2.4 add_device_randomness.....	196
B.8.3 Definition und Verwendung von neuen Schnittstellen.....	196
Appendix C Mapping zum Dokument von T-Systems.....	197
Appendix D Abkürzungen und Glossar.....	198
Appendix E Literaturverzeichnis.....	199

1 Einleitung

1.1 Motivation und Zielsetzung

*Random numbers should not
be generated with a method
chosen at random.*

Donald E.Knuth
The Art of Computer Programming

Kryptographische Verfahren sind heute unerlässlich für die Gewährleistung von Vertraulichkeit, Integrität und Authentizität digitaler verarbeiteter Daten. Zufallszahlen sind unverzichtbare Kern-Bestandteile dieser Krypto-Systeme. Sie werden für die Erzeugung kryptographisch sicherer Parameter, insbesondere des Schlüsselmaterials, benötigt und müssen daher auf ihre kryptographische Eignung hin untersucht und beurteilt werden. Nur so kann die Sicherheit der entsprechenden Krypto-Systeme gewährleistet werden.

Die Beurteilung der Eignung und Qualität kryptographischer Systeme ist in Deutschland Aufgabe des Bundesamtes für Sicherheit in der Informationstechnik (BSI), das daher auch diese Studie des Zufallszahlengenerators von Linux in Auftrag gegeben hat. Linux wird in vielen Servern, Desktopsystemen und mobilen, eingebetten IT-Geräten verwendet, die in sensiblen Bereichen von Wirtschaft und Verwaltung einschließlich der kritischen Infrastrukturen zum Einsatz kommen. Gute Zufallszahlen des LRNG sind somit eine Voraussetzung für die Sicherheit der Daten in Behörden und Unternehmen wie auch in den Endgeräten, die in den Haushalten der Bürger zum Einsatz kommen.

Der Betriebssystem-Kernel Linux stellt über die Gerätedateien `/dev/random` und `/dev/urandom` User-Space-Programmen eine Schnittstelle zu seinem Zufallszahlengenerator (LRNG) bereit. Dessen Funktionen, Eigenschaften und Nutzung werden im Rahmen dieser Analyse untersucht. Dazu gehören u. a. die Entropiegewinnung (d. h. Entropiequellen und deren Gerätetreiber), die Durchmischungs- und Ausgabefunktion des Linux-eigenen Entropiepools sowie die Übergabe an den User-Space (z. B. zu Anwendungsprogrammen).

Von besonderem Interesse ist bei dieser Untersuchung neben der Untersuchung der algorithmischen Anteile des LRNG die Abschätzung der Entropie der in den LRNG einfließenden Rohdaten: Es soll die Frage geklärt werden, ob der LRNG in der Lage ist, 100 Bit Entropie zeitnah, d.h. auch nach einem Systemstart, bereitzustellen.

Dieser Bericht wurde von der atsec information security GmbH im Auftrag des BSI unter der BSI-Projektnummer 966 angefertigt. Das BSI hält alle Rechte an diesem Dokument.

1.2 Methodik

Die vorliegende Dokumentation und Analyse des Analyse des Linux-Pseudozufallszahlengenerators (kurz LRNG oder auch Linux-RNG) erfolgt für den Linux-Kern in der Version 3.8 auf Prozessoren mit Intel-Architektur (x86).

Die Struktur der Analyse orientiert sich dabei an der Vorgängeranalyse des LRNG für den Linux-Kern in der Version 2.6.21.5 aus dem Jahr 2007. Eine kurze Gegenüberstellung der Gemeinsamkeiten und Unterschiede zu [LLT07] ist in Anhang C zu finden.

Wie auch in [LLT07] soll hier untersucht werden, ob der LRNG für die Verwendung in einem sicherheitsrelevanten Umfeld eingesetzt werden kann. Dies erfordert insbesondere, dass die für die Erzeugung von Zufallszahlen eingesetzten Entropiequellen in der Lage sind, die vom Auftraggeber geforderten 100 Bit Entropie zeitnah – auch nach einem Systemstart – zu generieren. Die Diskussion in Abschnitt 8.1 beantwortet obige Frage positiv.

Für eine etwas ausführlichere Zusammenfassung der Ergebnisse unserer Untersuchungen verweisen wir auf Kapitel 8.

Als Grundlage für die Untersuchungen in diesem Dokument dienen die folgenden Quellcode-Dateien des Linux-Kerns, in denen der LRNG implementiert ist:

- Kernfunktionalität: `drivers/char/random.c`
- Unterstützende Header-Datei: `include/linux/random.h`
- User Space API Header-Datei: `include/uapi/linux/random.h`
- Intel x86_64 RDRAND Unterstützung: `arch/x86/include/asm/archrandom.h`

1.3 Struktur des Dokuments

Diese Analyse enthält die folgenden Teile, welche größtenteils aufeinander aufbauen:

- Kapitel 2 beschreibt das Design des LRNG. Diese Beschreibung soll dem Leser helfen, die Messreihen zu interpretieren und die getroffenen Aussagen nachzuvollziehen.
- Kapitel 3 dient einem kurzen Überblick über bekannte Analysen und Angriffe.
- In Kapitel 4 wird der LRNG auf Konformität zu den Anforderungen von NTG.1 überprüft.
- In Kapitel 5 werden die Daten innerhalb der Entropie-Pools des LRNG experimentell auf eine gleichmäßige Verteilung untersucht. Zudem wird in weiteren Tests die Änderungsrate der Daten innerhalb der Entropie-Pools bestimmt und mit den theoretischen Werten verglichen. Alle Tests in diesem Kapitel wurden auf dem Kern 3.8 durchgeführt.
- Kapitel 6 betrachtet die Entropieschätzungen des LRNG. Dazu wird einerseits der zeitliche Verlauf der geschätzten Entropie gemessen. Andererseits werden die unterschiedlichen Hardware-Ereignisse und deren Entropie untersucht, wobei die Schätzungen auch mit theoretischen Werten verglichen werden. Alle Tests in diesem Kapitel sind auf dem Kern 3.8 durchgeführt.
- Kapitel 7 zeigt einen Test über die interne Verwendung von Zufallszahlen. Da in den vorherigen Kapiteln deutlich wird, dass ein Einbringen eines Seeds zum Systemstart äußerst wünschenswert ist, folgt noch eine Diskussion über ein sinnvolle Größe des Seeds. Alle Tests in diesem Kapitel sind auf dem Kern 3.8 durchgeführt.
- Kapitel 8 gibt einen kurzen Überblick über die wichtigsten Ergebnisse.
- Appendix A enthält eine Diskussion des verwendeten Analyseansatzes. Das vorliegende Dokument hat auch das Ziel, quantitative Aussagen zu treffen. Demzufolge müssen die Messinstrumente erläutert werden.
- Dieses Dokument soll regelmäßig bei dem Erscheinen von neuen Versionen des Linux-Kerns aktualisiert werden, Appendix B diskutiert diese Neuerungen. Falls relevante Änderungen in einem neuen Linux-Kern enthalten sind, werden die betroffenen Stellen in diesem Dokument entsprechend angepasst.
- Appendix C bietet eine kurze Gegenüberstellung dieses Dokuments mit der Analyse in [LLT07].
- Appendix D enthält ein kleines Glossar und die verwendeten Abkürzungen.

2 Das Design des Linux-RNG

2.1 Überblick

Die ursprüngliche Implementierung des Linux-RNG stammt aus dem Jahr 1994 und wurde von Theodore Ts'o erstellt.

Der Linux-RNG ist ein Pseudozufallszahlengenerator, der Hardware-Ereignisse als Entropie-Quellen verwendet. Kurz zusammengefasst ist die Funktionsweise wie folgt: Nach Auftreten eines Hardware-Ereignisses wird der entsprechende Ereigniswert auf seinen Entropiegehalt geschätzt und gegebenenfalls durch eine spezielle Funktion in den 4096 Bit großen primären Entropie-Pool, `input_pool` genannt, eingemischt. Dieser primäre Entropie-Pool speist auf Anforderung die zwei sekundären Entropie-Pools, `blocking_pool` und `nonblocking_pool`, die je 1024 Bit groß sind. Wie in [LPM10a] beschrieben wird, stellt der Linux-Kern für die Extraktion von Zufallszahlen durch Subjekte des User-Space die folgenden Gerätedateien bereit

- `/dev/random` und
- `/dev/urandom`,

die auf `blocking_pool` beziehungsweise `nonblocking_pool` zugreifen. Der Unterschied im Verhalten beider Gerätedateien besteht nach [LPM10a] im Blockieren (`/dev/random`) beziehungsweise dem fehlenden Blockieren (`/dev/urandom`) beim Auslesen von Zufallszahlen, falls nicht ausreichend Entropie vorhanden ist. Bei Anfragen an `/dev/random` und `/dev/urandom` werden dann Zufallszahlen durch eine komplexe Funktion, die insbesondere mehrfache Aufrufe von SHA-1 nutzt, aus den sekundären Entropie-Pools extrahiert. Als Entropie-Quellen dienen in der betrachteten Version des Linux-Kerns Blockgeräte (zum Beispiel Festplatten), Eingabegeräte (zum Beispiel Tastatur und Maus) und Interrupts.

2.1.1 Bemerkungen

- In einer Antwort [T06] auf die Arbeit von Gutterman et al. (siehe [GPR06]) erklärt Ts'o, dass aufgrund der zur Entwicklungszeit des LRNG geltenden Exportbeschränkungen der USA für Kryptographie auf den Einsatz von Verschlüsselungsalgorithmen zu Gunsten von kryptographischen Hashfunktionen, hier SHA-1, verzichtet wurde. Zudem wurde das System so gestaltet, dass auch ein Brechen der Kollisionsresistenz von SHA-1 nicht zu einer Kompromittierung des LRNG führt.
- Aktuelle Arbeiten zeigen, dass gerade im Bereich von Headless-Systemen, die beim ersten Systemstart Zufallszahlen für die Erzeugung von Schlüsseln benötigen, zu wenig Entropie vorhanden ist. Auch wenn von den Interrupts nicht viel Entropie zu erwarten ist, sollte die Verwendung der Interrupt-Ereignisse dieses Problem etwas bessern.
- In den nachfolgenden Untersuchungen wird deutlich werden, dass sowohl beim Systemstart als auch im laufenden Betrieb (zum Teil durch - unserer Meinung nach - verschwenderischen Umgang) ein Mangel an vorhandener Entropie auftreten kann. In dieser Hinsicht sollten auch neue Entropie-Quellen erschlossen werden. Bei Serversystemen ist beispielsweise eine Nutzung von CPU-Temperatur und besser der Lüfterdrehzahlen denkbar. Moderne mobile Geräte bieten zudem mit Kameras, Mikrofonen und Lagesensoren eine breite Palette möglicher Rauschquellen.

Die folgende Grafik (Abbildung 1) gibt einen Überblick über den gesamten Funktionsablauf und die Struktur des LRNG. Diese Graphik wird im folgenden detailliert beschrieben und diskutiert.

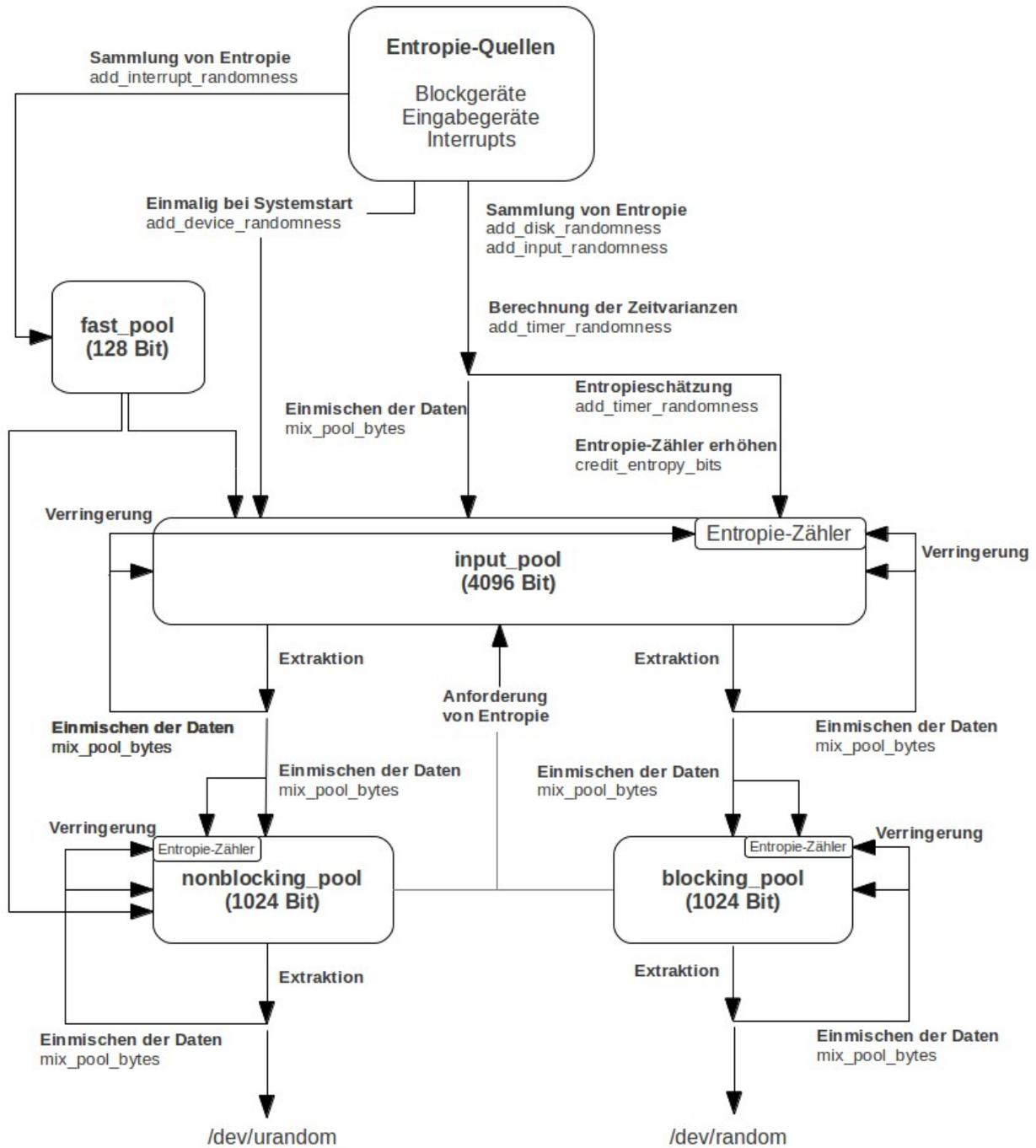


Abbildung 1: Aufbau des LRNG

2.2 Entropie-Pools

Der LRNG verwaltet drei getrennte Entropie-Pools, um die berechnete Entropie zu speichern und zu verarbeiten.

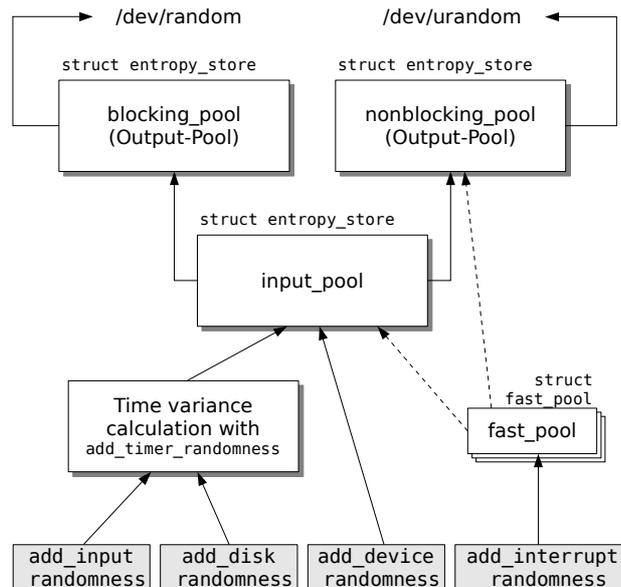


Abbildung 2: Beziehungen der Entropie-Pools und Rauschquellen

Abbildung 2 beschreibt die Beziehungen der verschiedenen Entropie-Pools und der Rauschquellen. Die Pfeile symbolisieren den Informationsfluss. Dabei sind die Rauschquellen grau schattiert.

Der LRNG verwaltet verschiedene Entropie-Pools, denen jeweils ein Zufallszahlengenerator zugeordnet ist. Diese separaten Zufallszahlengeneratoren stehen zueinander wie folgt in Beziehung:

- **input_pool** bezeichnet den primären Zufallszahlengenerator, der die Entropie direkt aus den Hardware-Ereignissen verarbeitet. Auf diesen Pool kann aus dem User-Space nicht direkt zugegriffen werden. Er ist 4096 Bit groß.
- Vom `input_pool` werden zwei sekundäre Zufallszahlengeneratoren gespeist, die als **blocking_pool** und **nonblocking_pool** bezeichnet werden. Vom User-Space aus werden der `blocking_pool` über die Gerätedatei `/dev/random` und der `nonblocking_pool` über `/dev/urandom` angesprochen. Beide Pools sind jeweils 1024 Bit groß.

Der Quellcode zum Verwalten der sekundären Zufallszahlengeneratoren (`blocking_pool` und `nonblocking_pool`) ist identisch, einschließlich der Datenstrukturen. Demzufolge sind alle Funktionen, die der LRNG bereitstellt, nur einmal implementiert und werden mit einer Referenz auf die Datenstrukturinstanz vom `blocking_pool` oder `nonblocking_pool` aufgerufen. Der einzige Unterschied im Quellcode zwischen beiden ist die Lesefunktion, welche Daten aus dem `blocking_pool` beziehungsweise aus dem `nonblocking_pool` für den User-Space ableitet.

2.3 Verwaltung der Pools

Die Implementierung des LRNG instanziiert eine separate Datenstruktur für jeden der drei in Abschnitt 2.2 diskutierten Pools. Für alle Pools wird die gleiche Datenstruktur verwendet, die die folgenden wichtigen Informationen enthält:

```

struct entropy_store {
    /* read-only data: */
    struct poolinfo *poolinfo;
    __u32 *pool;
    ...
    struct entropy_store *pull;
    ...
    unsigned add_ptr;
    unsigned input_rotate;
    int entropy_count;
    int entropy_total;
    unsigned int initialized:1;
    __u8 last_data[EXTRACT_SIZE];
};

```

Quellcode 1: drivers/char/random.c

Die Variablen in der Datenstruktur haben die folgende Bedeutung:

- Die Variable **poolinfo** enthält die Referenz auf das Polynom, welches für die Implementierung des linearen Schieberegisters verwendet wird. Derzeitig werden zwei verschiedene Polynome verwendet: eines für das Verarbeiten von Änderungen von `input_pool`. Das zweite Polynom wird für die Steuerung der Instanzen `blocking_pool` und `nonblocking_pool` verwendet. Weitere Informationen, wie die Polynome angewendet werden, werden in Abschnitt 2.5.2 diskutiert. Der folgende Quellcode erklärt die Definition der Polynome – dabei ist zu beachten, dass der erste Wert die Anzahl der 32 Bit Integer-Zahlen angibt und die folgenden Werte das Polynom definieren:

```

static struct poolinfo {
    int poolwords;
    int tap1, tap2, tap3, tap4, tap5;
} poolinfo_table[] = {
    /* x^128 + x^103 + x^76 + x^51 + x^25 + x + 1 -- 105 */
    { 128, 103, 76, 51, 25, 1 },
    /* x^32 + x^26 + x^20 + x^14 + x^7 + x + 1 -- 15 */
    { 32, 26, 20, 14, 7, 1 },

```

Quellcode 2: drivers/char/random.c

- Die Variable **pool** enthält den Zeiger auf die globale Variable, die den tatsächlichen Entropie-Pool umfasst. Der folgende Quellcode-Block zeigt die Definition der Pools für `input_pool`, `blocking_pool` und `nonblocking_pool`. Während der Initialisierung des LRNG werden die Zeiger auf diese globalen Variablen in die jeweilige Instanz der Variable `pool` geschrieben.

```

#define INPUT_POOL_WORDS 128
#define OUTPUT_POOL_WORDS 32
...
static __u32 input_pool_data[INPUT_POOL_WORDS];
static __u32 blocking_pool_data[OUTPUT_POOL_WORDS];
static __u32 nonblocking_pool_data[OUTPUT_POOL_WORDS];

```

Quellcode 3: drivers/char/random.c

- Der Wert von **pull** ist ein Zeiger auf den primären Zufallszahlengenerator. Entsprechend der Diskussion in Abschnitt 2.6 enthält diese Variable den Wert NULL für

`input_pool`. Hingegen ist der Wert in `blocking_pool` und `nonblocking_pool` eine Referenz auf `input_pool`.

- In der Variable **`add_ptr`** wird ein Index gespeichert, der auf die Integer-Zahl im Entropie-Pool zeigt, die bei dem vorangegangenen Hinzufügen von Werten in den Pool als letzte modifiziert wurde. Mittels dieses Zeigers verschiebt sich das „Fenster“ des Schieberegisters, wenn Werte in den Pool eingefügt werden. Weitere Informationen, wie Daten zu den Pools hinzugefügt werden, enthält Abschnitt 2.5.2.
- Die Integer-Zahl **`entropy_count`** enthält die geschätzte Entropie (in Bits) des Pools. Der Wert wird anhand der Heuristiken des LRNG vergrößert, wenn neue Werte zum Pool hinzugefügt werden, und verkleinert, wenn Zufallszahlen aus dem Pool abgerufen werden. Dieser Wert symbolisiert die Entropie, welche von dem jeweiligen Pool entsprechend der Heuristiken des LRNG angenommen wird.
- Die Variable **`input_rotate`** enthält den Rotationswert für die Eingangsdaten. Dieser Wert wird beim Hinzufügen von neuen Daten in den Pool benötigt, wie in Abschnitt 2.5.2 dargelegt.
- Die Variable **`entropy_total`** enthält einen Zähler, der die gesamte Entropie, die für einen Entropie-Pool erzeugt wurde, bis zum Überschreiten der Schranke von 128 (Bit) aufsummiert. Dieser Zähler wird dazu verwendet, die `initialized` Variable auf 1 (true) zu setzen, wenn der Zähler 128 (Bit) überschreitet.
- Bei der Implementierung der Funktion `add_interrupt_randomness` wird die Variable **`initialized`** Variable verwendet, um den Entropie-Pool auszuwählen, der mit Entropie von Interrupt-Ereignissen gespeist wird. Ist diese Variable für den `nonblocking_pool` auf 1 (true) gesetzt, wird der `input_pool` als Ziel verwendet. Ansonsten wird der `nonblocking_pool` verwendet.
- Die Variable **`last_state`** enthält die letzte Zufallszahl, welche aus dem entsprechenden Pool erzeugt wurde. Diese Variable wird für den FIPS 140-2 „Continuous Self Test“ verwendet und wird nur initialisiert, wenn der Kern im FIPS-Modus gestartet wurde.

Die besprochene Datenstruktur wird während der Übersetzung des Quellcodes für jeden der Pools initialisiert.

2.3.1 Verwaltung der Zeitvarianzen pro Rauschquelle

Für jede Hardware-Rauschquelle verwaltet der Kernel eine eigene Datenstruktur-Instanz, um die Zeitvarianzen von Ereignissen genau dieser Quelle zu zuordnen.

Abbildung 3 zeigt die drei Klassen von Rauschquellen. Für jede dieser Klassen instantiiert der Kern die Datenstruktur für die Zeitvarianzen wie folgt:

- Eingabegeräte (Geräte, die die „Konsole“ des Systems definieren: Tastatur, Maus, Tablet, etc.): Der Kern erzeugt eine Instanz der Datenstruktur für alle Eingabegeräte zusammen, d.h. die Zeitvarianzen der Ereignisse aller Eingabegeräte werden gemessen. Damit wird die Gesamtheit aller Eingabegeräte als eine Rauschquelle angesehen.
- Speichermedien: Pro Speichermedium wird eine Instanz der Datenstruktur vorgehalten. Dies bedeutet, dass die Zeitvarianzen der Ereignisse pro physischen Speichermedium gespeichert werden. Damit wird ein einzelnes physisches Speichermedium (z.B. eine Festplatte) als eine Rauschquelle definiert.
- Interrupts: Pro verwendetem IRQ wird eine Instanz der Datenstruktur vorgehalten. Demzufolge werden die Zeitvarianzen der Ereignisse pro IRQ gemessen. Dies bedeutet, dass ein IRQ eine Rauschquelle darstellt.

Die Datenstruktur für die Zeitvarianzen enthält die folgenden Informationen:

```

/* There is one of these per entropy source */
struct timer_rand_state {
    cycles_t last_time;
    long last_delta, last_delta2;
    unsigned dont_count_entropy:1;
};

```

Quellcode 4: drivers/char/random.c

Die Variablen `last_time`, `last_delta` und `last_delta2` werden für die Berechnung der Entropie verwendet. Abschnitt 2.5.3 erörtert die Bedeutung der Werte in diesen Variablen.

Die Variable `dont_count_entropy` ist derzeit bedeutungslos, da sie immer auf den Wert 1 gesetzt wird. Nur wenn dieser Wert 0 ist, wird ein Ereignis der entsprechenden Rauschquelle als neue Entropie verwendet.

2.4 Implementierung der Gerätedateien

Die Gerätedateien `/dev/random` und `/dev/urandom` werden im Kern registriert, indem die Dateisystemoperationen definiert werden – die Datenstruktur mit den Funktionszeigern, welche die verschiedenen Dateioperationen implementieren, wird gefüllt. Beide Gerätedateien werden mit dem folgenden Code registriert:

```

static const struct memdev {
    const char *name;
    mode_t mode;
    const struct file_operations *fops;
    struct backing_dev_info *dev_info;
} devlist[] = {
...
    [8] = { "random", 0666, &random_fops, NULL },
    [9] = { "urandom", 0666, &urandom_fops, NULL },

```

Quellcode 5: drivers/char/mem.c

Alle Gerätedefinitionen in der Variable `devlist` werden während des Startens des Kerns mittels der Funktion `chr_dev_init` aktiviert.

Die Dateioperationen für `/dev/random` sind folgende:

```

const struct file_operations random_fops = {
    .read = random_read,
    .write = random_write,
    .poll = random_poll,
    .unlocked_ioctl = random_ioctl,
    .fasync = random_fasync,
    .llseek = noop_llseek,
};

```

Quellcode 6: drivers/char/random.c

Die sehr ähnlichen, aber nicht gleichen Dateioperationen werden für `/dev/urandom` definiert:

```
const struct file_operations urandom_fops = {
    .read = urandom_read,
    .write = random_write,
    .unlocked_ioctl = random_ioctl,
    .fasync = random_fasync,
    .llseek = noop_llseek,
};
```

Quellcode 7: drivers/char/random.c

Im Folgenden werden die einzelnen Dateioperationen näher erläutert.

2.4.1 random_poll

Der Systemaufruf `poll` erlaubt Prozessen, sich vom Kern aufwecken zu lassen, entsprechend der gesetzten Optionen beim Systemaufruf (Lesen oder Schreiben). Diese Dateioperation wird ausschließlich für `/dev/random` unterstützt:

- `poll` für Lesen: Wenn die Entropieschätzung für den `input_pool` größer ist als ein bestimmter Referenzwert, weckt der Kern den entsprechenden Prozess auf. Dieser Mechanismus erlaubt Prozessen zu schlafen, falls nur unzureichend Entropie für `/dev/random` zur Verfügung steht – in diesem Fall würde die Nutzung des `read` Systemaufrufs den gesamten Prozess blockieren. Der Referenzwert kann zur Laufzeit des Kerns verändert werden, indem eine positive Integer-Zahl in die virtuelle Datei `/proc/sys/kernel/random/read_wakeup_threshold` geschrieben wird. Wenn der Prozess die Option `POLLIN` mit dem Systemaufruf `poll` nutzt, wird der Prozess in die Lesewarteschlange auf die Gerätedateien eingereiht. Siehe [LPM10b] für weiterführende Informationen.
- `poll` für Schreiben: Wenn die Entropieschätzung für den `input_pool` unter einen Referenzwert fällt, weckt der Kern den entsprechenden Prozess auf. Dieser Systemaufruf ist für Prozesse interessant, welche Entropie generieren und dem Kern übergeben: nur wenn die Entropie im Kern zu niedrig ist, muss ein eventuell vorhandener Entropie-Prozess aufgeweckt werden. Ansonsten schläft dieser und nutzt keine weitere CPU-Zeit. Der Referenzwert kann zur Laufzeit des Kerns verändert werden, indem eine positive Integer-Zahl in die virtuelle Datei `/proc/sys/kernel/random/write_wakeup_threshold` geschrieben wird. Wenn der Prozess die Option `POLLOUT` mit dem Systemaufruf `poll` nutzt, wird der Prozess in die Warteschlange für Schreibvorgänge auf die Gerätedateien eingereiht. Siehe [LPM10b] für weiterführende Informationen.

2.4.2 Lesen und Schreiben

Die Schreibfunktion `random_write`, die sowohl für `/dev/random`, als auch für `/dev/urandom` definiert ist, wird im Abschnitt 2.5.1.8 näher erläutert.

Die Lesefunktionen `random_read` für `/dev/random` und `urandom_read` für `/dev/urandom` werden im Abschnitt 2.6.1 dargestellt.

2.4.3 IOCTLs

IOCTL ist eine Abkürzung für input/output control und bezeichnet in vielen Betriebssystemen (einschließlich Linux) Systemaufrufe für gerätespezifische Operationen. Zur genauen Spezifikation der gewünschten Operation wird vom Aufrufer dabei als Parameter ein Request-Code übergeben.

Für beide Gerätedateien wird die gleiche Funktion für die Bearbeitung von IOCTLs registriert: `random_ioctl`. Demzufolge sind für beide Dateien die gleichen IOCTLs nutzbar, welche in Tabelle 1 spezifiziert sind. Diese Tabelle listet in der linken Spalte den Namen des IOCTLs auf. In der Spalte „Privileg“ werden gegebenenfalls geforderte Capabilities des Prozesses spezifiziert (wenn mindestens eine Capability definiert ist, muss der Prozess als privilegiert im System angesehen werden). Die rechte Spalte erklärt den jeweiligen Aufruf.

IOCTL	Privileg	Erklärung
RNDGETENTCNT	-	Lesen der Entropieschätzung des <code>input_pool</code> .
RNDADDDTOENTCNT	CAP_SYS_ADMIN	Die vom aufrufenden Prozess angegebene Integer-Zahl wird zur Entropieschätzung des <code>input_pool</code> entsprechend Quellcode addiert.
RNDADDENTROPY	CAP_SYS_ADMIN	Die vom aufrufenden Prozess übergebenen Werte werden zu dem <code>input_pool</code> auf gleiche Weise wie in Abschnitt 2.5.1.8 beschrieben hinzugefügt. Zusätzlich wird eine vom aufrufenden Prozess angegebene Integer-Zahl zur Entropieschätzung des <code>input_pool</code> entsprechend Quellcode 27 addiert.
RNDZAPENTCNT	CAP_SYS_ADMIN	Alle Entropie-Pools werden erneut initialisiert und damit auf den Ursprungszustand, wie in Abschnitt 2.5.1.6 beschrieben, zurückgesetzt.
RNDCLEARPOOL	CAP_SYS_ADMIN	Siehe RNDZAPENTCNT.

Tabelle 1: IOCTLs für `/dev/random` und `/dev/urandom`

2.4.4 fasync

Für beide Gerätedateien `/dev/random` und `/dev/urandom` wird die generische Kernfunktion für den Systemaufruf `fasync` genutzt und registriert.

Die `fasync` Option beim `open` Systemaufruf wird für asynchrones Aufwecken beim Eintreffen von vorgegebenen Ereignissen verwendet. Es ist zur Zeit unklar, inwieweit dieser Systemaufruf für `/dev/random` oder `/dev/urandom` relevant ist. Die Autoren haben noch keine User-Space-Implementierungen gesehen, welche diesen Systemaufruf mit `/dev/random` oder `/dev/urandom` nutzt.

2.5 Entropie

Das Ziel des LRNG ist:

- das Sammeln von Entropie von verschiedenen Hardwarekomponenten,
- das Hinzufügen der gesammelten Daten in die verschiedenen Pools, und
- die Schätzung der Entropie der gesammelten Daten.

Die folgenden Abschnitte erläutern diese Schritte.

2.5.1 Sammlung von Entropie

Der LRNG definiert Funktionen zur Sammlung von Entropie, welche vom Rest des Kerns verwendet werden können. Diese Funktionen werden an wohldefinierten Quellcode-Abschnitten aufgerufen, um Hardware-Ereignisse zu registrieren und zu sammeln. Sowohl diese Ereignisse, als auch die präzise Ereigniszeit, basierend auf einem hochauflösenden Zeitgeber, werden verwendet, um den `input_pool` zu durchmischen.

Wie im unteren Teil der Abbildung 3 zu sehen ist, werden verschiedene Sammelfunktionen definiert, welche jeweils für eine Klasse von Hardware-Ereignissen zu verwenden sind. Die folgenden Unterabschnitte erläutern jede dieser Sammelfunktionen.

Die in den folgenden Kapiteln speziell markierten Werte identifizieren die Roh-Entropie, welche zum `input_pool` hinzugefügt wird. Auf der Qualität dieser Roh-Entropie basiert die Stärke des vorliegenden LRNGs.

2.5.1.1 add_input_randomness

Die Input-Schicht des Kerns behandelt alle Eingabegeräte, welche lokal angeschlossen sind. Dazu zählen Tastaturen, Mäuse oder Tablets. Wenn der Kern ein Ereignis an einem der Eingabegeräte registriert – zum Beispiel das Drücken oder Loslassen einer Taste oder das

Bewegen der Maus pro Rasterpunkt – ruft der Kern die Funktion `add_input_randomness` auf, um das Ereignis dem LRNG zu melden.

Jedes Ereignis hat einen bestimmten Wert. So zum Beispiel hat jede Taste einen Key Code¹, welcher dem LRNG mittels `add_input_randomness` übergeben wird. Oder wenn die Maus bewegt wird, hat jede Dimension (z.B. links, rechts) und jede Richtung (z.B. vor, zurück) der Maus einen eigenen Wert.

Die Funktion `add_input_randomness` vergleicht den Wert eines aufgenommenen Ereignisses mit dem des vorherigen Ereignisses. Wenn das vorherige Ereignis den gleichen Wert aufweist (z.B. wenn die Maus 2 Rastereinheiten in die gleiche Richtung bewegt wird), wird das neue Ereignis verworfen. Ansonsten wird das Ereignis in den `input_pool` hinzugefügt. Folgender Quellcode zeigt die wichtigen Schritte:

```
void add_input_randomness(unsigned int type, unsigned int code,
                        unsigned int value)
{
    static unsigned char last_value;

    /* ignore autorepeat and the like */
    if (value == last_value)
        return;
    ...
    last_value = value;
    add_timer_randomness(&input_timer_state,
                        (type << 4) ^ code ^ (code >> 4) ^ value);
}
```

Quellcode 8: `drivers/char/random.c`

Der aufgezeigte Code enthält keine Locks, welche die Prüfung auf den Wert des vorherigen Ereignisses gegen gleichzeitig auftretende Ereignisse auf anderen CPUs schützen. Dies ist aber akzeptabel, da im schlimmsten Falle ein Ereignis nicht verworfen wird, das möglicherweise hätte verworfen werden können. Da es bei gleichzeitig auftretenden Ereignissen kaum möglich ist zu sagen, in welcher Reihenfolge diese Ereignisse zu bearbeiten sind, ist der Verzicht auf Locks unkritisch.

Die Funktion `add_input_randomness` nutzt die folgenden Daten, um einen Ereigniswert zu berechnen, welcher entsprechend Abschnitt 2.5.1.5 weiterverwendet wird:

4 niedrigstwertigste Bits des Ereignistyps

⊕

4 höchstwertigste Bits des Ereigniscodes

⊕

Ereigniswert

Dieser Ereigniswert wird mittels statistischer Analysemethoden in Abschnitt 6.2 auf dessen Qualität untersucht.

Die Zeitvarianzen für die Ereignisse von Eingabegeräten werden in der Datenstruktur `input_timer_state` gespeichert, welche in Abschnitt 2.3.1 bereits angesprochen wurde.

2.5.1.2 `add_interrupt_randomness`

Wie der Name richtig suggeriert, werden hier Hardware-Interrupts als Entropiequelle verwendet.

Bevor die Quellcode-Diskussion startet, muss das Konzept der Behandlung von Interrupts geklärt werden. Erst damit ist der Quellcode vollständig zu verstehen.

Die Behandlung kann mit folgender Abbildung erklärt werden:

1 Wenn man präzise ist, hat jede Taste zwei Key Codes: einen, wenn man die Taste drückt und einen, wenn man die Taste loslässt.

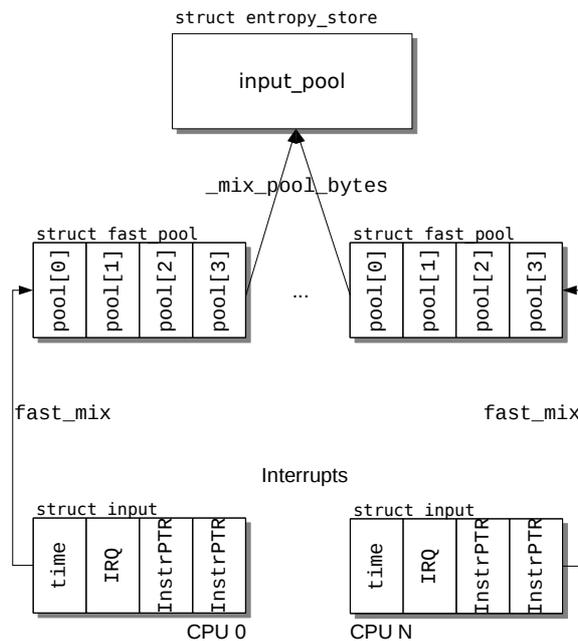


Abbildung 3: fast_pool und input_pool

Der LRNG definiert einen „kleinen Bruder“ der großen Entropie-Pools: den fast_pool. Dieser fast_pool hat vier u32 Wörter und umfasst damit 128 Bit. Des Weiteren wird für den fast_pool die Variable rotate vorgehalten, die konzeptionell identisch mit der input_rotate Variable der großen Entropie-Pools ist.

Es wird eine Instanz des fast_pool pro CPU angelegt. Je nachdem auf welcher CPU ein Interrupt auftritt wird der fast_pool für diese CPU verwendet. Die Ereigniswerte und Zeitstempel jedes Interrupts werden in die jeweiligen fast_pools eingemischt. Nach einer gewissen Anzahl von Interrupts, die zu einem fast_pool hinzugefügt wurden, wird der Inhalt dieses fast_pools direkt in den input_pool eingemischt. Damit umgeht die Funktion add_interrupt_randomness die Zeitstempelfunktion add_timer_randomness, die von den anderen Entropiesammelfunktionen verwendet wird.

Die zentrale Interrupt-Behandlung im Linux-Kern ruft add_interrupt_randomness beim Eintreffen eines Interrupts auf. Der folgende Quellcode zeigt, wie ein Interrupt-Ereignis verarbeitet wird:

```

void add_interrupt_randomness(int irq, int irq_flags)
{
...
    struct fast_pool      *fast_pool = &__get_cpu_var(irq_randomness);
    struct pt_regs        *regs = get_irq_regs();
...
    __u32                  input[4], cycles = random_get_entropy();

    input[0] = cycles ^ jiffies;
    input[1] = irq;
    if (regs) {
        __u64 ip = instruction_pointer(regs);
        input[2] = ip;
        input[3] = ip >> 32;
    }

    fast_mix(fast_pool, input, sizeof(input));
}

```

Quellcode 9: drivers/char/random.c

Als erster Schritt wird eine Referenz auf die `fast_pool` Instanz geholt, welche zur aktuellen CPU gehört. Es ist zu beachten, dass die Top-Halves, welche die Interrupts bearbeiten und die Entropiesammelfunktion aufruft, ebenfalls CPU-gebunden sind. Dies bedeutet, dass ein Interrupt für ein Gerät üblicherweise immer von der gleichen CPU bearbeitet wird.

Anschließend wird die `input`-Datenstruktur aufgebaut, die den Interrupt beschreibt und in den `fast_pool` eingemischt werden soll. Diese `input`-Datenstruktur enthält vier `u32` Wörter:

- Die mittels XOR kombinierten Werten aus Prozessorzyklen und Jiffies. Details zu diesen beiden Werten werden in Abschnitt 2.5.1.5 diskutiert. Im Gegensatz zur Bestimmung der Zeitvarianz von Hardware-Ereignissen entsprechend Abschnitt 2.5.1.5 werden hier absolute Zeitwerte verarbeitet.
- Die IRQ-Nummer des Interrupts wird verwendet.
- Der 64-Bit-Wert des Interrupt-Instruktionsszeigers wird in die beiden verbleibenden `u32` Werte gespeichert.

Diese nun erstellte `input` Datenstruktur wird mittels der Funktion `fast_mix` in den `fast_pool` eingemischt. Folgender Code verdeutlicht dies:

```

static void fast_mix(struct fast_pool *f, const void *in, int nbytes)
{
    const char      *bytes = in;
    __u32           w;
    unsigned        i = f->count;
    unsigned        input_rotate = f->rotate;

    while (nbytes--) {
        w = rol32(*bytes++, input_rotate & 31) ^ f->pool[i & 3] ^
            f->pool[(i + 1) & 3];
        f->pool[i & 3] = (w >> 3) ^ twist_table[w & 7];
        input_rotate += (i++ & 3) ? 7 : 14;
    }
    f->count = i;
    f->rotate = input_rotate;
}

```

Quellcode 10: drivers/char/random.c

Die Implementierung von `fast_mix` arbeitet byteweise. Das bedeutet, dass die vier u32 Bit-Werte der den Interrupt darstellenden `input`-Variable als 16 einzelne Bytes verarbeitet werden. Die Verarbeitung ist konzeptionell identisch mit der Mix-Funktion der „großen“ Entropie-Pools, die in Abschnitt 2.5.2 ausführlich erklärt wird. Es gibt nur einen kleinen Unterschied: anstelle eines Polynoms für die ausgewählten Pool-Wörter werden nur das i -te und das $(i+1)$ -te Wort des Pools verwendet (wobei i das aktuelle Wort des `fast_pools` markiert).

Das Mischen der `fast_pool`-Instanzen stellt aber noch nicht die Verbindung zu den Entropie-Pools `input_pool` oder `nonblocking_pool` her. Diese Verbindung wird durch folgenden Code hergestellt:

```
void add_interrupt_randomness(int irq, int irq_flags)
{
    ...
    if ((fast_pool->count & 1023) &&
        !time_after(now, fast_pool->last + HZ))
        return;

    fast_pool->last = now;

    r = nonblocking_pool.initialized ? &input_pool : &nonblocking_pool;
    __mix_pool_bytes(r, &fast_pool->pool, sizeof(fast_pool->pool), NULL);
    ...
    credit_entropy_bits(r, 1);
}
```

Quellcode 11: `drivers/char/random.c`

Der Code, der die Verbindung zu den Entropie-Pools darstellt, wird nur ausgeführt, wenn folgende Bedingungen für den betrachteten `fast_pool` erfüllt sind:

- Es wurden genau 1024 Bytes in den Pool eingemischt. Da ein Interrupt mit 16 Bytes dargestellt wird, bedeutet diese Bedingung, dass 64 Interrupts in den `fast_pool` eingemischt werden müssen.
- Das letzte Einmischen des `fast_pools` in den Entropie-Pool muss mindestens so lange zurückliegen, wie die Variable `HZ` groß ist, wobei `HZ` eine Variable ist, die zur Kompilierzeit gesetzt wird. Standardmäßig beträgt sie 250 Hz (also 4 Millisekunden). Dies bedeutet, dass ein zeitlicher Mindestabstand zwischen dem Einmischen des `fast_pools` in den Entropie-Pool vorliegen muss.

Nun wird der Entropie-Pool ausgewählt, der mit der neuen Entropie versorgt werden soll. Am Anfang wird direkt der `nonblocking_pool` von den Interrupts versorgt. Das Ziel ist, den User Space beim Startvorgang mit ausreichend Entropie zu versorgen. Erst wenn die Variable `initialized` des `nonblocking_pools` gesetzt ist (was nach dem Erhalten von 128 Bits Entropie geschieht - d.h. der `nonblocking_pool` wird beim Startvorgang mit 128 Bit Entropie gefüllt), wird komplett auf den `input_pool` umgeschaltet. Die Wörter des `fast_pools` werden nun in den ausgewählten Entropie-Pool eingemischt.

Zum Abschluss wird der Entropiezähler des ausgewählten Entropie-Pools um eins erhöht. Dies bedeutet schlussendlich, dass 64 Interrupts² ein Bit an Entropie unterstellt wird.

Abschließend noch ein Wort dazu, wann `add_interrupt_randomness` aufgerufen wird: Die Handler-Funktion, welche die Top-Half für die Behandlung eines Interrupts aufruft, führt auch die `add_interrupt_randomness` Funktion aus.

Ein Interrupt-Ereignis wird also folgendermaßen von `add_interrupt_randomness` in die Entropie-Pools eingemischt: Wenn ein `fast_pool` bereit ist, seine Daten in den Entropie-Pool

² Oder ein Vielfaches davon, wenn der zeitliche Mindestabstand zwischen 64 Interrupts kleiner als `HZ` ist.

einzumischen, werden folgende Werte entsprechend den in Abschnitt 2.5.2 erklärten Mechanismen weiterverwendet:

alle vier u32 Wörter des fast_pool

Dieser Ereigniswert wird mittels statistischer Analysemethoden in Abschnitt 6.2 auf dessen Qualität untersucht.

2.5.1.3 add_disk_randomness

Die dritte Funktion, welche vom LRNG bereitgestellt wird, um Entropie zu sammeln, ist `add_disk_randomness`. Diese Funktion wird von dem zentralen Code-Pfad in der Blockgeräteschicht aufgerufen, der jeden Festplattenzugriff durchführt.

Folgender Quellcode stellt die Verarbeitung von Ereignissen sicher.

```
void add_disk_randomness(struct gendisk *disk)
{
    if (!disk || !disk->random)
        return;
    ...
    add_timer_randomness(disk->random, 0x100 + disk_devt(disk));
}
```

Quellcode 12: drivers/char/random.c

Der erste Schritt ist das Sicherstellen der Integrität des Kerns. Nur wenn die Datenstruktur für die Zeitvarianzen für das Blockgerät definiert ist, wird das Ereignis verwertet. Da jedes Blockgerät eine solche Datenstruktur hat – die Funktion `alloc_disk_node` zur Allokation eines neuen Blockgerätes stellt dies sicher – ist die Prüfung nur eine Absicherung des Kerns gegen Programmierfehler:

```
struct gendisk *alloc_disk_node(int minors, int node_id)
{
    ...
    rand_initialize_disk(disk);
```

Quellcode 13: block/genhd.c

Die Funktion `alloc_disk_node` stellt damit sicher, dass die Datenstruktur mit den Zeitvarianzen entsprechend der Diskussion in Abschnitt 2.3.1 vorhanden ist.

Die aufgerufene Funktion `rand_initialize_disk` belegt einen leeren Speicherbereich und registriert diesen in `disk->random`.

Die Funktion `disk_devt` holt die Variable `disk->devt` der Blockgerätedatenstruktur. Diese Integer-Zahl definiert eineindeutig das Blockgerät im Kern. Die High-Bits dieser Zahl stellen die Major-Nummer dar und die Low-Bits stellen die Minor-Nummer dar.

Ähnlich wie bei der Verwaltung von Interrupts wird die Funktion `add_disk_randomness` nur unter einer Bedingung aufgerufen:

```
static bool blk_update_bidi_request(struct request *rq, int error,
                                   unsigned int nr_bytes,
                                   unsigned int bidi_bytes)
{
    ...
    if (blk_queue_add_random(rq->q))
        add_disk_randomness(rq->rq_disk);
    ...
```

Quellcode 14: block/blk-core.c

Nur wenn die Warteschlange für das Blockgerät das Flag `QUEUE_FLAG_ADD_RANDOM` gesetzt hat (`blk_queue_add_random` verifiziert dieses Flag), wird das Ereignis dem LRNG übergeben.

Standardmäßig ist dieses Flag für jedes Blockgerät gesetzt, einschließlich Solid-State-Disks! Dieses Flag kann aber mittels der Datei „add_random“, welche in /sys für jedes Blockgerät existiert, verändert werden. Nur wenn diese Datei den Wert 1 enthält, werden Ereignisse des entsprechenden Blockgeräts an den LRNG weitergeleitet.

Wenn nun ein Festplattenereignis eintritt, wird die Gerätenummer, welche die Major- und Minor-Nummer des Geräts enthält, wie folgt verwendet, um einen Ereigniswert für das Ereignis des Blockgerätes zu berechnen und entsprechend Abschnitt 2.5.1.5 weiter zu verwenden:

Blockgerätenummer + 0x100

Dieser Ereigniswert wird mittels statistischer Analysemethoden in Abschnitt 6.2 auf dessen Qualität untersucht.

Der Grund, warum Blockgeräte als Entropiequelle verwendet werden, liegt in der physikalischen Beschaffenheit von Blockgeräten und der Qualität des hochauflösenden Zeitgebers im Kern. Der Zeitgeber ist so hochauflösend, dass Zeitvarianzen beim Lesen eines Sektors einer Festplatte gemessen werden können. Solche Varianzen treten typischerweise auf, wenn die sich drehende Platte und die Leseköpfe nicht exakt an der gleichen Position befinden wie bei dem letzten Zugriff. Die Zeit, bis die Platte sich so weit dreht, damit der Sektor gelesen werden kann, ist damit anders. Obwohl die Zeitunterschiede sehr klein sind, können diese gemessen werden, da der Zeitgeber im Kern eine viel höhere Auflösung hat, als die genannten Varianzen.

Die Entropiegewinnung basierend auf den beschriebenen Zeitvarianzen fällt allerdings in sich zusammen, wenn Lesezugriffe auf Blockgeräte nicht mehr solchen physikalischen Schwankungen unterliegen. Dies trifft auf Solid-State-Disks und USB-Sticks zu. Für diese Geräte muss die oben genannte Datei add_random auf 0 gesetzt werden. Standardmäßig werden alle Blockgeräte, die für den Kernel als physische Geräte vorliegen, zur Entropiegewinnung verwendet, d.h. die Datei add_random enthält eine 1. Aufgrund der fehlenden physikalischen Voraussetzungen zur Entropiegewinnung, sollten für folgende Geräte die entsprechende add_random Datei eine Null enthalten:

- Solid-State-Disks,
- Flash-RAM basierte Blockgeräte, wie USB-Sticks, SD-Karten, Compact-Flash-Karten,
- Jeglicher Flash-RAM, welcher zum Beispiel in eingebetteten Geräten zu finden ist,

Virtualisierte Festplatten hingegen sind als unkritisch anzusehen, sofern sie „normale“ Festplatten mit den oben genannten physikalischen Charakteristika sind. Der Hintergrund ist, dass zwar sowohl das Hostsystem diese Festplattenzugriffe für seinen eigenen LRNG verwendet, als auch der virtualisierte Gast. Da aber beide LRNGs getrennt sind und die Nutzer der LRNGs ebenfalls separat sind, kann eine entsprechende synchrone Rauschquelle akzeptiert werden. Die LRNGs der Gastsysteme untereinander sind als unabhängig anzusehen, da der Gastkern ja nur Entropie gewinnt, wenn dieser Kern einen Festplattenzugriff durchführt. Wenn nun ein anderer Gastkern auf die gleiche oder eine andere Festplatte zugreift, ist dieser Zugriff nicht für andere Gäste sichtbar.

2.5.1.4 add_device_randomness

Im Gegensatz zu den vorher diskutierten Entropiesammelfunktionen ist das Ziel von add_disk_randomness, ausschließlich den input_pool und nonblocking_pool zur Initialisierungszeit der aufrufenden Gerätetreiber weiter zu durchmischen. Dies bedeutet, dass add_disk_randomness **nur einmal** von Gerätetreibern zur Initialisierungszeit der Treiber mit treiberspezifischen Daten aufgerufen wird.

Diese Entropiesammelfunktion muss von Gerätetreibern, welche relativ zufällige Daten während der Initialisierung vorfinden, direkt aufgerufen werden. Diese zufälligen Daten werden direkt in die beiden Entropie-Pools eingemischt gefolgt dem Einmischein eines Zeitstempels in beide Entropie-Pools.

Die Entropieschätzung der Entropie-Pools wird nicht verändert.

Ein Gerätetreiber-Ereignis wird also folgendermaßen in `add_device_randomness` mit der Mix-Funktion entsprechend Abschnitt 2.5.2 verarbeitet:



2.5.1.5 `add_timer_randomness`

Abbildung 3 stellt klar, dass die in den vorangegangenen Kapiteln diskutierten Entropiesammelfunktionen ihre Werte an `add_timer_randomness` weitergeben. Diese Funktion hat zum Ziel, die Zeitvarianzen zum vorherigen Ereignis auf Basis der in Abschnitt 2.3.1 diskutierten Datenstruktur zu berechnen. Das Resultat dieser Zeitvarianzberechnung wird zusammen mit den aus den Entropiesammelfunktionen berechneten Werten zum `input_pool` hinzugefügt. Der Code stellt sicher, dass weder `blocking_pool` noch `nonblocking_pool` mit Roh-Entropie aus den Entropiesammelfunktionen und dieser Zeitvarianzberechnung gespeist werden.

Die Zeitvarianz wird mittels folgendem Code berechnet:

```
static void add_timer_randomness(struct timer_rand_state *state, unsigned num)
{
    struct {
        long jiffies;
        unsigned cycles;
        unsigned num;
    } sample;
    ...
    sample.jiffies = jiffies;

    sample.cycles = random_get_entropy();
    sample.num = num;
    mix_pool_bytes(&input_pool, &sample, sizeof(sample));
}

```

Quellcode 15: `drivers/char/random.c`

Der Code erzeugt eine Datenstruktur, welche:

- den derzeitigen Jiffies-Wert mit 64 Bit Größe auf einem 64 Bit System,
- einen Zeitstempel mit 32 Bit Größe, welcher mit `random_get_entropy` berechnet wird³,
- den Ereigniswert, welcher aus den vorher beschriebenen Entropiesammelfunktionen erzeugt wurde,

verwendet. Es muss beachtet werden, dass der Compiler ein Padding der Datenstruktur durchführt, sodass die einzelnen Variablen 8 Bytes benötigen. Demzufolge ist ein `sizeof(sample)` immer noch 24 Byte auf eine 64 Bit System.

Zum Füllen der Variable `cycles` implementiert der aufgezeigte Quellcode folgende Logik: Falls ein Hardware-Zufallszahlengenerator entsprechend der Diskussion in Abschnitt 2.7 vorliegt, wird der anstelle des Zeitstempels ein Wert aus diesem Hardware-Zufallszahlengenerator verwendet. Ohne einen Hardware-Zufallszahlengenerator wird der Zeitstempel basierend auf der Funktion `random_get_entropy` verwendet.

³ Zu beachten hier ist, dass `random_get_entropy` einen 64 Bit-Wert zurückliefert, welcher automatisch in einen 32-Bit-Wert umgewandelt wird. Der GCC verwirft hierbei die 32 höherwertigen Bits und kopiert die 32 niederwertigen Bits in die Variable.

Die Funktion `random_get_entropy` ist eine hardwareabhängige Methode, um den CPU-Zeitgeber auszulesen. Diese Funktion nutzt die folgenden Prozessorfunktionen der entsprechenden CPU Architekturen:

- RDTSC (Read Time Stamp Counter)-Instruktion auf Intel-x86- und AMD-Opteron-Prozessoren
- STCK (Store Clock)-Instruktion auf den IBM System-Z-Prozessoren
- MFTB (Move From Time Base)-Instruktion auf PowerPC-kompatiblen Prozessoren
- Auf ARM-Systemen wird der Register Wert aus dem internen Coprozessor P15 ausgelesen, welcher mittels Opcode 0 und CRm = c14 ausgelesen wird:

```
static inline cycle_t arch_counter_get_cntpct(void)
{
    u32 cval1, cvalh;

    asm volatile("mrrc p15, 0, %0, %1, c14" : "=r" (cval1), "=r" (cvalh));

    return ((cycle_t) cvalh << 32) | cval1;
}
```

Quellcode 16: `arch/arm/kernel/arch_timer.c`

Jede der genannten Instruktionen liefert einen 64-Bit-Wert als Zeitstempel zurück. Auf den Intel- und AMD-Prozessoren wird dieser Zeitgeber bei jedem Taktzyklus um eins erhöht, selbst wenn eine HLT Instruktion verarbeitet wird. Dies bedeutet, dass der Zeitgeber ca. 1 Milliarde Zählungen pro Sekunde auf einem 1GHz-Prozessor durchführt. Der Wert des Zeitgebers wird auf null gesetzt, wenn der Prozessor einen Reset durchführt.

Auf der System-Z-Architektur wird der Zeitgeber alle 2^{-12} Mikrosekunden (d.h. alle 244 Pikosekunden) erhöht. Der Wert ist in dem Time-Of-Day (TOD)-Zeitgeber gespeichert, welcher beim Starten des Kerns initialisiert wird.

Auf einer PowerPC-CPU wird der Zeitgeber nach jeweils 32 Taktzyklen inkrementiert. Dies resultiert in 31.250.000 Zählungen pro Sekunde auf einem 1 GHz Prozessor. Der Wert des Zeitgebers wird auf null gesetzt, wenn der Prozessor einen Reset durchführt.

Der direkte Zugang zu den Hardwarezeitgebern eliminiert potentielle Probleme von software-basierten Zeitgebern wie Zeitvarianzen, die durch das Ausführen der Zeitgeber-Software entstehen. Zudem bieten die genannten Zeitgeber die höchste verfügbare Auflösung auf der jeweiligen CPU.

Diese Datenstruktur wird nun der in Abschnitt 2.5.2 beschriebenen Funktion übergeben. Diese Funktion konvertiert diese Datenstruktur in einen void-Zeiger um. Demzufolge werden die Daten in der `sample` Datenstruktur als einfacher Bytestrom interpretiert, welcher die Teilwerte von `sample` konkateniert. Damit wird nun folgender Wert als Roh-Entropie zum `input_pool` hinzugefügt:

Zeitstempel . Jiffies . Ereigniswert⁴

Dieser Wert wird mittels statistischer Analysemethoden in Abschnitt 6.2 auf dessen Qualität untersucht.

Neben der Erzeugung der Roh-Entropie führt `add_timer_randomness` auch die Berechnung der Entropieschätzung für das eben verarbeitete Ereignis durch. Die Diskussion der Entropieschätzung erfolgt in Abschnitt 2.5.3.

2.5.1.6 Initialisierung

Während der Initialisierung des LRNG beim des Starten des Kerns werden alle Entropie-Pools mit Werten initialisiert, damit sie auf jeden Fall nicht leer sind. Die Funktion `rand_initialize`

⁴ „.“ wird als Konkatenationsoperator verwendet, ähnlich der Nutzung in Perl.

implementiert die Initialisierung des LRNG und wird während des Startvorgangs des Kerns aufgerufen. Diese Funktion ruft `init_std_data` für jeden der Entropie-Pools auf:

```
static int rand_initialize(void)
{
    init_std_data(&input_pool);
    init_std_data(&blocking_pool);
    init_std_data(&nonblocking_pool);
    return 0;
}
Quellcode 17: driver/char/random.c
```

```
static void init_std_data(struct entropy_store *r)
{
    ...
    r->entropy_count = 0;
    ...
    now = ktime_get_real();
    mix_pool_bytes(r, &now, sizeof(now));
    for (i = r->poolinfo->POOLBYTES; i > 0; i -= sizeof(flags)) {
        if (!arch_get_random_long(&flags))
            break;
        mix_pool_bytes(r, &flags, sizeof(flags));
    }
    mix_pool_bytes(r, utsname(), sizeof(*(utsname())));
}
Quellcode 18: drivers/char/random.c
```

Die Funktion `init_std_data` umfasst folgende wichtige Schritte: Mit dem abgebildeten Code werden die folgenden Schritte pro Entropie-Pool umgesetzt:

- Die Entropieschätzung wird für den entsprechenden Pool auf 0 gesetzt. Dies bedeutet, dass der Kern annimmt, dass sich in diesem Pool keine Entropie befindet.
- Die aktuelle Zeit wird gelesen und zu dem Pool hinzugefügt. Die Auflösung der Zeit kann im Kernquellcode nachgelesen werden:

```

/*
 * ktime_t:
 *
 * On 64-bit CPUs a single 64-bit variable is used to store the hrtimers
 * internal representation of time values in scalar nanoseconds. The
 * design plays out best on 64-bit CPUs, where most conversions are
 * NOPs and most arithmetic ktime_t operations are plain arithmetic
 * operations.
 *
 * On 32-bit CPUs an optimized representation of the timespec structure
 * is used to avoid expensive conversions from and to timespecs. The
 * endian-aware order of the tv struct members is chosen to allow
 * mathematical operations on the tv64 member of the union too, which
 * for certain operations produces better code.
 *
 * For architectures with efficient support for 64/32-bit conversions the
 * plain scalar nanosecond based representation can be selected by the
 * config switch CONFIG_KTIME_SCALAR.
 */

```

Quellcode 19: include/linux/ktime.h

- Falls ein Hardware-Zufallszahlengenerator von der Hardware angeboten wird, werden so viele Zufallszahlen aus diesem Generator zu dem Entropie-Pool hinzugefügt, wie der Entropie-Pool groß ist. Damit wird der Entropie-Pool weiter durchmischt, um einen vorher nicht bestimmbareren Zustand zu erreichen.
- System-spezifische Informationen werden gelesen und zu dem entsprechenden Entropie-Pool hinzugefügt:

```

struct new_utsname {
    char sysname[__NEW_UTS_LEN + 1];
    char nodename[__NEW_UTS_LEN + 1];
    char release[__NEW_UTS_LEN + 1];
    char version[__NEW_UTS_LEN + 1];
    char machine[__NEW_UTS_LEN + 1];
    char domainname[__NEW_UTS_LEN + 1];
};

```

Quellcode 20: include/linux/utsname.h

Die Bezeichnung der Variablen in `new_utsname` gibt bereits einen Hinweis auf die enthaltene Information. Zu bemerken ist, dass beim Starten des Kerns einige Variablen noch leer sind (so zum Beispiel der Hostname). Man muss aber festhalten, dass diese Informationen als öffentlich und deterministisch anzusehen sind.

2.5.1.7 Nutzung des LRNG

Die Sammlung und Verarbeitung der Entropie mit der LRNG-Implementierung basiert auf einigen Annahmen, wie sich das System verhält. Von diesen Annahmen lassen sich Nutzungsregeln ableiten, die unbedingt einzuhalten sind, damit die Qualität der Daten aus `/dev/random` oder `/dev/urandom` den in Kapitel 5 getroffenen Aussagen entspricht..

Eine Liste der Regeln im Umgang mit dem LRNG ist in [Mü12] vollständig dokumentiert. Einen kurzen Überblick über diese Regeln gibt die folgende Liste:

- Dem LRNG muss beim Starten ein Seed eingemischt werden.
- Vor dem Ausschalten des Systems müssen Zufallszahlen aus dem LRNG als Seed für den nächsten Startvorgang gespeichert werden.

- Die während der initialen Installation erzeugte Entropie muss für den ersten Startvorgang als Seed gespeichert werden.
- Bei der Konfiguration einer Full Disk Encryption, muss zusätzliche Entropie für den LRNG angefordert werden.
- Bei der Nutzung von LiveCDs muss während des Startvorgangs anstelle von Seed zusätzliche Entropie für den LRNG angefordert werden.
- Wenn bestimmte Blockgeräte verwendet werden, dürfen diese nicht als Entropie-Quelle verwendet werden. Dies gilt für alle Blockgeräte, die keine Zeitvarianzen beim Lesen oder Schreiben aufgrund von physikalischen Eigenschaften aufweisen. Vor allem gilt dies für virtuelle Umgebungen und SSDs, sowie Flash RAM.

2.5.1.8 Schreiben von Daten in /dev/(u)random

Beide Gerätedateien, /dev/random und /dev/urandom, erlauben User-Space-Prozessen Daten in diese Dateien zu schreiben. Normal konfigurierte Linux-Systeme erlauben jedem unprivilegierten Prozess, beliebige Datenmengen in beide Dateien zu schreiben.

Die Funktion, welche das Schreiben von Daten verarbeitet, ist identisch für beide Gerätedateien, wie in den Quellcode-Abschnitten 6 und 7 zu sehen ist: es wird die Funktion `random_write` verwendet.

Die Funktion `random_write` ruft die Hilfsfunktion `write_pool` auf, um die Daten zu den Entropie-Pools hinzuzufügen. Unabhängig davon, ob die Schreibfunktion via /dev/random oder /dev/urandom aufgerufen wird, werden die übergebenen Daten immer zu den Entropie-Pools `blocking_pool` und `nonblocking_pool` hinzugefügt.

Die Funktion `write_pool` fügt Daten zu dem angegebenen Entropie-Pool wie folgt hinzu:

```
static int
write_pool(struct entropy_store *r, const char __user *buffer, size_t count)
{
...
    __u32 buf[16];
...
    while (count > 0) {
        bytes = min(count, sizeof(buf));
...
        count -= bytes;
...
        mix_pool_bytes(r, buf, bytes);
...
    }
}

```

Quellcode 21: drivers/char/random.c

Der Quellcode zeigt, dass die vom User-Space bereitgestellten Daten in Blöcken zu maximal 16 Bytes zum Entropie-Pool hinzugefügt werden. Das eigentliche Hinzufügen der Daten zum Entropie-Pool erfolgt mit der Funktion `mix_pool_bytes`, welche in Abschnitt 2.5.2 erklärt wird.

Es ist wichtig klarzustellen, dass die Entropieschätzung in dem gesamten Schreibvorgang nicht verändert wird. Dies bedeutet, dass der Entropie-Pool verändert wird, ohne dass der LRNG dieser Veränderung irgend welche Entropie beimisst. Wenn der Aufrufer die Entropieschätzung verändern möchte, muss er einen in Abschnitt 2.4.3 erläuterten IOCTL verwenden.

Ein wichtiger Fakt ist abschließend noch zu nennen: der `input_pool` wird niemals durch das Schreiben von Daten in /dev/random oder /dev/urandom verändert.

2.5.2 Hinzufügen von Daten zu Entropie-Pools

In den vorangegangenen Kapiteln wurden eingehend die Quellen für Daten diskutiert, welche vom LRNG verarbeitet werden. Nachdem die Daten aus den Rauschquellen extrahiert wurden, müssen sie nun zu den entsprechenden Entropie-Pools hinzugefügt werden.

Da alle Entropie-Pools auf der gleichen Datenstruktur basieren, implementiert der LRNG die Logik für das Hinzufügen von neuen Daten zu einem Entropie-Pool nur einmal. Die folgende Diskussion gilt demzufolge gleichermaßen für `input_pool`, `blocking_pool` und `nonblocking_pool`. Die folgenden Funktionen müssen vom Aufrufer mit einer Referenz auf den entsprechenden Entropie-Pool aufgerufen werden. Entsprechend der vorangegangenen Abschnitte und Abbildung 3 werden die folgenden Entropie-Sammler verwendet:

- Der `input_pool` wird von den Funktionen aufgerufen, die Hardware-Ereignisse sammeln:
 - `add_input_randomness` via `add_timer_randomness`,
 - `add_interrupt_randomness`,
 - `add_disk_randomness` via `add_timer_randomness`,
 - `init_std_data` - siehe Abschnitt 2.5.1.6 für eine Erklärung der verarbeiteten Informationen.
- Die Varianten für `blocking_pool` und `nonblocking_pool` werden von den Funktionen aufgerufen, die mit dem User-Space interagieren:
 - `random_write` via `write_pool`,
 - `init_std_data` - siehe Abschnitt 2.5.1.6 für eine Erklärung der verarbeiteten Informationen.

Die Ausnahme ist die Funktion `xfer_secondary_pool`, welche auch Daten in einen Entropie-Pool hinzufügt. Entsprechend der Diskussion in Abschnitt 2.6 werden entweder der `blocking_pool` oder der `nonblocking_pool` mit Daten aus dem `input_pool` verändert.

Daten werden zu den jeweiligen Entropie-Pools mittels der Funktion `mix_pool_bytes` hinzugefügt, welche das Locking vornimmt und sofort `_mix_pool_bytes` aufruft. Diese Funktion implementiert gleichzeitig zwei Mechanismen:

- Hinzufügen von Daten in den Entropie-Pool
- Auslesen der Rohwerte des Entropie-Pools, nachdem die Daten hinzugefügt wurden

Der letztere Mechanismus wird nicht verwendet, wenn Daten aus den Entropiequellen hinzugefügt werden.

Die Kernlogik von `_mix_pool_bytes` ist ein lineares Schieberegister, dessen Daten noch mit einem „Twist“ verarbeitet werden. Der folgende Quellcode zeigt die wichtigsten Schritte:

```

static __u32 const twist_table[8] = {
    0x00000000, 0x3b6e20c8, 0x76dc4190, 0x4db26158,
    0xedb88320, 0xd6d6a3e8, 0x9b64c2b0, 0xa00ae278 };
...
* The pool is stirred with a primitive polynomial of the appropriate
* degree, and then twisted. We twist by three bits at a time because
* it's cheap to do so and helps slightly in the expected case where
* the entropy is concentrated in the low-order bits.
*/
static void _mix_pool_bytes(struct entropy_store *r, const void *in,
                           int nbytes, __u8 out[64])
{
...
    input_rotate = ACCESS_ONCE(r->input_rotate);
    i = ACCESS_ONCE(r->add_ptr);
    /* mix one byte at a time to simplify size handling and churn faster */
    while (nbytes--) {
        w = rol32(*bytes++, input_rotate & 31);
        i = (i - 1) & wordmask;

        /* XOR in the various taps */
        w ^= r->pool[i];
        w ^= r->pool[(i + tap1) & wordmask];
        w ^= r->pool[(i + tap2) & wordmask];
        w ^= r->pool[(i + tap3) & wordmask];
        w ^= r->pool[(i + tap4) & wordmask];
        w ^= r->pool[(i + tap5) & wordmask];

        /* Mix the result back in with a twist */
        r->pool[i] = (w >> 3) ^ twist_table[w & 7];

        /*
         * Normally, we add 7 bits of rotation to the pool.
         * At the beginning of the pool, add an extra 7 bits
         * rotation, so that successive passes spread the
         * input bits across the pool evenly.
         */
        input_rotate += i ? 7 : 14;
    }
...

```

Quellcode 22: drivers/char/random.c

Die Funktion verarbeitet die Eingabewerte entsprechend den folgenden Schritten und verändert damit das von `r->pool` angezeigte Feld, also entsprechend Abschnitt 2.3 den Inhalt der Rohform des Entropie-Pools.

- Hole ein Byte der Eingabedaten mit dem Offset entsprechend der Anzahl der Schleifendurchläufe. Der erste Schleifendurchlauf impliziert 1 Byte Offset, der zweite Durchlauf hat 2 Byte Offset. Das Byte wird in eine vier Byte große Variable mittels der `rol32()` Funktion konvertiert. Das Eingabe-Byte wird um eine Anzahl von Bits nach links rotiert, was durch `(input_rotate AND 31)` berechnet wird. Wie der Wert `input_rotate` zustande kommt, wird weiter unten beschrieben. Zum Beispiel: `input_rotate` ist Null, dann wird das betrachtete Byte in dem 32 Bit Array rechts positioniert und alle übrigen, links stehenden 24 Bits mit Null aufgefüllt. Wenn `input_rotate` 2 ist, sieht das Bit Array folgendermaßen aus: 22 Null Bits, Eingabebyte, 2 Null Bits. Abschnitt 7.2 stellt eine detaillierte Analyse von `input_rotate` zur Verfügung.

- Der Index auf den Pool wird geholt, welcher auf `r->add_ptr` basiert. Dieser Index zeigt auf den zuletzt modifizierten Wert des Pools. Dieser Index wird um eins dekrementiert, um auf den nächsten Wert zu zeigen, welcher nun modifiziert wird (d.h. der Index wird nach links verschoben). Natürlich wird ein Index-Umbruch durchgeführt, wenn der neue Indexwert außerhalb des Größenbereiches des Pools liegt.
- Nun werden die im Schritt 1 erzeugten 4 Bytes mittels XOR mit den folgenden Werten verknüpft:
 - derzeitiger Wert des Pools an der Stelle, auf die der Index zeigt
 - derzeitiger Wert des Pools an der Stelle auf die der Index plus dem ersten Polynomwert zeigt. Dieser Indexwert wird gegebenenfalls umgebrochen, um im Bereich des Pools zu bleiben.
 - Schritt 3.b) wird für die Polynomwerte zwei bis fünf wiederholt.

Folgende Abbildung illustriert die Verwendung des Polynoms anhand eines Beispiels. Das Beispiel aktualisiert den Pool `input_data->input_pool_data`. Zu beachten ist, dass für den `input_pool` das erste Polynom aus dem Quellcode 2 verwendet wird. Das angegebene Beispiel gilt analog für die anderen Entropie-Pools `blocking_pool` und `nonblocking_pool` mit der Abweichung, dass das zweite Polynom aus Quellcode 2 verwendet wird. Das Beispiel nimmt an, dass der Index des Wertes, der verändert wird, 40 ist (d.h. `i==40` entsprechend dem Quellcode 22):

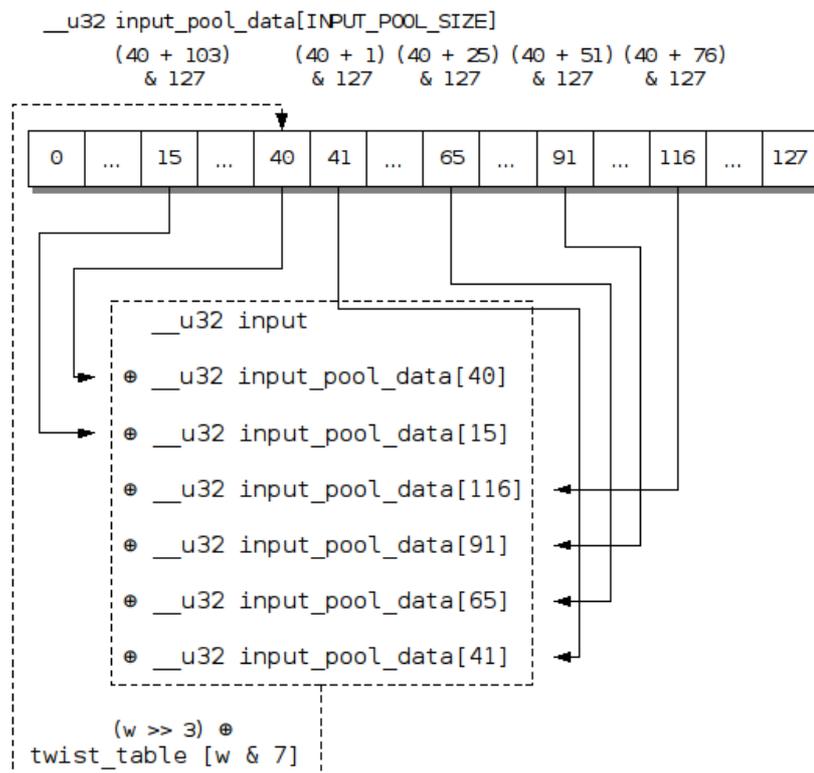


Abbildung 4: Berechnung eines neuen Pool-Wertes für den `input_pool`

- Der in Schritt 3 erzeugte Wert wird nun nochmal mit dem Twist-Wert aus der Variable `twist_table` mittels XOR kombiniert.
- Nun wird der in Schritt 4 berechnete Wert als neuer Wert in den Pool an der Indexstelle eingesetzt.

- Der Wert `input_rotate` für Schritt eins wird nun berechnet: es werden 7 Bits als Rotationswert hinzugefügt, außer wenn der Index auf den Anfang zeigt (also null ist), dann werden 14 Bits als Rotationswert verwendet.
- Die Schritte 1 bis 6 werden nun für alle noch verbleibenden Bytes des Eingabedatenstroms wiederholt.

Wie schon in der Erklärung der Schritte ersichtlich, wird hier die Entropieschätzung nicht aktualisiert.

2.5.3 Abschätzung der Entropie

Die Gesamtheit aller vorangegangenen Abschnitte hat sich mit dem Problem beschäftigt, wie Roh-Entropie gesammelt wird und wie diese verarbeitet wird. Ein wichtiger Punkt blieb bis jetzt aber noch offen: welche Heuristik verwendet der LRNG, um die gewonnene Entropie zu quantifizieren? Wie in der Einleitung bereits erläutert und in Abschnitt 2.6.1 genau beschrieben, blockiert die Lesefunktion von `/dev/random` jeden Prozess, wenn der Kern annimmt, dass zu wenig Entropie vorhanden ist.

Abschnitt 2.5.2 hat bereits klargestellt, dass das Hinzufügen von Daten in die Pools nicht automatisch zu einer Aktualisierung der Entropieschätzung führt.

Die Abschätzung der Entropie pro gesammeltem Ereignis wird im zweiten Teil der Funktion `add_timer_randomness` durchgeführt. Der erste Teil dieser Funktion, die Definition der Roh-Entropie, wurde bereits in Abschnitt 2.5.1.5 erläutert und ist nicht Gegenstand dieses Kapitels.

Da nur `add_timer_randomness` die Entropieschätzung durchführt, werden Daten für die Entropie-Pools aus anderen Quellen (z.B. das Schreiben von Daten in die Gerätedateien `/dev/random` oder `/dev/urandom`) nicht als Quelle von Entropie angesehen⁵.

Der LRNG misst nur die Entropie des primären Entropie-Pools `input_pool`. Die beiden anderen Entropie-Pools unterliegen nicht der Entropieschätzung. Die Entropieschätzung basiert auf dem Konzept, dass der LRNG die Entropie des einzelnen Ereignisses abschätzt und diese Entropieschätzung zu dem bereits vorhandenen Wert der Entropieschätzung addiert.

Die Schätzung der Entropie eines Ereignisses wird standardmäßig nur alle 4096 Ereignisse durchgeführt. Dabei werden die Ereignisse pro CPU gezählt. Falls die Entropieschätzung aber unter den Schwellwert von 3584 fällt, der durch folgenden Code definiert ist,

```
#define INPUT_POOL_WORDS 128
...
/*
 * When the input pool goes over trickle_thresh, start dropping most
 * samples to avoid wasting CPU time and reduce lock contention.
 */

static int trickle_thresh __read_mostly = INPUT_POOL_WORDS * 28;

static DEFINE_PER_CPU(int, trickle_count);

Quellcode 23: drivers/char/random.c
```

wird für jedes Ereignis die Entropie abgeschätzt und verwendet. Folgender Code setzt diesen Schwellwert um, wobei die Variable `trickle_count` für jede CPU separat definiert ist:

5 Natürlich kann ein privilegierter Prozess den Entropieschätzer verändern, indem ein spezieller, in Abschnitt 2.4.3 diskutierter IOCTL verwendet wird. Da dieser aber nicht standardmäßig in laufenden Systemen verwendet wird, kann dieser IOCTL hier vernachlässigt werden.

```
static void add_timer_randomness(struct timer_rand_state *state, unsigned num)
{
...
    /* if over the trickle threshold, use only 1 in 4096 samples */
    if (input_pool.entropy_count > trickle_thresh &&
        ((__this_cpu_inc_return(trickle_count) - 1) & 0xffff))
        goto out;
...

```

Quellcode 24: drivers/char/random.c

Wenn der eben besprochene Quellcode das Ereignis annimmt (d.h. der Schwellwert ist nicht erreicht), wird das Ereignis, wie in dem Abschnitt 2.5.1.5 erläutert, zu dem `input_pool` hinzugefügt. Nachdem der `input_pool` entsprechend verändert wurde, wird nun die Entropieabschätzung dieses Ereignisses durchgeführt. Dabei werden die Zeitvarianzen zu den vorangegangenen Ereignissen gebildet, die in der Zeitvarianz-Datenstruktur entsprechend Abschnitt 2.3.1 gespeichert werden.

Die Ereigniszeit des derzeitigen Ereignisses ist t_n .

Die Ereigniszeit des vorangegangenen Ereignisses ist t_{n-1} .

Entsprechend werden die Ereigniszeiten der weiter vorher liegenden Ereignisse mit t_{n-2} und t_{n-3} definiert.

Die Ereigniszeit und die Deltas werden in der in Abschnitt 2.3.1 diskutierten Zeitvarianz-Datenstruktur für die verschiedenen Klassen von Ereignissen gespeichert.

Der Kern berechnet nun die folgenden Werte:

- $delta = |t_n - t_{n-1}|$
- $delta_2 = |(t_n - t_{n-1}) - (t_{n-1} - t_{n-2})| = |t_n - 2t_{n-1} + t_{n-2}|$
- $delta_3 = |((t_n - t_{n-1}) - (t_{n-1} - t_{n-2})) - ((t_{n-1} - t_{n-2}) - (t_{n-2} - t_{n-3}))| = |t_n - 3t_{n-1} + 3t_{n-2} - t_{n-3}|$

Basierend auf diesen Deltas wird nun die Entropieschätzung `input_pool->entropy_count` wie folgt verändert:

- Verwendung des kleinsten Betrags der Werte von `delta`, `delta2`, `delta3`, d.h. Berechnung von $delta = \min(delta, delta_2, delta_3)$:

```
static void add_timer_randomness(struct timer_rand_state *state, unsigned num)
{
...
    if (delta > delta2)
        delta = delta2;
    if (delta > delta3)
        delta = delta3;
...

```

Quellcode 25: drivers/char/random.c

- Dividieren von `delta` durch 2; dadurch wird in der nachfolgenden Berechnung das Maß der gewonnenen Entropie um 1 Bit verringert.
- Das Maß der gewonnenen Entropiebits entspricht dem höchsten gesetzten Bit in `delta`. Zum Beispiel ist für den Wert `00010110` das Entropiemaß 5, weil das höchstwertige gesetzte Bit an der 5. Stelle von rechts steht.
- Das Entropiemaß wird auf ein Maximum von 11 begrenzt.

Die Schritte 2 bis 4 werden durch folgenden Code implementiert:

```
static void add_timer_randomness(struct timer_rand_state *state, unsigned num)
{
...
        /*
         * delta is now minimum absolute delta.
         * Round down by 1 bit on general principles,
         * and limit entropy estimate to 12 bits.
         */
        credit_entropy_bits(&input_pool,
                           min_t(int, fls(delta>>1), 11));
...

```

Quellcode 26: drivers/char/random.c

- Die aus dem Ereignis gewonnene Entropie wird nun zur Entropieschätzung addiert. Dabei wird berücksichtigt, dass die Entropieschätzung maximal den Wert 4096 annehmen darf:

```
static void credit_entropy_bits(struct entropy_store *r, int nbits)
{
...
        entropy_count = orig = ACCESS_ONCE(r->entropy_count);
        entropy_count += nbits;
        if (entropy_count < 0) {
...
                entropy_count = 0;
        } else if (entropy_count > r->poolinfo->POOLBITS)
                entropy_count = r->poolinfo->POOLBITS;
        if (cmpxchg(&r->entropy_count, orig, entropy_count) != orig)

```

Quellcode 27: drivers/char/random.c

Die Berechnung der Entropieschätzung für ein Ereignis impliziert folgendes:

- Ein Ereignis hat maximal eine Entropie von 11 (siehe Schritt 4).
- Die minimale Zeitvarianz zwischen den vier betrachteten Ereignissen wird als Entropie definiert.

Die verwendete Methode ist eine Heuristik, die annimmt, dass die niederwertigen Bits der Zeitvarianzen von Hardware-Ereignissen unvorhersagbar sind. Selbst das Ausführen von zwei identische Codesequenzen auf einem „ruhigen“ System resultiert in unterschiedlichen Ausführungszeiten, und damit in unterschiedlichen Zeitvarianzen. Der Grund für diese Varianzen sind Interrupts, welche vom Kern unbedingt behandelt werden müssen und damit die Ausführung der betrachteten Codesequenz kurzzeitig unterbrechen. Ebenfalls hat der Zustand des Prozessors (Inhalt der Caches, der Branch-Prediction-Unit, des TLB, etc.) einen Einfluss auf die Ausführungszeit.

2.5.3.1 Einheit der Entropieschätzung und theoretische Grundlagen

Der bisher dargelegte Code definiert die Einheit der Entropieschätzung nicht. Demzufolge ist hier nicht ersichtlich, ob die Entropieschätzung den Wert in Bits, Bytes oder in anderen Einheiten interpretiert.

Um aber Klarheit zu erreichen, werden einige Informationen der folgenden Kapitel vorweg genommen: die Entropieschätzung verwaltet die Entropie in Bits. Dies ist ersichtlich, wenn man die Extraktion von Zufallszahlen betrachtet. Den Extraktionsfunktionen muss der Aufrufer die Größe der gewünschten Zufallszahl in Bytes mitteilen. Bei Vergleichen zwischen diesem Größenwert und der Entropieschätzung wird ersterer mit 8 multipliziert:

```
static void xfer_secondary_pool(struct entropy_store *r, size_t nbytes)
{
...
    if (r->pull && r->entropy_count < nbytes * 8 &&
...
    Quellcode 28: drivers/char/random.c
```

Zur Analyse und Bewertung des LRNG benötigt man zudem messbare Vergleichsgrößen. Hierzu stehen unterschiedliche, mehr oder weniger gut geeignete Definitionen des Entropiebegriffs zur Verfügung. In diesem Bericht wurde, wie in den meisten bisherigen Analysen des LRNG auch, die Definition von Shannon verwendet. Es hat sich jedoch herausgestellt (siehe dazu auch die Arbeit [LRSV12] von Lacharme et al.), dass die Shannon-Entropie für Bewertungen und Analysen des LRNG unter bestimmten Bedingungen geeignet ist, jedoch für die theoretische Erklärung in vielen Fällen ungeeignet ist. Der Grund ist, dass für die Berechnung der Shannon-Entropie bereits zu Beginn die Wahrscheinlichkeitsverteilung vollständig bekannt sein muss. Wir bemerken an dieser Stelle, dass in unseren Messungen diese Bedingung stets erfüllt ist und somit die Verwendung der Shannon-Entropie gerechtfertigt ist.

In [LRSV12] wurde festgestellt, dass andere bekannte Entropieschätzer, deren Methodik auf anderen theoretischen Grundlagen basiert, im Fall des LRNG aus diversen Gründen gar nicht in Frage kommen.

Erst kürzlich wurde von Benjamin Pousse in [P12] ein neuer Ansatz zur theoretischen Erklärung des LRNG-Entropieschätzers veröffentlicht. Pousse verwendet dafür die sogenannte Kolmogorov-Komplexität, die durch die kürzest mögliche Beschreibung einer Nachricht definiert ist. Mit anderen Worten ist die Kolmogorov-Komplexität einer Bitfolge die Bitlänge der optimalen Komprimierung der betrachteten Bitfolge.

Diese Definition lässt bereits erahnen, dass die Kolmogorov-Komplexität nicht berechenbar ist. Pousse zeigt jedoch in [P12], dass mittels Polynominterpolation eine Näherung an die Kolmogorov-Komplexität bestimmt werden kann und dass dieser Ansatz im Entropieschätzer des LRNG implizit implementiert ist.

2.5.3.2 RAID- und Plattencaches

In Serversystemen und zunehmend auch in Desktop-Systemen finden sich Festplatten und Plattencontroller mit Caches. Diese Caches können verschiedene Cache-Strategien implementieren. Die üblichen Strategien sind:

- **Read-Ahead:** Wenn das Betriebssystem eine Leseoperation startet, werden die Daten in der Nähe der gelesenen Stelle auch gelesen und im Cache gespeichert. Falls das Betriebssystem eine weitere Leseoperation startet, um die Nachbardaten zu dem vorherigen Datenblock zu lesen, liefert der Cache sofort die Daten. D.h. es wird nicht mehr die Platte selbst bewegt.
- **Write-Back:** Wenn das Betriebssystem Daten auf die Platte schreibt, werden diese Daten vom Cache zuerst in den Cache geschrieben. Sofort im Anschluss an die Speicherung im Cache meldet der Controller das erfolgreiche Speichern der Daten auf Platte. Erst später werden die Daten aber tatsächlich auf die Platte geschrieben.

Wenn der LRNG als Teil des Gastbetriebssystems in einer Virtualisierungslösung arbeitet, spielt der Puffer-Cache des Hostsystems die gleiche Rolle wie die genannten Platten- oder Controller-Caches.

In beiden Fällen sind die Lese/Schreiboperationen um viele Größenordnungen schneller, da die langsame Festplatte nicht bewegt werden muss. Festplattenzugriffe werden in Millisekunden gemessen, RAM und Cache-Zugriffe in Nanosekunden.

Wenn nun der Cache die Lese/Schreiboperationen stark beschleunigt, muss man sich nun fragen, in wieweit der LRNG noch nutzbar ist, da die Entropie der Operationen um ein vielfaches kleiner ist als mit Platteninteraktion. Die Qualität des LRNG liegt immer bei der

Entropieschätzung. Diese basiert, wie oben beschrieben auf der ersten, zweiten und dritten Ableitung des Zeitstempels. Dabei wird der Zeitstempel in Jiffies gemessen. Und genau die Nutzung dieses groben Zeitstempels hilft hier: die Operationen mit den Caches sind auf jeden Fall schneller als der Jiffies Zeitstempel. Demzufolge kann immer davon ausgegangen werden, dass Plattenzugriffe, welche von Caches gepuffert werden, mit null Bit Entropie abgeschätzt werden.

D.h. ein System, welches Caches zwischen dem LRNG und den Platten hat, wird eine erheblich geringere Erhöhung des Entropieschätzers verzeichnet.

Diese Aussage kann man auch auf Solid State Disks (SSDs) übertragen, sofern man die Geschwindigkeit der Lese/Schreiboperationen auf SSDs mit betrachtet. Wenn diese Operationen so schnell sind, dass sie weit unterhalb der Jiffies-Grenze liegen, gilt die Erklärung zu Caches ebenfalls.

2.6 Extraktion von Zufallszahlen

Der LRNG stellt verschiedene Schnittstellen zur Verfügung, um Daten aus dem LRNG abzuleiten. Alle diese Schnittstellen verwenden den zentralen, im Folgenden beschriebenen Mechanismus.

Dieser Mechanismus ist in zwei Funktionen separat implementiert: `extract_entropy` und `extract_entropy_user`. Er ist identisch in beiden Funktionen implementiert. Die Unterschiede beider Funktionen liegen in Zusatzprüfungen, welche nicht relevant für die Extraktion der Zufallszahlen sind⁶.

Der vorgestellte Mechanismus funktioniert für das Extrahieren von Zufallszahlen aus dem `blocking_pool` und dem `nonblocking_pool`. Die genannten Funktionen müssen mit einer Referenz auf den gewünschten Entropie-Pool aufgerufen werden. Der Quellcode, welcher für die folgenden Schritte zuständig ist und in den genannten Funktionen aufzufinden ist, lautet:

```
static ssize_t extract_entropy(struct entropy_store *r, void *buf,
                             size_t nbytes, int min, int reserved)
{
...
    xfer_secondary_pool(r, nbytes);
    nbytes = account(r, nbytes, min, reserved);
...
    while (nbytes) {
        extract_buf(r, tmp);
...
        i = min_t(int, nbytes, EXTRACT_SIZE);
        memcpy(buf, tmp, i);
        nbytes -= i;
        buf += i;
...
    }
...
}

Quellcode 29: drivers/char/random.c
```

Die Extraktion von Zufallszahlen erfolgt nach folgendem Muster:

- Es wird zunächst eine Anzahl von Bytes aus dem primären Entropie-Pool gelesen und dem aktuellen Entropie-Pool mittels der Funktion `xfer_secondary_pool` hinzugefügt. Diese Aktion wird nur durchgeführt, wenn alle folgenden Bedingungen erfüllt sind:
 - Es muss einen primären Entropie-Pool geben (dies ist der `input_pool` für `blocking_pool` und `nonblocking_pool`; für den `input_pool` gibt es keinen primären Entropie-Pool).

⁶ `extract_entropy` führt einen Continuous-Self-Test nach FIPS 140-2 durch. `extract_entropy_user` prüft, ob der aufrufende Prozess schlafen gelegt werden kann.

- die Entropieschätzung des aktuellen Entropie-Pools zeigt, dass weniger Entropie⁷ enthalten ist als die vom Aufrufer angeforderten Bytes.
- die Entropieschätzung ist kleiner als sein Maximalwert von 1024. Es ist sinnlos, neue Entropie aus dem primären Entropie-Pool zu extrahieren, wenn der aktuelle Pool bereits mit Entropie gesättigt ist.

Die Entnahme von Zufallszahlen aus dem primären Entropie-Pool `input_pool` erfolgt über den gleichen Extraktionsmechanismus als Rekursion, der in diesem Kapitel diskutiert wird. Natürlich ist die Bedingung 1.i) für den primären Entropie-Pool nicht erfüllt, weshalb der komplette Schritt 1 für den `input_pool` wegfällt.

Die minimale Anzahl von Bytes, welche aus dem primären Pool geladen wird, entspricht dem Wert, der in `/proc/sys/kernel/random/read_wakeup_threshold` spezifiziert ist. Standardmäßig sind dies 64 Bits.

Der Übertrag der Bytes aus dem primären Entropie-Pool erfolgt mit diesen Schritten:

- Entnahme der berechneten Bytes aus dem primären Entropie-Pool über den in diesem Kapitel diskutierten Extraktionsmechanismus.
- Hinzufügen der extrahierten Bytes zum aktuellen Entropie-Pool mit der in Abschnitt 2.5.2 beschriebenen Vorgehensweise.
- Die Anzahl der extrahierten Bits wird der Entropieschätzung des aktuellen Entropie-Pools entsprechend der Codesequenz 27 „gutgeschrieben“.
- Nun wird die Entropieschätzung des aktuellen Entropie-Pools mit der Anzahl der extrahierten Bits „belastet“. Dabei wird im Falle von `blocking_pool` sichergestellt, dass nie mehr Bits extrahiert werden, als die Entropieschätzung an Entropie enthält. Stehen weniger Bits zur Verfügung, wird die Anzahl der zu extrahierenden Bytes entsprechend nach unten korrigiert. Im Falle des `nonblocking_pools` wird die Anzahl der zu extrahierenden Bytes nicht verändert.

Diese Anzahl der zu extrahierenden Bytes wird nun von der Entropieschätzung abgezogen. Die Entropieschätzung kann nicht negativ werden, d.h. das Minimum der Entropieschätzung ist Null.

- Nun wird die angeforderte und gegebenenfalls in Schritt 2 angepasste Anzahl von Bytes aus dem aktuellen Entropie-Pool mittels der Funktion `extract_buf` erzeugt. Diese Funktion führt die folgenden Schritte durch:
 - Initialisiert SHA-1
 - Generierung eines SHA-1-Hashwertes aller Rohdaten des aktuellen Entropie-Pools – der Zwischenspeicher für die SHA-1-Operation wird noch nicht gelöscht.
 - Der generierte SHA-1-Wert wird zu dem aktuellen Entropie-Pool entsprechend der Diskussion in Abschnitt 2.5.2 hinzugefügt. In diesem Schritt werden die nächsten 16 Werte aus den Rohdaten des Entropie-Pools ausgelesen.

```
static void _mix_pool_bytes(struct entropy_store *r, const void *in,
                          int nbytes, __u8 out[64])
{
    ...
    if (out)
        for (j = 0; j < 16; j++)
            ((__u32 *)out)[j] = r->pool[(i - j) & wordmask];
    ...
}

```

Quellcode 30: `drivers/char/random.c`

⁷ Diese Entropie wird in Bits gemessen, wie in Abschnitt 2.5.3.1 erläutert.

- Die in Schritt 3.iii) ausgelesenen 64 Bytes – der Rückgabewert von `_mix_pool_bytes` ist die Variable `out`, welche als ein Array bestehend aus 64 8 Bit `unsigned int` definiert ist – werden wiederum in die bestehende SHA-1-Instanz eingefügt und es wird ein weiterer SHA-1-Wert berechnet – dieser SHA-1-Wert wird damit mit dem initialen Zustand der Rohdaten und den neuen 64 Bytes berechnet.
- Dieser SHA-1-Wert wird nun halbiert, indem die linken 80 Bit des SHA-1-Wertes mit den rechten 80 Bit mit XOR verknüpft werden. Weiterhin werden diese 80 Bit mit einer Zufallszahl aus dem Hardware-Zufallszahlengenerator mittels XOR verknüpft. Dabei werden nur die ersten 80 Bit der Hardware-Zufallszahl mit den 80 Bit des gefalteten SHA-1 Wertes verbunden:

```
static void extract_buf(struct entropy_store *r, __u8 *out)
{
    ...
    /*
     * In case the hash function has some recognizable output
     * pattern, we fold it in half. Thus, we always feed back
     * twice as much data as we output.
     */
    hash[0] ^= hash[3];
    hash[1] ^= hash[4];
    hash[2] ^= rol32(hash[2], 16);

    /*
     * If we have a architectural hardware random number
     * generator, mix that in, too.
     */
    for (i = 0; i < LONGS(EXTRACT_SIZE); i++) {
        unsigned long v;
        if (!arch_get_random_long(&v))
            break;
        hash.l[i] ^= v;
    }
    ...
}

```

Quellcode 31: drivers/char/random.c

- Die nun vorhandenen 80 Bits werden als Zufallszahl für eine Runde definiert. Die gegebenenfalls vorhandenen Hardware-Zufallszahlenbits, welche über die 80 Bit-Grenze hinausgehen, werden ignoriert.
- Die generierten 80 Bits in Schritt 3 werden mit der Anzahl der vom Aufrufer angeforderten Bytes verglichen:
 - Wenn die Anzahl der Bytes aus Schritt 2 mehr als 10 Bytes beträgt, wird Schritt 3 noch einmal ausgeführt und die neu erzeugten Daten mit denen der vorherigen Runde konkateniert. Dies wird so lange durchgeführt, bis so viele Bytes aus dem Entropie-Pool extrahiert wurden, wie angefordert.
 - Wenn weniger als die generierten 10 Bytes angefordert worden (oder in der letzten Runde in Schritt 4.i müssen weniger als 10 Bytes erzeugt werden, da eine Anzahl von Bytes zu generieren ist, welche nicht durch 10 teilbar ist), werden die überzähligen Bytes der generierten 10 Bytes abgeschnitten.

Der nun generierte Wert aus den konkatenierten Teilwerten der SHA-1-Operation ist die Zufallszahl.

2.6.1 Extraktion von Daten via Gerätedateien

Der in Abschnitt 2.6 diskutierte Extraktionsmechanismus wird nun von den Lesefunktionen der Gerätedateien `/dev/random` und `/dev/urandom` wie folgt verwendet:

- `/dev/random`: Wenn ein Prozess lesend auf die Gerätedatei zugreift, wird die Funktion `random_read` aufgerufen. Diese Funktion ruft den Mechanismus aus Abschnitt 2.6 mit dem `blocking_pool` auf. Wenn dieser Mechanismus nun weniger Bytes als angefordert (aufgrund der Diskussion in Abschnitt 2.6 - Schritt 2) zurückliefert, wird der aufrufende Prozess schlafen gelegt:

```
static ssize_t
random_read(struct file *file, char __user *buf, size_t nbytes, loff_t *ppos)
{
...
    while (nbytes > 0) {
...
        n = extract_entropy_user(&blocking_pool, buf, n);
...
        if (n == 0) {
...
            wait_event_interruptible(random_read_wait,
                input_pool.entropy_count >=
                    random_read_wakeup_thresh);
...
            nbytes -= n;
        }
    }
}

```

Quellcode 32: `drivers/char/random.c`

Der Prozess wird so lange schlafen, bis die Entropieschätzung des `input_pools` einen Entropiewert von mehr als den in `/proc/sys/kernel/random/read_wakeup_threshold` angegebenen Wert aufweist – standardmäßig ist dieser Wert auf 64 Bits gesetzt. Wenn die Entropieschätzung einen höheren Wert aufweist, wird der Prozess aufgeweckt, und es werden weiter Daten aus dem `blocking_pool` extrahiert. Wenn wieder unzureichend Entropie vorhanden ist, wird der Prozess wieder schlafen gelegt. Ansonsten wird die Leseoperation zu Ende geführt.

- `/dev/urandom`: Eine Leseoperation auf `/dev/urandom` wird von der Funktion `urandom_read` behandelt. Diese Funktion ruft den Mechanismus aus Abschnitt 2.6 mit dem `nonblocking_pool` auf. Dieser Mechanismus kann niemals weniger Bytes als angefordert zurückliefern – siehe Diskussion in Abschnitt 2.6, Schritt 2.

2.6.2 Extraktion von Daten innerhalb des Kerns

Für kerninterne Zwecke stellt der LRNG folgende Funktionen bereit:

- `get_random_bytes` stellt dem Kern eine Zufallszahlenquelle zur Verfügung. Diese Funktion hat die folgenden Mechanismen zur Zufallszahlengenerierung zur Auswahl, wobei der erste Mechanismus Priorität hat, falls er vorhanden ist:
 - Mit dem Erscheinen von Hardware-Zufallszahlengeneratoren in allgemein erhältlichen Systemen⁸ nutzt die Funktion `get_random_bytes` diese Hardware-Zufallszahlengeneratoren. Die von `get_random_bytes` geforderte Anzahl an Bytes wird von dem Hardware-RNG angefordert und direkt an den Aufrufer durchgereicht. Genaue Informationen zu der Unterstützung von Hardware-RNGs werden in Abschnitt 2.7 bereitgestellt.
 - Falls der Hardware-RNG nicht verfügbar ist, ruft diese Funktion den Mechanismus aus Abschnitt 2.6 mit dem `nonblocking_pool` auf. Damit werden immer genauso viele Bytes generiert, wie angefordert – siehe Diskussion in Abschnitt 2.6, Schritt 2.

⁸ Die erste Unterstützung für Hardware-Zufallszahlengeneratoren im Rahmen des LRNG erfolgte für die Intel-x86 RDRAND-Instruktion, welche ab der IvyBridge Prozessorgeneration ausgeliefert wird.

Demzufolge wird dem Kern eine dem /dev/urandom vergleichbare Quelle bereitgestellt.

- `get_random_int` ist eine weitere kerninterne Schnittstelle zum LRNG. Die darüber generierten Daten werden **nicht** als kryptographisch stark interpretiert. Die Funktion generiert einen MD5-Hash der aktuellen Systemzeit. Falls ein Hardware-RNG zur Verfügung steht, wird dieser verwendet.
- `randomize_range` ist ebenfalls eine kerninterne Schnittstelle zum LRNG, welche auf der vorher beschriebenen Funktion `get_random_int` basiert.
- `next_pseudo_random32` ist eine weitere kerninterne Schnittstelle zum LRNG. Die darüber generierten Daten werden **nicht** als kryptographisch stark interpretiert, da keine Verbindung zu den Entropiepools besteht. Die Funktion generiert einen Integer-Wert bestehend aus einem Seed verändert um statische Werte.

2.7 Hardware-Zufallszahlengeneratoren

Der LRNG hat folgende Funktionen in seinem Quellcode, die an vorher beschriebenen Orten platziert sind:

```
#ifdef CONFIG_ARCH_RANDOM
# include <asm/archrandom.h>
#else
static inline int arch_get_random_long(unsigned long *v)
{
    return 0;
}
static inline int arch_get_random_int(unsigned int *v)
{
    return 0;
}
#endif
Quellcode 33: include/linux/random.h
```

Diese Funktionen haben die folgende Bedeutung:

- `arch_get_random_long` gibt eine Zufallszahl vom Hardware-RNG als long-Wert zurück.
- `arch_get_random_int` gibt eine Zufallszahl vom Hardware-RNG als int-Wert zurück.

Standardmäßig liefern beide Funktionen null zurück, welches vom LRNG als ein Nichtvorhandensein eines Hardware-RNGs interpretiert wird.

Falls zur Kern-Übersetzungszeit aber `asm/archrandom.h` genutzt wird, stehen hier Ersatzfunktionen, welche den Hardware-RNG befragen.

Die folgenden Abschnitte diskutieren die bereits unterstützten Hardware-RNGs.

2.7.1 Intel RDRAND-Instruktion

Beginnend mit der Intel IvyBridge x86-Prozessorgeneration implementiert Intel die Instruktion RDRAND. Diese Instruktion bietet einen Zugang zu einem Hardware-RNG, der auf einer Hardware-Rauschquelle basiert und dessen Werte durch einen deterministischen RNG verarbeitet. Dieser deterministische RNG ist ein SP800-90A kompatibler DRBG mit AES im CTR-Operationsmodus als Kern⁹.

Die Verbindung zwischen der RDRAND-Instruktion und dem LRNG wird über die vorher genannten Funktionen bereitgestellt. Die Implementierung dieser Funktionen ist in `arch/x86/include/asm/archrandom.h` zu finden. Der dort enthaltene Assemblercode stellt

9 Genaue Informationen zu den Interna der RDRAND-Instruktion sind Buch [Intel Digital Random Number Generator](#) zu finden.

sicher, dass die Funktion nur Werte zurückliefert, wenn das CPUID-Register einen Marker für das Vorhandensein der RDRAND-Instruktion enthält.

3 Bekannte Analysen des Linux-RNG

2006 haben Gutterman et al. in [GPR06] eine Analyse des LRNG, wie er in Kernversion 2.6.10 verwendet wird, veröffentlicht. Die Arbeit [LRSV12] von Lacharme et al. aus dem Jahr 2012 gilt für die Kernversion 2.6.30.7 (und höhere Versionen) und zeigt, dass einige der in [GPR06] beschriebenen Angriffe in neueren Kernversionen nicht mehr möglich sind. An dieser Stelle sollte man sich jedoch darüber im Klaren sein, dass einige Angriffe Zugriff auf bestimmte Systemressourcen erfordern, die nur durch eine vollständige Kontrolle über das System selbst sichergestellt werden können. Kann ein Angreifer beispielsweise den `blocking_pool` oder den `nonblocking_pool`, die sich beide im Betriebssystemkern befinden, auslesen, so hat er entsprechend der Von-Neumann-Architektur auch auf jeden anderen Speicherbereich Zugriff. In diesem Fall gibt es für den Angreifer offensichtlich Möglichkeiten, sein gewünschtes Ziel weitaus einfacher zu erreichen.

3.1 Angriffe von Gutterman et al. und deren aktuelle Relevanz

Wir geben jetzt einen kurzen Überblick über die einzelnen Angriffe von Gutterman et al., sowie eventuell bereits getroffene Gegenmaßnahmen (d.h. spätestens ab Kernversion 2.6.30.7) wie sie in [LRSV12] beschrieben werden.

3.1.1 Denial-of-Service-Angriffe

Zwei unterschiedliche Denial-of-Service-Angriffe werden in [GPR06] vorgestellt. Der erste besteht einfach in durchgehendem Anfordern von Zufallszahlen aus dem `blocking_pool`. Als Lösungsvorschlag wird die (unpraktikable) Einführung von Quotas vorgeschlagen. Durch andauerndes Lesen aus `/dev/urandom`, also aus dem `nonblocking_pool`, war es ihnen zudem möglich, die Entropie des `input_pool` schneller zu verringern, als sie durch den Entropiesammler erhöht wurde. Die Folge war auch hier ein DOS-Angriff auf den `input_pool` und damit auch auf den `blocking_pool`. Dieser Angriff ist, wie Lacharme et al. bemerken, mindestens ab Kernversion 2.6.30.7 nicht mehr möglich, da bei Anfragen auf `/dev/urandom` stets mindestens 8 Bytes Entropie im `input_pool` verbleiben müssen.

3.1.2 Verwendung in Diskless-Systemen

Wie in Kapitel 2 beschrieben, wird beim Herunterfahren des Systems der aktuelle Inhalt der Entropie-Pools gespeichert und bei erneutem Systemstart wieder in die Entropie-Pools geschrieben. Dieser Sicherheitsmechanismus, der Angriffe aufgrund der relativ einfach vorhersagbaren Abläufe beim Systemstart und der daraus resultierenden geringen Entropie verhindern soll, ist bei Live-Systemen wie Knoppix und anderen „Diskless-Systemen“ wie OpenWRT nicht gegeben. Ts'o sieht hierin jedoch keinen Fehler des LRNG, sondern der Anwendung.

3.1.3 (Enhanced-)Backward-Secrecy¹⁰

Im betrachteten Fall kennt ein Angreifer den aktuellen Inhalt eines Entropie-Pools (oder aller Entropie-Pools) und möchte den vorhergehenden Zustand bestimmen. Bei der in [GPR06] analysierten Kernversion wurden im Fall von `blocking_pool` und `nonblocking_pool` nur 3 Wörter à 32 Bit verändert, sodass hier 2^{96} Möglichkeiten zu testen wären. Für mehr als die Hälfte der Fälle wurde zudem eine Verbesserung des Angriffs, die lediglich 2^{64} Tests benötigt, vorgestellt. Für den `input_pool` benötigt der generische Angriff wegen der Veränderung von 9 Wörtern à 32 Bit 2^{288} Tests. Nach [LRSV12] wird dieser Angriff ab Kernversion 2.6.26 durch Änderung des Aufrufs der Feedback-Funktion unterbunden, sodass die beste generische Methode, den vorherigen Zustand zu rekonstruieren, eine Komplexität von $O(2^{160})$ hat (siehe dazu auch Abschnitt 4.3).

¹⁰ In der genannten Arbeit wird statt „Enhanced-Backward-Secrecy“ der Begriff „Forward-Security“ genutzt.

3.2 Analyse von Lacharme et al.

Wir fassen weitere Ergebnisse aus [LRSV12] zusammen:

3.2.1 LRNG ohne Eingabe in die Entropie-Pools

Betrachtet man den LRNG unter der Voraussetzung, dass keine neue Eingaben in die Entropie-Pools durchgeführt werden, so verhält sich der LRNG wie ein gewöhnlicher deterministischer RNG, etwas präziser: wie die Kopplung von linearen Schieberegistern (LFSR). Die Ausgabe wird dementsprechend nach einer gewissen Anzahl von Aufrufen zyklisch. Aus der Wahl nicht irreduzibler Polynome zur Definition der LFSR resultiert hier (unnötigerweise) eine Periodenlänge, die kleiner der maximal möglichen Periodenlänge ist; dies wird jedoch nicht als sicherheitskritisch angesehen.

3.2.2 (Enhanced-)Forward-Secrecy¹¹

Hier werden zwei Fälle betrachtet: Im ersten Fall kennt ein Angreifer den Zustand von einem der beiden Output-Pools, nicht aber den Inhalt von `input_pool` und möchte den nachfolgenden Zustand des entsprechenden Output-Pools bestimmen. Da nach Anforderung von Entropie aus dem `input_pool` mindestens 64 Bit übertragen werden, benötigt ein Angriff im Durchschnitt 2^{63} Tests.

Im zweiten Fall sind dem Angreifer sowohl der entsprechende Output-Pool, als auch `input_pool` bekannt und er möchte wieder den nächsten Zustand in Erfahrung bringen. Nimmt man an, dass bis zur nächsten Ausgabe k Bits Entropie gesammelt werden, so können die folgende Fälle eintreten. Ist k mindestens 64, so muss der Angreifer aus 2^k Möglichkeiten wählen. Ist $k < 64$ und der Entropiezähler hoch genug, so reichen im Durchschnitt 2^{k-1} Tests aus. Auf diese Weise wird jedoch die Entropie reduziert, sodass ab einem bestimmten Punkt die vorhandene Entropie zu gering sein wird und dann so lange Entropie gesammelt wird, bis k mindestens 64 Bit groß ist. Ab diesem Punkt führt dies ebenfalls zu einer Sicherheit von 64 Bit.

3.2.3 Eingabebasierte Angriffe

Eine wichtige Erkenntnis, die von Lacharme et al. mathematisch bewiesen wird, ist, dass das Einmischen neuer Daten in die Entropie-Pools niemals zu einer Verringerung der enthaltenen Entropie führt. Aus diesem Grund sind sogenannte eingabebasierte Angriffe nicht möglich. D.h. ein Angreifer kann ohne Kenntnis der Entropie-Pools keinen Nutzen aus der Kontrolle über die Entropie-Eingabe ziehen¹².

3.2.4 Bewertung der Entropieabschätzung

Die Entropieabschätzung muss gewährleisten, dass tatsächlich genug Entropie vorhanden ist; dementsprechend sollte die Abschätzung eher konservativ gehalten werden. Die Messungen in [LRSV12] bestätigen dies für den LRNG.

3.3 Schlussfolgerungen aus [GPR06] und [LRSV12]

Am Ende der Arbeiten [GPR06] und [LRSV12] werden einige Empfehlungen zur Weiterentwicklung des LRNG gegeben. Diese beziehen sich größtenteils auf direkte Gegenmaßnahmen zu den oben beschriebenen Schwachstellen. Zudem bemängeln Gutterman et al. die unnötige Komplexität des LRNG-Designs (insbesondere die ihrer Meinung nach zu großen Poolgrößen und das häufige Aufrufen von SHA-1) und schlagen stattdessen die Verwendung der Barak-Halevi-Konstruktion vor.

Wie bereits in den einzelnen Abschnitten erwähnt, wurden die in [GPR06] beschriebenen Mängel in neueren Kernversionen größtenteils behoben.

Lacharme et al. bescheinigen dem LRNG ein gutes Sicherheitsniveau. Neben dem Hinweis auf die einfache Lösung des in 3.2.1 angesprochenen Problems, weisen sie auch darauf hin, dass

¹¹ In der genannten Arbeit „Backward Security“ genannt.

¹² Die Funktionsweise des Entropieschätzers wird in Abschnitt 2.5.3 ausführlich beschrieben.

die Verwendung einer aktuellen Hashfunktion wie SHA-3 statt SHA-1 ein vollständiges Redesign erfordern würde.

Das von Lacharme et al. bedauerte Fehlen einer theoretischen Grundlage für die Entropieschätzer wurde, wie bereits in 2.5.3.1 erwähnt, von Pousse (siehe [P12]) behoben.

4 Abdeckung der Anforderungen an NTG.1

Die RNG Funktionalitätsklasse NTG.1 ist in [KS11], Abschnitt 4.10 definiert. Der vorliegende Abschnitt listet die Anforderungen der Funktionalitätsklasse auf und vergleicht diese Anforderungen mit der tatsächlichen Implementierung.

Die folgenden Abschnitte spiegeln die Anforderungen aus Abschnitt 4.10 von [KS11] wider.

Die Analyse zeigt folgendes: `/dev/random` **erfüllt** die Anforderungen von NTG.1 auf x86-Systemen. Aufgrund der Anforderung von NTG.1.6, die Entropie empirisch zu messen und dem Umstand, dass die Messungen ausschließlich auf einem x86-System durchgeführt wurden, gepaart mit dem Fakt, dass die Entropieschätzung teilweise auf der hardwareabhängigen Messung der Zeitgeber basiert, können die empirisch ermittelten Werte nicht einfach auf andere Systeme übertragen werden. Demzufolge liefert dieses Dokument keine Aussage zu NTG.1 für andere Hardwareplattformen.

4.1 NTG.1.1

Die Anforderung NTG.1.1 ist definiert als:

„The RNG shall test the external input data provided by a non-physical entropy source in order to estimate the entropy and to detect non-tolerable statistical defects under the condition [assignment: requirements for NPTRNG operation].“

Unabhängig von `/dev/random` oder `/dev/urandom` wird diese funktionelle Anforderung bei der Verarbeitung von Ereigniswerten mit dem `input_pool` wie folgt implementiert – folgende Aussagen müssen demnach in dem Assignment der Anforderung auftauchen:

- Für jede Klasse von Rauschquellen implementieren die Entropie-Sammelfunktionen Prüfungen, ob der Ereigniswert „sinnvoll“ ist. Konkret werden folgende Prüfungen implementiert:
 - Eingabegeräte: Die Funktion `add_input_randomness` implementiert den in Quellcode 7 dargestellten Code. Dieser Code verwirft jeden eingehenden Ereigniswert, wenn er gleich dem vorhergehenden Ereigniswert ist.
 - Blockgeräte: Die Funktion `add_disk_randomness` ist implementiert entsprechend der Diskussion im Quellcode 13. Dieser Code verwirft ein Ereignis, wenn keine entsprechende Blockgerätedatenstruktur vorhanden ist, oder das Blockgerät nicht als geeignet für Zufallszahlen angesehen ist.
 - Interrupts: Es werden die Informationen als Entropiequelle verwendet, welche für das Abarbeiten des Interrupts vom System notwendig sind. Wenn diese Daten nicht korrekt sind, kann der Interrupt vom System nicht verarbeitet werden und das Gerät wird nicht funktionieren. In einem solchen Fall wird der Linux Kern abstürzen.

Mit den diskutierten Vorkehrungen werden statistische Auffälligkeiten vermieden.

- Die Heuristik in `add_timer_randomness`, welche die Entropie abschätzt, ist anhand der Kolmogorow-Entropieabschätzung modelliert. Diese Abschätzung impliziert, dass ein Ereignis maximal 11 Bit an Entropie aufweisen kann. Die quantitativen Tests in Abschnitt 6.2.7 zeigen, dass die Abschätzung relativ konservativ ist.

Mit diesen Resultaten kann gesagt werden, dass die Verarbeitung der Eingabewerte der verschiedenen Rauschquellen den Anforderungen von NTG.1.1 entspricht.

4.2 NTG.1.2

Die Anforderung NTG.1.2 ist definiert als:

„The internal state of the RNG shall have at least [assignment: Min-entropy]. The RNG shall prevent any output of random numbers until the conditions for seeding are fulfilled.“

Die Analyse in Abschnitt 4.9 und die darin enthaltenen Aussagen gelten für diese Anforderung ebenso, wobei der Wert für „Min-entropy“ natürlich nicht willkürlich hoch angesetzt werden darf.

Auf Basis des Designs und der statistischen Analyse können folgende Assignments für Min-entropy getroffen werden: „... as many bits of theoretical minimal entropy as requested by the caller“ – /dev/random blockiert den Lesevorgang bis Entropie für eine RNG Runde, d.h. 80 Bit, vorhanden sind. Diese 80 Bit werden an den Aufrufer übergeben. Wenn der Aufrufer mehr Zufallszahlen anfordert, erhält der Aufrufer immer 80 Bit-Blöcke, falls entsprechend Entropie vorhanden ist, bis seine Anforderung erfüllt ist.

Die Tabelle in NTG.1.6 zeigt die kumulierte heuristische Entropie pro Ereigniswert. Dieser Wert ist die gewichtete, durchschnittliche Entropie, die der LRNG einem Hardware-Ereignis zuweist. Auf diesen heuristischen Entropiewerten basiert das Blockieren des Lesens bei /dev/random.

Die Graphen in den Tests in Abschnitt 6.2.7 enthalten die minimale Entropie für die Ereigniswerte, welche mit den heuristischen Entropiewerten von 0, 1 und 8 bewertet werden.

$$0,50(\text{minimale Entropie Ereignis mit heuristischer Entropie von } 0) * 0,5144(\text{Häufigkeit}) + \\ 7,32(\text{minimale Entropie Ereignis mit heuristischer Entropie von } 1) * 0,431(\text{Häufigkeit}) + \\ 1,11(\text{minimale Entropie Ereignis mit heuristischer Entropie von } 8) * 0,001(\text{Häufigkeit}) = \\ 3,413$$

Wenn diese minimalen Entropiewerte nun mit den in der Tabelle von NTG.1.6 genannten Häufigkeiten des Auftretens der Ereignisse mit heuristischer Entropie von 0, 1, und 8 gewichtet werden, kommen wir auf eine durchschnittliche minimale Entropie von: 3,413.

Damit ist die theoretisch berechnete minimale Entropie pro Hardware-Ereignis höher als die heuristische Entropie, welche in NTG.1.6 genannt ist. Da die Berechnung der theoretischen minimalen Entropie nicht für alle Ereignisse berechnet wurde, kann der errechnete Wert als untere Abschätzung angesehen werden.

Wie oben beschrieben stellt der blockierende Mechanismus von /dev/random sicher, dass mindestens genauso viel heuristische Entropie in dem `blocking_pool` ist, wie Zufallsbits angefragt werden. Mit der gezeigten Berechnung der theoretischen minimalen Entropie ist damit auch sichergestellt, dass pro Bit an extrahierter Zufallszahl auch ein Bit an minimaler Entropie enthalten ist.

4.3 NTG.1.3

In diesem Abschnitt betrachten wir die Anforderung NTG.1.3:

„The RNG provides backward secrecy even if the current internal state and the previously used data for reseeding, resp. for seed-update, are known.“

Da der Extraktionsmechanismus für alle Entropie-Pools identisch ist, gilt die nachfolgende Argumentation für alle drei Entropie-Pools `input_pool`, `blocking_pool` und `nonblocking_pool`. Alle Aussagen im weiteren Verlauf über „den Entropie-Pool“ gelten deshalb für jeden dieser drei Pools¹³.

Wir setzen jetzt voraus, dass ein Angreifer die in NTG.1.3 spezifizierten Informationen hat, also

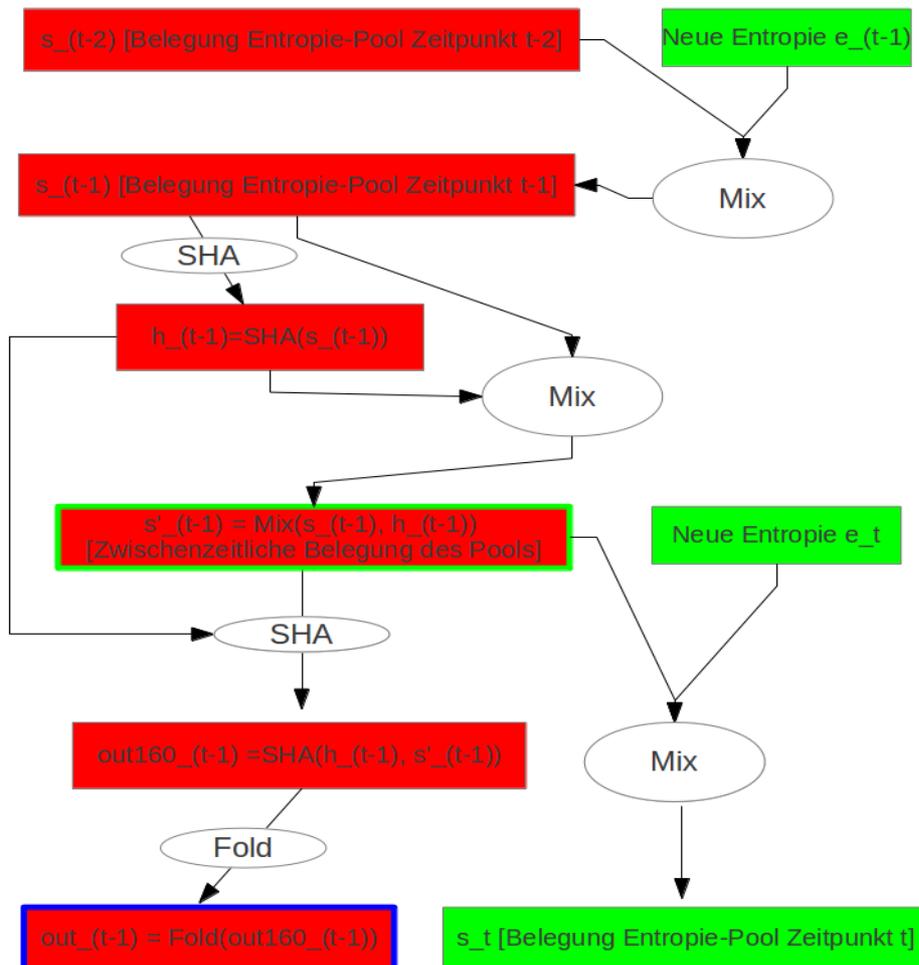
- den Zustand des Entropie-Pools `s_t` zum Zeitpunkt `t`, sowie
- alle Daten für Reseeding und Seed-Update für alle Zeitpunkte `0,1,...,t`.

Daraus möchte ein Angreifer die vorangegangene Ausgabe

- `out_(t-1)`

¹³ User können nicht direkt auf `input_pool` zugreifen, zudem ist `input_pool` nicht als reiner DRNG zu sehen.

bestimmen. Die folgende Abbildung veranschaulicht die Situation¹⁴¹⁵,



wobei grüner Hintergrund dem Angreifer bekannte Daten, roter Hintergrund dem Angreifer unbekannte Daten kennzeichnet. Der zusätzliche blaue Rahmen um out_{t-1} soll darauf hinweisen, dass der Angreifer diesen Wert (out_{t-1}) bestimmen möchte.

Wie aus der Abbildung ersichtlich wird, muss ein Angreifer für weitere Berechnungen zunächst s'_{t-1} aus s_t und e_t rekonstruieren. Wir nehmen hier an, dass der Angreifer hierzu in der Lage ist, da im Worst-Case-Szenario nach der Ausgabe der letzten Zufallszahl out_{t-1} gar keine neue Entropie eingemischt wird und dann $s_t = s'_{t-1}$ gelten würde, d.h. s'_{t-1} wäre dem Angreifer ohne weitere Rechnung bekannt. Deshalb wurde s'_{t-1} grün eingerahmt.

Für die Bestimmung von out_{t-1} fehlt dem Angreifer noch h_{t-1} . Dieses müsste er aus $s'_{t-1} = \text{Mix}(s_{t-1}, h_{t-1})$ berechnen, was praktisch unmöglich ist. Der Grund hierfür liegt in den Eigenschaften der Mischfunktion Mix und der Hashfunktion SHA-1:

- Beide Funktionen sind auf dem betrachteten Definitionsbereich nicht invertierbar.
- Für beide Funktionen ist es schwierig, überhaupt ein Urbild zu einem bestimmten Ausgabewert zu finden¹⁶. Die Bestimmung der gesamten Urbildmenge ist deshalb und besonders aufgrund der Mächtigkeit dieser Menge von entweder 1024 Bit oder 4096

14 Für eine ausführliche Darstellung der Funktionsweise des LRNG verweisen wir auf Kapitel 2.

15 Die Beschreibung ist eine Vereinfachung, da die Problematik der parallelen Verarbeitung von Ereignissen/Anfragen im LRNG nicht berücksichtigt wird. Das Schaubild erläutert ausschließlich die Abarbeitung einer isolierten Runde des LRNG. Bei der Betrachtung paralleler Anfragen/Ereignissen können sich einige der aufgezeigten Zustände überlappen.

Bit vollkommen jenseits jeder Berechnungsmöglichkeit in einer Zeitspanne die nicht in Äonen angegeben wird.

Somit lässt sich feststellen, dass die Extraktion von Zufallszahlen aus allen drei Entropie-Pools der Anforderung NTG.1.3 entsprechen; **insbesondere erfüllt /dev/random die Anforderung NTG.1.3.**

Bemerkung

Der Betrachter sollte sich das ganze Bild des Betriebssystems vor Augen halten. Alle drei Entropie-Pools werden im Betriebssystem-Kern verwaltet. Wenn ein Angreifer in den Kern eindringen kann und die Pools oder die Mix-Funktion beobachten oder beeinflussen kann, dann wurden **alle** Sicherheitsbarrieren des Betriebssystems überwunden. Ein Angreifer, der dieses Ziel erreicht hat, kann viel mehr Macht ausnutzen:

- Beobachtung jedes Leseaufrufs von /dev/random oder /dev/urandom um jegliche neu generierte Zufallszahlen abzufangen.
- Auslesen aller Speicherbereiche, die ggf. Schlüsselwerte oder Zustände von via /dev/random oder /dev/urandom ge-seedeten DRNGs enthalten, welche auf vorhergehenden Zufallszahlen beruhen.
- Beobachten, verändern, Replay, etc. aller kryptographischen Operationen im System.
- Direkter Zugang zu allen geschützten Bereichen des Betriebssystems und damit einhergehend das Überwinden aller Sicherheitsbarrieren wie Zugriffskontrollen, inklusive denen, welche auf Kryptographie aufsetzen.

Demzufolge muss die geführte Diskussion als rein formelle Diskussion angesehen werden, welche keine praktische Relevanz besitzt. In der Praxis wird ein Angreifer, der theoretisch die Entropie-Pools und deren Verarbeitung beobachten kann, viel gefährlichere Aktionen – auch aus Sicht von kryptographischen Mechanismen – durchführen als die, die in NTG.1.3 angenommen werden.

4.4 NTG.1.4

Die Anforderung NTG.1.4 ist definiert als:

„The RNG generates output for which [assignment: number of strings] strings of bit length 128 are mutually different with probability [assignment: probability].“

Die Entropie-Pools für die Ausgabe, `blocking_pool` und `nonblocking_pool`, haben eine Größe von 1024 Bit. Entsprechend der Analyse in Abschnitt 4.8 kann als Worst-Case-Szenario angesetzt werden, dass nach der Generierung von 1024 Bit Zufallszahlen via /dev/random oder /dev/urandom diese Entropie-Pools neu vom `input_pool` geseedet werden.

Die beiden Assignments in der Anforderung werden mit den folgenden Formeln angegeben. Aus diesen Formeln können reale Werte berechnet werden, die in den Assignments eingesetzt werden.

4.4.1 /dev/random

Vorüberlegung - Ausgabe von Bitstrings der Länge 128

Möchten wir Ausgaben von jeweils 128 Bit, so werden, wie in Abschnitt 2.6 beschrieben, zweimal 80 Bit aus dem LRNG gezogen, konkateniert und die letzten 32 Bit verworfen.

Idealfall - der perfekte Zufallszahlengenerator

Im Idealfall unterliegen die ausgegebenen Bitstrings einer Gleichverteilung. Hier erwartet man nach dem Geburtstagsparadoxon nach 2^{64} Ausgaben von je 128 Bit eine Kollision mit einer Wahrscheinlichkeit von

$$P(\text{Kollision nach } 2^{64} \text{ Ausgaben}) \approx 0.3935.$$

16 Für SHA-1 ist der Aufwand direkt an die Kollisionsresistenz gebunden und somit als extrem hoch einzustufen.

Auf die gleiche Weise, wie obige Wahrscheinlichkeit bestimmt werden kann, leiten wir jetzt eine Formel für die Wahrscheinlichkeit her, dass nach n Ausgaben von je 128 Bit keine Kollision auftritt. Die Anzahl an Möglichkeiten für die Ausgabe von n paarweise verschiedenen Bitstrings der Länge 128 ist durch

$$A = 2^{128} \cdot (2^{128} - 1) \cdot \dots \cdot (2^{128} - n + 1)$$

gegeben, woraus sich für die Wahrscheinlichkeit, dass keine Kollision nach der Ausgabe von n 128-Bit-Wörtern auftritt, als

$$P(n) = \frac{A}{(2^{128})^n}$$

ergibt. Mit Hilfe der (etwas groben) Abschätzung¹⁷

$$A = 2^{128} \cdot (2^{128} - 1) \cdot \dots \cdot (2^{128} - n + 1) > (2^{128} - n + 1)^n$$

erhält man eine leicht berechenbare untere Schranke für P:

$$P(n) > \left(\frac{2^{128} - n + 1}{2^{128}} \right)^n$$

Damit ergibt sich beispielsweise, dass 2^{55} aufeinanderfolgende Ausgabe-Bitstrings der Länge 128 mit einer Wahrscheinlichkeit von $P > 0.999996$ paarweise verschieden sind.

Bemerkung:

- Umformuliert bedeuten obige Werte, dass $k > 2^{55}$ Bitstrings der Länge 128 entnommen werden können und mit einer Wahrscheinlichkeit von $P > 1 - \epsilon$, für $\epsilon = 3.8e-6$, keine Kollision auftritt, also die ausgegebenen Bitstrings paarweise verschieden sind. (Zum Vergleich betrachte man die Anforderung $k > 2^{34}$, $\epsilon < 2^{-16}$ aus Tabelle 13 in [KS11] für AVA_VAN.5 (high).)
- Setzt man in diese Formel $n = 2^{64}$ ein, so erhält man folgende Abschätzung für die Gegenwahrscheinlichkeit zu der obigen Formel aus dem Geburtstagsparadoxon:

$$P(\text{Keine Kollision nach } 2^{64} \text{ Ausgaben}) > \frac{1}{e} \approx 0.3678.$$

Vergleicht man diese Abschätzung mit dem tatsächlichen Wert von etwa 0.6065^{18} , so fällt auf, dass die obige Abschätzung recht ungenau ist. Die tatsächlichen Werte für die Wahrscheinlichkeiten $P(n)$ sind in der Realität also noch deutlich höher. Somit werden selbst deutlich mehr als 2^{55} aufeinanderfolgende Ausgabe-Bitstrings der Länge 128 mit einer Wahrscheinlichkeit von $P > 1 - 2^{-16}$ paarweise verschieden sein.

Interpretation

Die folgenden Argumente, die größtenteils auf der ausführlichen Beschreibung der Funktionsweise des LRNG aus Kapitel 2 basieren, stützen die Vermutung, dass der LRNG dem oben beschriebenen Idealfall sehr nahe kommt:

- Sowohl `blocking_pool` als auch `nonblocking_pool` haben die enorme Größe von 1024 Bit. Für jede feste SHA-1-Ausgabe von 160 Bit ist die Urbildmenge somit extrem groß, sodass hier keine Abweichungen von der Gleichverteilung zu erwarten sind. Zudem wird ein großer Teil der Belegung des jeweiligen Entropie-Pools bei jeder Zufallszahlen-Ausgabe durch das Zurückmischen geändert.

¹⁷ Wir verwenden (statt der üblichen Abschätzung mittels der Stirling-Formel) diese grobe Abschätzung zur einfachen Berechnung einer unteren Schranke für die Wahrscheinlichkeit P. Diese Berechnung verwendet extrem große Zahlen.

¹⁸ Da $1 - 0.3935 = 0.6065$.

- Für den `blocking_pool` gilt zusätzlich, dass nur Zufallszahlen ausgegeben werden, wenn ausreichend Entropie vorhanden ist. Falls nötig wird dementsprechend Entropie aus dem `input_pool` angefordert und eingemischt, was ebenfalls zu einer umfangreichen Veränderung der Belegung des Entropie-Pools führt und folglich eine Gleichverteilung der Ausgabewerte begünstigt. Wir bemerken hier noch, dass das Reseeding spätestens nach 6 Ausgaben von jeweils 128 Bit geschieht.
- SHA-1 wurde bezüglich der Verteilung der Ausgabewerte bereits ausführlich untersucht, sodass man alleine bei den berechneten SHA-1-Werten von einer Verteilung nahe einer Gleichverteilung ausgeht.
- Zudem wird vor der Ausgabe von jeweils 80 Bit noch eine Faltung des SHA-1-Werts durchgeführt. Diese Faltung trägt ebenfalls zu einer gleichmäßigen Verteilung der Ausgabe bei.

Somit halten wir fest, dass `/dev/random` die Anforderungen von NTG.1.4 erfüllt.

Bemerkung

Auch wenn Nutzer keinen direkten Zugriff auf den `input_pool` haben, halten wir fest, dass die vorgestellte Untersuchung sowohl für die Extraktion aus dem `blocking_pool` als auch dem `input_pool` gültig ist.

4.4.2 `/dev/urandom`

Die für `/dev/random` getroffene Best-Case Analyse in Abschnitt 4.4.1 basiert rein auf der Gleichverteilung, welche durch SHA-1 gegeben ist. Demzufolge gilt diese Analyse ebenfalls für `/dev/urandom`, da der gleiche Extraktionsmechanismus wie für `/dev/random` verwendet wird. In Abschnitt 4.8 ist erläutert, dass ein Aufrufer von `/dev/urandom` immer Zufallszahlen erhält, auch wenn die Entropieschätzung für den LRNG dies nicht erlauben. In diesem Fall basiert die Zufallszahl auf der Qualität von SHA-1.

Damit erfüllt `/dev/urandom` die Anforderungen von NTG.1.4.

4.5 NTG.1.5

Die Anforderung NTG.1.5 ist definiert als:

„Statistical test suites cannot practically distinguish the internal random numbers from output sequences of an ideal RNG. The internal random numbers must pass test procedure A [assignment: additional test suites].“

Die Durchführung der Test Procedure A entsprechend [AIS2031] wurde im Abschnitt 7.3 durchgeführt.

Die Anforderung an die zusätzlichen Tests wird mit den Analysen in Kapitel 5 ff. als abgedeckt angesehen. Dies gilt für alle Entropie-Pools. Vor allem die Tests in Abschnitten 7.4 und 7.5 sind explizite Bestätigungen für die Anforderungen von NTG.1.5.

Entsprechend Abschnitten 7.4 und 7.5 kann das Assignment der Anforderung von NTG.1.5 wie folgt ausgestaltet werden: „and the BSI Test Suite A as well as the dieharder¹⁹ test suite“

Damit erfüllen `/dev/random` und `/dev/urandom` die Anforderungen von NTG.1.5.

4.6 NTG.1.6

Die Anforderung NTG.1.6 ist definiert als:

„The average Shannon entropy per internal random bit exceeds 0.997.“

Die Idee, die Erfüllung dieser Anforderung zu zeigen ist nun wie folgt:

Wir zeigen experimentell, dass die Entropieschätzung Ereignisse im Mittel mit deutlich weniger Entropie bewertet, als deren tatsächliche Shannon-Entropie ist. Somit enthält der `input_pool` praktisch immer deutlich mehr Shannon-Entropie als der Entropiezähler des LRNG anzeigt. Fordert nun `blocking_pool` N Bit Entropie an, so wird diese (falls vorhanden)

19 Test suite is provided at <http://www.phy.duke.edu/~rgb/General/dieharder.php>.

(i) aus dem `input_pool` übertragen, (ii) der Entropiezähler des `input_pool` um N verringert und (iii) der Entropiezähler des `blocking_pool` um N erhöht. Somit enthält auch der `blocking_pool` stets mehr Shannon-Entropie als der Entropiezähler des LRNG angibt. Da für die Ausgabe von N Zufallsbits aus dem `blocking_pool` der Entropiezähler mindestens N betragen muss und dieser Zähler nach der Ausgabe um N erniedrigt wird, folgt (informell), dass jedes ausgegebene Zufallsbit ein Bit Shannon-Entropie enthält, die Anforderung NTG.1.6 somit erfüllt ist.

Es folgt jetzt die detaillierte Ausführung dieser Idee.

Die Anforderung zielt auf den Vergleich zwischen der theoretischen Shannon-Entropie und der LRNG-Heuristik für das Abschätzen der Entropie ab. Nur wenn die heuristische Entropieschätzung pro Ereignis maximal gleich oder geringer ist als die Shannon-Entropie pro Ereignis, ist diese Anforderung als gesichert anzusehen. Dabei ist anzumerken, dass ein Bit an heuristisch ermittelter Entropie vom LRNG als ein Bit an „wahrer“ Entropie angesehen wird.

Abschnitt 6.2.7 enthält genau diesen Vergleich, wenn auch nicht für alle heuristischen Entropiewerte²⁰. Diese Analyse sagt Folgendes aus:

- Heuristische Entropiewerte von 0 enthalten eine Shannon-Entropie von rund 1,86. Damit ist die Heuristik des LRNG sehr konservativ.
- Heuristische Entropiewerte von 8 enthalten eine Shannon-Entropie von 3,39. Damit ist die Heuristik des LRNG zu optimistisch.

Abschnitt 6.2.1 enthält ein Histogramm pro Rauschquelle mit der relativen Anzahl von heuristischen Entropiewerten. Dabei treten die heuristischen Entropiewerte von 0 in 60% bis 90% aller Ereigniswerte auf – wir nehmen hier den Wert Mittelwert entsprechend der unten stehenden Tabelle. Die heuristischen Entropiewerte von 8 treten in weniger als 1% der Fälle auf entsprechend unten stehender Tabelle.

Aus den in Abschnitt 6.2.1 verwendeten Daten kann man folgende Tabelle erstellen – diese Tabelle wurde mit Hilfe des R-Programms `entropy_per_event/shannon_vs_heuristic.r` erstellt. Dabei sind die Spalten folgendermaßen zu interpretieren:

- Erste Spalte: Heuristische Entropiewerte.
- HID: Die Häufigkeit der heuristischen Entropiewerte der Eingabegeräte. Diese Spalte ist identisch mit dem entsprechenden Histogramm in Abschnitt 6.2.1.
- Disk: Die Häufigkeit der heuristischen Entropiewerte der Blockgeräte. Diese Spalte ist identisch mit dem entsprechenden Histogramm in Abschnitt 6.2.1.
- Mean: Der Mittelwert der Spalten HID und Disk.
- Entropie: Multiplikation des Mittelwertes mit den Heuristischen Entropiewerten.
- Kum. Entropie: Kumulierte Entropie – Addition der Entropiewerte.

	HID	Disk	IRQ	Mean	Entropie	Kum. Entropie
0	0.6268	0.9147	0.0016	0.5144	0	0
1	0.2351	0.0597	0.9982	0.431	0.431	0.431
2	0.0396	0.0131	0.0001	0.0176	0.0352	0.4662

²⁰ Um eine vollständige und umfassende Analyse vorlegen zu können, müssten in Abschnitt 6.2.7 Tests für alle theoretisch möglichen heuristischen Entropiewerte, d.h. 0 bis einschließlich 11 durchgeführt werden. Es wird aber angenommen, dass die massive Unterschätzung der Entropie durch den hohen Anteil der heuristischen Entropie von 0 Bits (zwischen 60 und 80% aller Ereignisse werden mit 0 Bit Entropie abgeschätzt) alle möglichen Überschätzungen in den höheren heuristischen Entropiewerten mehr als ausgleicht.

3	0.0381	0.0005	0	0.0129	0.0386	0.5048
4	0.0246	0.0008	0	0.0085	0.0338	0.5386
5	0.0245	0.003	0	0.0092	0.0458	0.5844
6	0.0038	0.0023	0	0.002	0.0122	0.5966
7	0.0025	0.0042	0	0.0022	0.0156	0.6122
8	0.0014	0.0015	0	0.001	0.0079	0.6201
9	0.001	0.0002	0	0.0004	0.0036	0.6237
10	0.0008	0	0	0.0003	0.0026	0.6263
11	0.0019	0	0	0.0006	0.0071	0.6334

Der letzte Wert in der Spalte „Kum. Entropie“ zeigt die durchschnittliche Entropie pro Ereignis bezogen auf alle möglichen heuristischen Entropiewerte.

Damit kann man folgende Analyse durchführen:

- Heuristische Entropie - aus Tabelle: 0,6334
- Shannon-Entropie: $1,86 * 0,5144 + 3,39 * 0,001 = 0,96$

Damit ergibt sich, dass die Shannon-Entropie über die beobachteten Ereignisse alleine bei den gemessenen heuristischen Entropiewerten 0 und 8 erheblich höher ist, als die heuristisch ermittelte Entropie über alle heuristischen Entropiewerte. Dies wiederum impliziert, dass die Anforderung an `input_pool` als erfüllt anzusehen ist.

Der Transfer von Entropie vom `input_pool` zu dem `blocking_pool` und `nonblocking_pool` ist derart gestaltet, dass eine Entnahme von einem Bit Zufall aus dem `input_pool` die Entropieschätzung des `input_pools` um ein Bit verringert. Gleichzeitig wird beim Hinzufügen dieses Bits zum `blocking_pool` oder `nonblocking_pool` die jeweilige Entropieschätzung um ein Bit erhöht. Damit verändert der Transfermechanismus die oben getroffene Aussage nicht – sie trifft demnach auf den gesamten LRNG zu.

Zu beachten ist, dass die oben genannten empirischen Werte durch Tests auf x86- Systemen ermittelt wurden. Wie aus den Tests in Abschnitt 6.2 ersichtlich, steuern die Prozessorzyklen den Großteil der Entropie bei. Die Prozessorzyklen werden entsprechend der Beschreibung im Abschnitt 2.5.1.5 hardwareabhängig gelesen, wobei die Möglichkeit besteht, dass auf einigen Plattformen überhaupt keine Zyklen gelesen werden. Es wird vermutet, dass die heuristische Entropieabschätzung im LRNG das Fehlen bzw. das veränderte Maß der Prozessorzyklen abfängt. Dennoch gilt die getroffene Aussage für die Messungen auf x86-Systemen und kann nicht einfach auf andere Hardwareplattformen übertragen werden.

Somit halten wir fest, dass `/dev/random` die Anforderungen von NTG.1.6 erfüllt.

4.7 Struktur des LRNG

Die Anforderung an die Struktur des LRNG ist definiert als:

„A non-physical true RNG comprises three parts:

- the input pre-computation block, which computes the input for the internal DRNG from several external input signals provided by (usually several) entropy sources,
- the entropy pool, which collects entropy and computes the output,
- the control block, which prevents the output of random numbers until the RNG has sufficient entropy to ensure the randomness of the output.“

Die Betrachtung des LRNG, ob die genannten Anforderungen abgedeckt sind, ist im Folgenden separiert nach den Benutzerschnittstellen /dev/random und /dev/urandom.

4.7.1 /dev/random

Der erste Punkt ist wie folgt abgedeckt:

- Als Rauschquellen werden prinzipiell 3 Quellen verwendet:
 - Zeitverhalten der Zugriffe auf Blockgeräte,
 - Zeitverhalten von Ereignissen erzeugt von Eingabegeräten, und
 - Zeitverhalten des Auftretens von Interrupts.

Dabei ist zu sagen, dass die Interrupt-Quelle in der jetzigen Implementierung im Endeffekt deaktiviert ist.

- Die Ereignisse der Rauschquellen werden in einem Entropie-Pool, dem `input_pool`, gesammelt. Dabei werden die neuen Ereignisse mittels eines linearen Schieberegisters mit `Twist` zu den bestehenden Werten des `input_pools` hinzugefügt.

Der zweite Punkt ist wie folgt abgedeckt:

- Der Entropie-Pool `blocking_pool` sammelt die Entropie aus dem `input_pool`. Weiterhin wird aus dem `blocking_pool` die Zufallszahl erzeugt.

Der dritte Punkt ist wie folgt abgedeckt:

- Die Lese-Funktion von /dev/random stoppt den lesenden Prozess im Falle des Fehlens von Entropie im `input_pool`. Der gestoppte Prozess wird fortgesetzt, sobald Entropie bereit steht.

4.7.2 /dev/urandom

Der erste Punkt ist wie folgt abgedeckt:

- Als Rauschquellen werden prinzipiell 3 Quellen verwendet:
 - Zeitverhalten der Zugriffe auf Blockgeräte,
 - Zeitverhalten von Ereignissen erzeugt von Eingabegeräten, und
 - Zeitverhalten des Auftretens von Interrupts.

Dabei ist zu sagen, dass die Interrupt-Quelle in der jetzigen Implementierung im Endeffekt deaktiviert ist.

- Die Ereignisse der Rauschquellen werden in einem Entropie-Pool, dem `input_pool`, gesammelt. Dabei werden die neuen Ereignisse mittels eines linearen Schieberegisters mit `Twist` zu den bestehenden Werten des `input_pools` hinzugefügt.

Der zweite Punkt ist wie folgt abgedeckt:

- Der Entropie-Pool `nonblocking_pool` sammelt die Entropie aus dem `input_pool`. Weiterhin wird aus dem `nonblocking_pool` die Zufallszahl erzeugt.

Der dritte Punkt ist aber **nicht** abgedeckt. /dev/urandom kann demzufolge nicht als konsistent mit den Anforderungen von NTG.1 angesehen werden.

4.8 DRG.3 Anforderungen

Die Anforderung für DRG.3 ist wie folgt formuliert:

„The class NTG.1 combines security capabilities of deterministic RNGs and security capabilities similar to those of physical true RNGs. By clause (NTG.1.3) the entropy pool with its updating mechanism and output function (viewed as a DRNG) is DRG.3-conformant.“

Bei der Betrachtung im Rahmen von DRG.3 muss man die Entropie-Pools isoliert voneinander betrachten, da DRG.3 nur einen deterministischen RNG definiert. Demzufolge gilt:

- `blocking_pool` und `nonblocking_pool` stellen jeweils einen eigenständigen DRNG dar.
- `input_pool` ist ein NPTRNG. Da dieser Pool maßgeblich an der Qualität von `/dev/random` beteiligt ist und darüber hinaus auch die Ausgabe von Zufallszahlen stoppt, wenn im `input_pool` zu wenig Entropie vorhanden ist, kann der `input_pool` selbst als NPTRNG entsprechend NTG.1 angesehen werden.

Demzufolge beziehen sich die Anforderungen von DRG.3 auf `blocking_pool` und `nonblocking_pool`. Der `input_pool` wird entsprechend des in Kapitel 2 erläuterten Designs nicht für die Ausgabe von Zufallszahlen verwendet. Dieser Entropie-Pool muss als Seed-Quelle für die beiden anderen Pools angesehen werden. Wenn man die Verwaltung dieser Entropie-Pools isoliert betrachtet, stellen sie jeweils deterministische RNG dar. Damit können sie auf Erfüllung der Anforderungen von DRG.3 geprüft werden.

Die folgenden Kapitel diskutieren die Anforderungen von DRG.3 und zeigen, dass sowohl der `blocking_pool` als auch der `nonblocking_pool` Entropie-Pool die DRG.3 Anforderungen erfüllen.

4.8.1 DRG.3.1

„If initialized with a random seed [selection: using PTRNG of class PTG.2 as random source, using PTRNG of class PTG.3 as random source, using NPTRNG of class NTG.1 as random source, [assignment: other requirements for seeding]], the internal state of the RNG shall [selection: have [assignment: amount of entropy], have [assignment: work factor], require [assignment: guess work]].

Sowohl der `blocking_pool`, als auch der `nonblocking_pool` werden vom `input_pool` geseeded, welcher ein NPTRNG der Klasse NTG.1 ist.

Bezüglich der Entropie gilt:

- Für `blocking_pool` gilt Aussage in Abschnitt 4.2, aufgrund der Eigenschaft des Blockierens von Leseoperationen, wenn die Entropieschätzung zu niedrig ist. Weiterhin ist zu bedenken, dass die Entropieschätzung, welche die Entropie in Bits angibt, um eins verringert wird, wenn ein Bit an Zufall extrahiert wird.
- Für `nonblocking_pool` gilt aufgrund der Eigenschaft der fehlenden Blockierung folgendes: Unter folgenden Annahmen:
 - dass die initiale Installation genügend Entropie (mindestens 100 Bits, aber typisch eher 1000 Bits entsprechend der Entropieschätzung – entsprechende Tests wurden durchgeführt) erzeugt,
 - dass entsprechend den Nutzungsanforderungen in Abschnitt 2.5.1.7 die Entropie für den ersten Neustart gespeichert wird, und
 - dem absoluten Worst-Case Szenario, dass nach der Installation keine Entropie mehr von Hardware-Ereignissen aufgezeichnet werden (z.B. headless System bei dem niemals HID Ereignisse anfallen und ein System mit SSD, welche entsprechend Abschnitt 2.5.1.7 keine Entropie liefern)

gilt folgendes: Aufgrund der Nutzung von SHA-1 als Whitening Funktion bleibt die Entropie, welche beim Start eingemischt wird, erhalten.

Selbst wenn ein Angreifer den Entropie-Pool ständig lesen würde und den für ihn Best-Case (für uns Worst-Case) annimmt, dass:

- das angegriffene System ohne Last ist,
- der Angreifer die optimale Systemlast erzeugt, indem er so viele Leseprozesse erzeugt, wie CPUs vorhanden sind (mehr würde zusätzliche Last vom Betriebssystem erzeugen, weniger würde die Leistung des System nicht ausreizen),
- der Aufwand, die gelesenen Daten wegzuspeichern (z.B. auf Platte), würde Null betragen, und

- der Aufwand, die gelesenen Daten zu analysieren und damit zukünftige oder vergangene Zufallszahlen zu berechnen, würde Null betragen,

können folgende Messungen durchgeführt werden: Auf einem Testsystem mit einem Intel i7 2,7 GHz Sandy Bridge Prozessor ist die Lesegeschwindigkeit aus /dev/urandom beim Lesen von 50 MByte:

```
dd if=/dev/urandom of=/dev/null count=100000
100000+0 records in
100000+0 records out
51200000 bytes (51 MB) copied, 5.34589 s, 9.6 MB/s
```

Die generierten Daten werden nach /dev/null geschrieben, um jegliche Systemlast für das Speichern auszuklammern.

Zur einfacheren Berechnung, nehmen wir an, dass wir eine Lesegeschwindigkeit von 16 MB/s haben (fast 60% über der gemessenen Geschwindigkeit). Um zum Beispiel 2^{64} Bit zu lesen benötigt die Leseoperation folgende Zeit:

16 MB/s entsprechen 2^{24} Bytes/s, also 2^{61} Bytes. Demnach benötigt der Lesevorgang $2^{61} / 2^{24}$ s, was etwa 4358 Jahren entspricht.

Selbst auf einem System mit 64 Prozessoren benötigt die Leseoperation noch etwa $4358/64=68$ Jahre. Nehmen wir an, dass jeder Prozessor 3 Hyperthreads und damit im Endeffekt 192 Prozessoren hat, benötigt das Lesen noch über 20 Jahre. Wohlgedemert, alles unter der unwahrscheinlichen Annahme, dass keine Hardware-Ereignisse auftreten.

Mit dieser beispielhaften Berechnung lässt sich ableiten, dass sich die Entropie im nonblocking_pool in einem angemessenen Zeitraum nicht verändert, da sie mittels der Qualität von SHA-1 geschützt wird.

Damit kann festgestellt werden, dass die initiale Entropie in /dev/urandom über die gesamte Laufzeit des Zufallszahlengenerators sich nicht verändert – wie oben genannt kann als Worst-Case ein Minimum von 100 Bit angenommen werden.

4.8.2 DRG.3.2

„The RNG provides forward secrecy.“

Die Nutzung von SHA-1 als Whitening Funktion und das Einmischen eines SHA-1 Wertes entsprechend der Illustration in Abschnitt 4.3 stellt sicher, dass der Entropie-Pool bei jeder Erzeugung einer Zufallszahl gemischt wird:

- Das Einmischen eines SHA-1 Wertes in den Entropie-Pool stellt sicher, dass der Pool verändert wird bevor eine Zufallszahl entnommen wird. Dabei ist zu bemerken, dass der erzeugte SHA-1 Hash nur einen Teil des Pools verändert.
- Die Berechnung eines SHA-1 Wertes über den gesamten Entropie-Pool stellt sicher, dass jegliche Schiefen in der Wahrscheinlichkeitsverteilung der Bits geglättet werden.

Damit lässt sich aussagen, dass mittels SHA-1 die forward secrecy gewährleistet wird.

4.8.3 DRG.3.3

„The RNG provides backward secrecy even if the current internal state is known.“

Die Analyse aus Abschnitt 4.3 hat gezeigt, dass alle drei Entropie-Pools des LRNG die Anforderung NTG.1.3 erfüllen. Somit **erfüllen** sowohl /dev/random als auch /dev/urandom erst recht DRG.3.3 und damit DRG.3.

Wir bemerken noch, dass die Problematik der (Enhanced-)Backward-Secrecy den Entwicklern des LRNG bewusst war und ist. Dementsprechend wurde das in der Konstruktion berücksichtigt und auch in einem Kommentar

```
/*
 * We mix the hash back into the pool to prevent backtracking
 * attacks (where the attacker knows the state of the pool
 * plus the current outputs, and attempts to find previous
```

```
* outputs), unless the hash function can be inverted. By  
* mixing at least a SHA1 worth of hash data back, we make  
* brute-forcing the feedback as hard as brute-forcing the  
* hash.  
*/
```

innerhalb der Funktion `extract_buf` vermerkt.

4.8.3.1 DRG.3.4

„The RNG initialized with a random seed [assignment: requirements for seeding] generates output for which [assignment: number of strings] strings of bit length 128 are mutually different with probability [assignment: probability].“

Siehe Abschnitt 4.4.

Damit erfüllen sowohl `/dev/random` als auch `/dev/urandom` diese Forderung.

4.8.3.2 DRG.3.5

DRG.3.5: „Statistical test suites cannot practically distinguish the random numbers from output sequences of an ideal RNG. The random numbers must pass test procedure A [assignment: additional test suites].“

Per Definition müssen alle Zufallszahlen, welche vom `blocking_pool` und vom `nonblocking_pool` abgeleitet werden, die genannten Tests bestehen. Der Grund hierfür liegt in der Nutzung von SHA-1. Die ausgegebene Zufallszahl ist ein SHA-1 Wert über den gesamten Pool. Wenn die genannten Tests Probleme aufzeigen, würde dies im Umkehrschluss bedeuten, dass Eigenschaften von SHA-1 nicht mehr gelten und damit SHA-1 gebrochen wurde.

4.9 Vorhandensein einer Entropieschätzung

Die Anforderung für das Vorhandensein einer Entropieschätzung ist wie folgt formuliert:

„The security capability (NTG.1.1) checks the external input signals from the entropy sources with regard to total failure and non-tolerable weaknesses. Usually, an ‘entropy counter’ (applying heuristic rules) is kept to provide plausibility that enough fresh entropy is mixed up with the current internal state. The entropy counter reduces the (estimated) entropy of the internal state by m whenever m bits are output. If the value of the entropy counter is smaller than m the output of an m -bit string is prohibited. One says the input is “rated for entropy estimation“.“

Die Betrachtung des LRNG, ob die genannten Anforderungen abgedeckt sind, ist im Folgenden separiert nach den Benutzerschnittstellen `/dev/random` und `/dev/urandom`.

4.9.1 /dev/random

Die Entropie-Pools, welche bei `/dev/random` eine Rolle spielen, sind der `input_pool` und der `blocking_pool`. Für beide Entropie-Pools hält der LRNG eine Entropieschätzung vor, welche ständig entsprechend den eingehenden und ausgehenden Werten verändert wird.

Falls die Entropieschätzung für den `blocking_pool` kleiner ist als die angeforderte Zufallszahl, dann versucht der LRNG die fehlende Entropie aus dem `input_pool` in den `blocking_pool` zu übertragen. Im Fall, dass selbst der `input_pool` zu wenig Entropie zur Befriedigung des Aufrufers enthält, wird der aufrufende Prozess einfach gestoppt und erst wieder fortgesetzt, wenn genügend Entropie vorhanden ist.

Beim Erzeugen der Zufallszahl aus dem `input_pool` oder `blocking_pool` wird die Entropieschätzung um die Anzahl der erzeugten Bits verringert.

Dabei ist zu beachten, dass die Lesefunktion von `/dev/random` die angeforderte Größe der Zufallszahl blockweise abarbeitet – dabei ist ein Block 80 Bit, entsprechend der Größe des gefalteten SHA-1 Hashes, welcher von dem Entropie-Pool berechnet wurde. D.h. wenn genügend Entropie für die Erzeugung eines Blocks an Zufallsbits vorhanden ist, wird der Block generiert und dem Aufrufer übergeben. Ansonsten wird die Erzeugung des Blocks wie beschrieben gestoppt.

Damit erfüllt `/dev/random` die Anforderung an die Entropieschätzung.

4.9.2 `/dev/urandom`

Die Entropie-Pools, welche bei `/dev/urandom` eine Rolle spielen, sind der `input_pool` und der `nonblocking_pool`. Für beide Entropie-Pools hält der LRNG eine Entropieschätzung vor, welche ständig entsprechend den eingehenden und ausgehenden Werten verändert wird.

Falls die Entropieschätzung für den `nonblocking_pool` kleiner ist als die angeforderte Zufallszahl, versucht der LRNG die fehlende Entropie aus dem `input_pool` in den `nonblocking_pool` zu übertragen. Wenn der `input_pool` nicht ausreichend Entropie besitzt, wird dennoch ein Zufallswert aus dem `input_pool` berechnet, der zu dem `nonblocking_pool` hinzugefügt wird. Da die Extraktionsfunktion gleichzeitig eine Änderung des `input_pools` verursacht, werden ständig neue, aufgrund von SHA-1 nicht vorhersagbare Werte zu dem `nonblocking_pool` hinzugefügt. Dies bedeutet, dass obwohl die Entropieschätzung zu wenig Entropie signalisiert, Zufallszahlen erzeugt werden.

Beim Erzeugen der Zufallszahl aus dem `input_pool` oder `nonblocking_pool` wird die Entropieschätzung um die Anzahl der erzeugten Bits verringert, bis minimal auf 0.

Damit erfüllt `/dev/urandom` die Anforderung an die Entropieschätzung **nicht**.

5 Testreihen I: Untersuchung der Entropie-Pools

In Kapitel 4 haben wir gesehen, dass `/dev/urandom` die Anforderungen an DRG.3 erfüllt und `/dev/random` sogar konform zu NTG.1 ist. Dort wurde insbesondere betrachtet, welche Möglichkeiten ein Angreifer hat, wenn er den Inhalt eines oder mehrerer Entropie-Pools zu einem bestimmten Zeitpunkt kennt. Neben der physischen Kontrolle über ein System könnte auch eine extrem einseitige Verteilung der Daten in den Entropie-Pools ein solches Szenario begünstigen. In diesem Kapitel wird mit Hilfe einiger Messungen getestet, ob die Entropie-Pools des LRNG die zuletzt genannte (nicht gewünschte) Eigenschaft besitzen.

Für die Sammlung der relevanten Daten wurde, wie in der Einleitung bereits erwähnt, SystemTap verwendet; die Diagramme wurden mit Hilfe der Programmiersprache R erstellt. Eine ausführliche Beschreibung der technischen Aspekte, sowie die Angaben zur eingesetzten Hardware sind in Appendix A zu finden.

Alle Tests wurden mit folgender Version des Linux-Kerns durchgeführt: 3.6.

5.1 Motivation und Erklärung der verschiedenen Tests

Bevor wir einen kurzen Überblick über die durchgeführten Messungen geben, sollte bemerkt werden, dass alle Tests jeweils mit einer vorher festgelegten Stichprobengröße S durchgeführt wurden. Durch zeitliche Einschränkungen bedingt, wird S dabei meist im Bereich von 1000 bis 100000 gewählt. Aufgrund der dazu vergleichsweise enormen Größe der Entropie-Pools (4096 Bit bzw. 1024 Bit) ist die Aussagekraft einiger Tests sehr beschränkt, zudem rechnet man auch mit etwas größeren Abweichungen von erwarteten Verteilungen.

Wenn nicht anders angegeben, wurden alle Tests sowohl im laufenden Betrieb als auch zum Systemstart jeweils für alle drei Entropie-Pools durchgeführt. Bei den Tests zum Systemstart wird bei jedem Startvorgang nur zu zwei bestimmten Zeitpunkten (einmal vor Start des User-Space und einmal vor Hinzufügen des Seeds) gemessen, danach wird ein Restart initiiert. Wichtig ist hierbei zu bemerken, dass im Gegensatz dazu die Tests im laufenden Betrieb eine bestimmte Anzahl direkt aufeinander folgende Zustände untersuchen.

Die Motivation für die ersten drei Tests liefert eine wünschenswerte und vielleicht auch erwartete Eigenschaft: Die Daten der Entropie-Pools sollten annähernd einer Gleichverteilung folgen. Während das extreme Gegenteil, also das Auftreten lediglich einiger weniger Zustände, einem Angreifer offensichtlich sehr zu Gute kommen würde, muss hier bemerkt werden, dass auch eine stärkere Abweichung von einer Gleichverteilung die Qualität des LRNG nicht vollständig untergräbt. Selbst die Annahme, dass 2^{200} Belegungen (das entspricht beim `input_pool` einem Anteil von 2^{-3896} und bei den Output Pools einem Anteil von 2^{-824}) zusammen mit einer Wahrscheinlichkeit von über 50% auftreten, würde alleine keinen auch nur annähernd praktikablen Angriff ermöglichen.

Aufgrund des Designs des LRNG, genauer gesagt dem Einmischen von Hashwerten des Pools in den Pool bei jeder Extraktion von Zufallszahlen, erwartet man natürlich eine Verteilung der Pooldaten, die qualitativ ähnlich nahe einer Gleichverteilung ist, wie sie die verwendete Hashfunktion (SHA-1) aufweist.

- **Test 1 (Abschnitt 5.2) - Bestimmung der Anzahl der gesetzten Bits (=Bit-Summe) innerhalb des Entropie-Pools in S Beobachtungen:** Ausgehend von einer Gleichverteilung der Daten im jeweiligen Entropie-Pool erwartet man, dass der Mittelwert der gemessenen Bit-Summen etwa der Hälfte der Poolgröße entspricht und die Bit-Summen einer Normalverteilung folgen. Eine deutliche Abweichung würde hier für eine Abweichung von der Gleichverteilung sprechen. Ein positives Testergebnis kann jedoch auch bei einer extremen Ungleichverteilung auftreten, etwa wenn bei allen Messungen die erste Hälfte der Bits des Entropie-Pools gesetzt ist und die andere Hälfte nicht. Bei den Tests zum Systemstart werden auch die Bit-Summen innerhalb der einzelnen 32-Bit-Wörter untersucht.
- **Test 2 (Abschnitt 5.3) - Bestimmung der Anzahl der gesetzten Bits für jede einzelne Bitposition innerhalb des Entropie-Pools in S Beobachtungen:** Geht man wieder von einer Gleichverteilung der Daten des Entropie-Pools und zusätzlich

von einer hinreichenden Stichprobengröße aus, so erwartet man, dass jede Bit-Summe nahe $S/2$ liegt. Mit anderen Worten: Die Wahrscheinlichkeit, dass das Bit an einer beliebig, aber fest gewählten Bitposition des Entropie-Pools gesetzt ist, liegt nahe 50%. Auch hier ist ein positives Testergebnis nicht hinreichend, wie das Beispiel der kontinuierlich abwechselnden Belegungen „101010...“ und „010101...“ zeigt.

Zusätzlich wird die Anzahl der gesetzten Bits pro Bitposition auch innerhalb der einzelnen 32-Bit-Wörter untersucht. Dies ist auch als Test auf Abhängigkeiten innerhalb eines Wortes zu sehen.

Nach diesen Tests kann man folgende Überlegung anstellen: Angenommen für jede Bitposition i des Entropie-Pools gilt

$$P(b_i=1) \approx P(b_i=0) \approx \frac{1}{2}.$$

Weiter sei n die Größe des betrachteten Entropie-Pools in Bit und seien a_1, \dots, a_n beliebig aber fest gewählte Bits. Nimmt man nun an, dass die Bits innerhalb der Entropie-Pools unabhängig voneinander gesetzt sind, gilt für die Belegung b_1, \dots, b_n des betrachteten Pools

$$P(b_1=a_1, b_2=a_2, \dots, b_n=a_n) = P(b_1=a_1) \cdot P(b_2=a_2) \cdot \dots \cdot P(b_n=a_n) = \left(\frac{1}{2}\right)^n,$$

was gleichbedeutend mit der Gleichverteilung innerhalb des Entropie-Pools wäre.

Die angesprochene stochastische Unabhängigkeit der Bits der Entropie-Pools wäre offenbar eine sehr wünschenswerte Eigenschaft der Entropie-Pools.

- **Test X - Messung S aufeinander folgender Zustände des Entropie-Pools und damit stichprobenhafte Untersuchung einzelner Bitpositionen und Bitgruppen auf stochastische Unabhängigkeit:** Dieser Test wird hier nicht durchgeführt, da aufgrund der enormen Größe der Entropie-Pools und der sich daraus ergebenden noch größeren Anzahl an möglichen Abhängigkeiten eine Untersuchung, die ein einigermaßen aussagekräftiges Ergebnis liefert, vollkommen außer Reichweite ist.

Natürlich wäre es wünschenswert, dass sich der Inhalt der Entropie-Pools zu unterschiedlichen Beobachtungszeitpunkten in etwa der Hälfte der Bits unterscheidet, sodass die Kenntnis eines Zustands des Pools auch aus statistischen Gründen keine Rückschlüsse auf die Belegung zu einem anderen Zeitpunkt zulässt. Auch wenn aufgrund des Designs ein derartiges Verhalten zu erwarten ist, soll folgender Test als dieses Argument untermauern:

- **Test 3 (Abschnitt 5.4) - Bestimmung der Anzahl an Änderungen für jede einzelne Bitposition innerhalb des Entropie-Pools in S Beobachtungen:** Man könnte annehmen, dass der erwartete und gewünschte Wert für jede einzelne Bitposition bei etwa der Hälfte der Stichprobengröße, also bei $S/2$ liegt. Dies wäre gleichbedeutend damit, dass bei jedem Einmischen neuer Daten in den Pool jedes Bit mit einer Wahrscheinlichkeit von etwa 50% verändert wird. Allerdings muss hier in Betracht gezogen werden, dass beim Einmischen von Daten in den Pool gar nicht alle Bits des Entropie-Pools einer potentiellen Änderung unterliegen. Die theoretischen Werte für die oben genannten Wahrscheinlichkeiten werden in Abschnitt 5.4.1 hergeleitet. **Aufgrund der unterschiedlichen Testdurchführungen ist diese Bemerkung nur für die Tests im laufenden Betrieb, nicht jedoch für die Tests zum Systemstart relevant. Bei diesen erwartet (und wünscht) man tatsächlich, dass für jede Bitposition etwa $S/2$ Änderungen gemessen werden.** Das Beispiel mit einer kontinuierlichen Wiederholung der „111...“, „111...“, „000...“, „000...“ (auf zwei identische Zustände folgen zwei Zustände, bei denen alle Bits geflippt wurden) zeigt die Grenzen der Aussagekraft von Test 3 auf.

Wie bei Test 2 wird zusätzlich die Anzahl der Änderungen für jede Bitposition auch innerhalb der einzelnen 32-Bit-Wörter untersucht, um eventuell Abhängigkeiten aufzudecken zu können.

Zur Veranschaulichung der Unterschiede zwischen Test 1 (Abschnitt 5.2), Test 2 (Abschnitt 5.3) und Test 3 (Abschnitt 5.4), soll folgendes Beispiel dienen:

Gegeben sei ein Entropie-Pool mit 10 Bits. Des Weiteren wird eine Stichprobengröße von 3 angenommen. Dann können die vorliegenden Untersuchungen wie folgt dargestellt werden:

Bitposition	1	2	3	4	5	6	7	8	9	10	Werte für 5.2 (Test 1)
1. Beobachtung	1	0	0	0	1	1	0	1	1	1	6
2. Beobachtung	0	0	1	1	0	1	1	1	0	0	5
3. Beobachtung	0	1	1	1	0	1	0	1	1	1	7
Werte für 5.3 (Test 2)	1	1	2	2	1	3	1	3	2	2	
Werte für 5.4 (Test 3)	1	1	1	1	1	0	2	0	2	2	

Die Testresultate werden meistens mittels selbsterklärenden Box-Whisker-Plots und Histogrammen veranschaulicht und dann noch interpretiert. Die Histogramme beinhalten zusätzlich zu der graphischen Veranschaulichung der Daten die theoretische Normalverteilung als rot gestrichelte Linie wenn die Daten aus einem perfekten RNG stammen würden - dies bedeutet, dass die aufgezeigte Normalverteilung aus der Standardabweichung (Sigma), dem Mittelwert der beobachteten Daten und der Stichprobengröße errechnet wurde.

Dabei ist zu bemerken, dass eine Abweichung der Testergebnisse von den theoretischen Vorhersagen bei den Tests zum Systemstart grundsätzlich als Hinweis auf die Notwendigkeit des Reseedings zu werten ist, falls keine andere Erklärung für die Abweichungen vorliegt.

Bemerkung zu der Aussagekraft der Tests

Wir bemerken, dass bei Vorliegen einer wünschenswerten Verteilung der Daten in den Entropie-Pools alle Tests positiv beendet werden, ein positives Ergebnis in allen Tests aber nicht als Nachweis einer wünschenswerten Verteilung gelten kann. Betrachten wir als Beispiel ein kontinuierliches Abwechseln der folgenden Zustände des Entropie-Pools:

```
101010...
101010...
010101...
010101...
```

Sei P die Größe des Entropie-Pools in Bit und S der Stichprobenumfang. Test 1 liefert hier $P/2$ für jeden Zustand, Test 2 ergibt für jede Bitposition etwa $S/2$ als Anzahl der gesetzten Bits und Test 3 zeigt etwa $S/2$ Änderungen für jede Bitposition.

Wie zu Beginn der Testbeschreibungen bereits erwähnt wird, ist ein so extremes Verhalten aufgrund der Funktionsweise des LRNG nicht zu erwarten.

5.2 Test 1 - Gesetzte Bits innerhalb des Entropie-Pools

Wie in 5.1 beschrieben, wird in diesem Test zu jedem Beobachtungszeitpunkt die Anzahl der gesetzten Bits innerhalb des jeweils betrachteten Entropie-Pools bestimmt.

5.2.1 Test 1 im laufenden Betrieb - Testansatz

Ausgehend von den Anfangsüberlegungen wurde folgender Test erstellt, welcher im Verzeichnis `entropy_pool_distribution` zu finden ist:

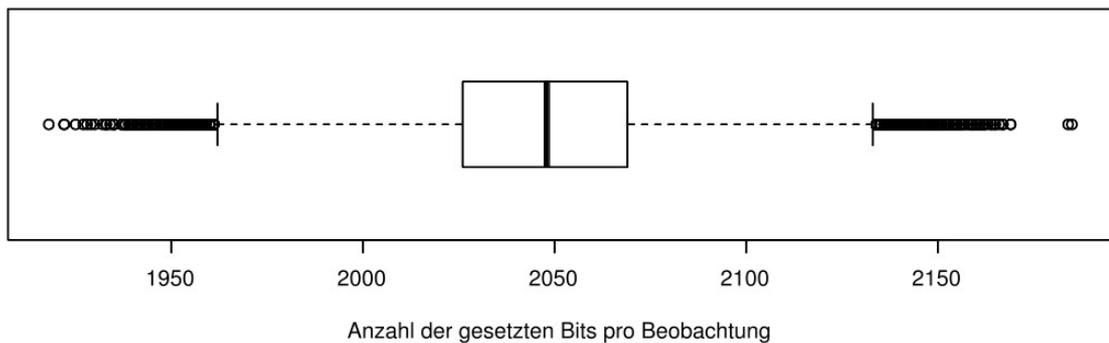
- Ein SystemTap-Skript `entropy_pool_distribution.stp` wurde erstellt, in welchem die Stichprobenanzahl mit der Variable `num_samplings` angepasst werden kann. Dieses Skript bestimmt die Anzahl der gesetzten Bits im jeweils betrachteten Entropie-Pool bei jedem Auslesen.
- Das SystemTap-Skript wird mit dem Bash-Skript `gendata.sh` aufgerufen.
- Ein R-Project-Analyseprogramm `entropy_pool_distribution.r` erstellt die folgenden Graphiken aus der mittels dem SystemTap-Skript erzeugten Datenreihe.

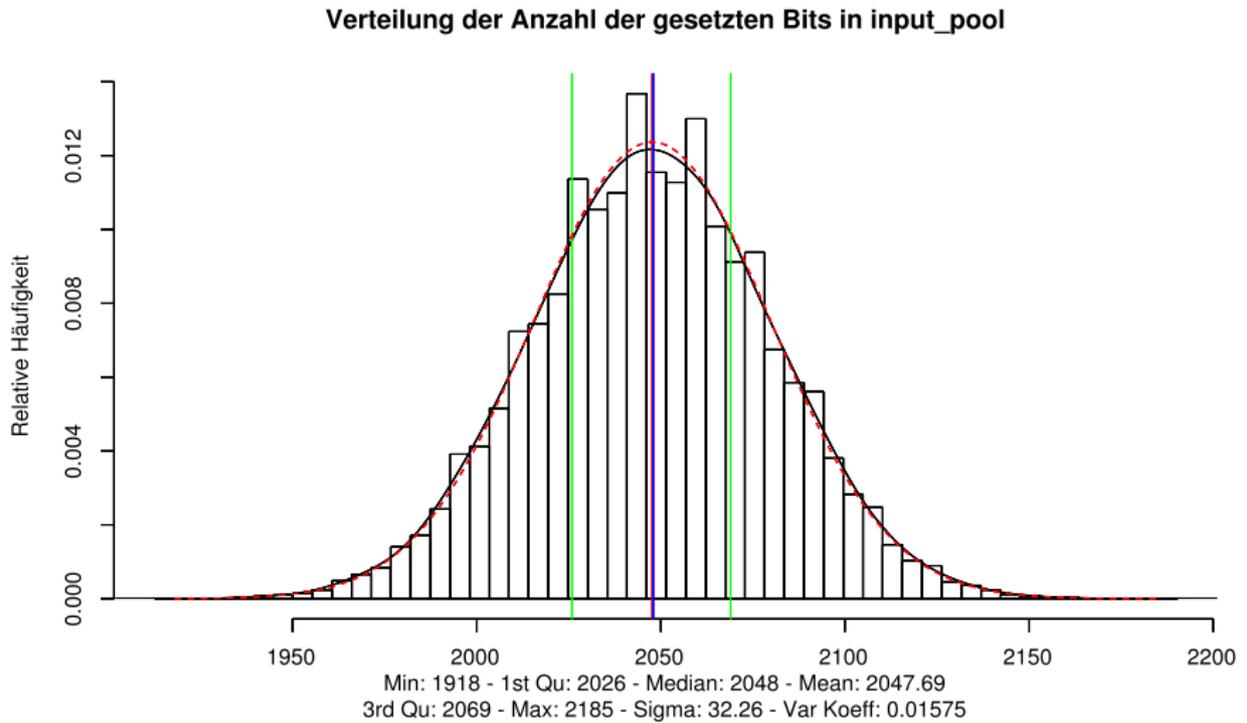
Die gewählte Stichprobengröße beträgt $S=100.000$.

5.2.2 Test 1 - Gesetzte Bits im Entropie-Pool `input_pool`

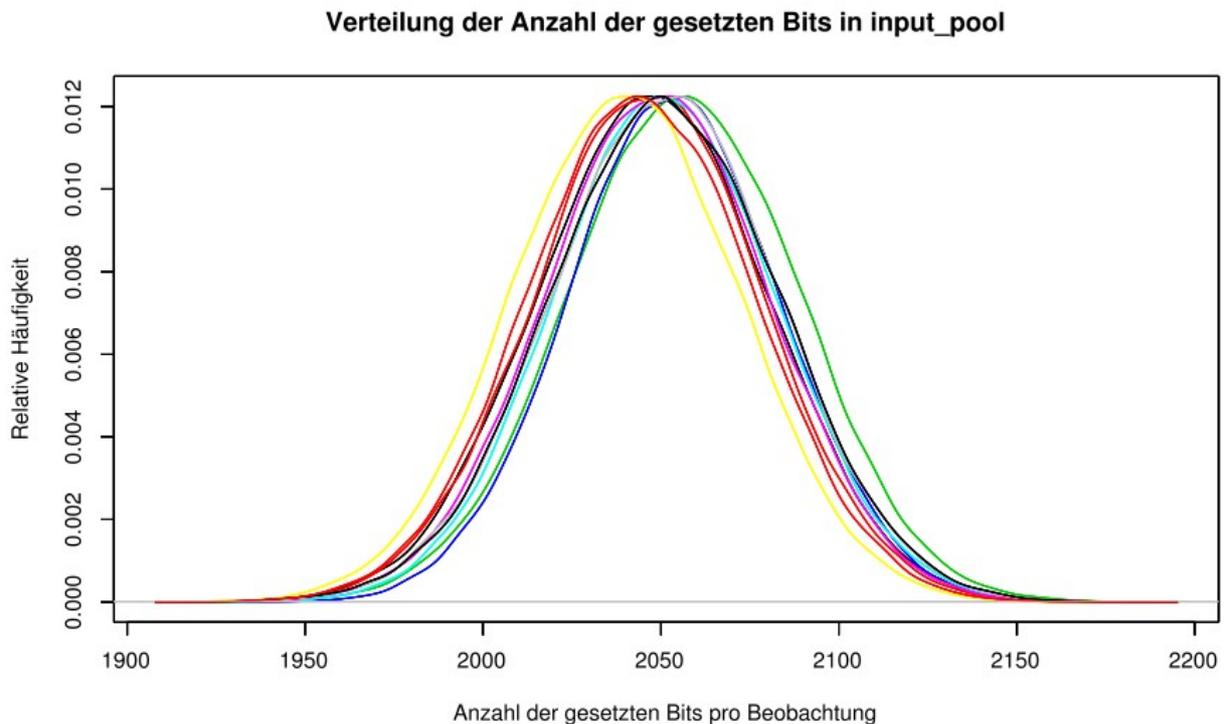
5.2.2.1 Graphische Veranschaulichung der Testresultate und Interpretation der Ergebnisse

Verteilung der Anzahl der gesetzten Bits in `input_pool`





Das Verhalten der Verteilung über die Zeit wurde mit 10 Testreihen mit der oben angegebenen Stichprobengröße durchgeführt. Die schwarze Kurve entspricht der ersten Testreihe, die obigem Box-Whisker-Plot und dem zugehörigen Histogramm zugrunde liegt.



Die Verteilungen zeigen die erwarteten Resultate:

- Mittelwert und Median sind fast identisch und liegen bei der Hälfte der Entropie-Pool-Größe in Bits.

- Das Histogramm zeigt eine fast perfekte Normalverteilung.
- Die Streuung der Werte um den Mittelwert ist relativ gering, wie im Histogramm und im Box-Whisker-Plot ersichtlich.
- Die Veränderung der Verteilung beim Wiederholen des Tests ist ebenfalls sehr gering.
- Der Variationskoeffizient zeigt, dass die Verteilung sehr „schmal“ im Verhältnis der Gesamtdaten ist.

Die Ergebnisse stehen nicht im Widerspruch zu der Vermutung, dass die Daten im `input_pool` gleichverteilt sind.

5.2.3 Test 1 - Gesetzte Bits im Entropie-Pool `nonblocking_pool`

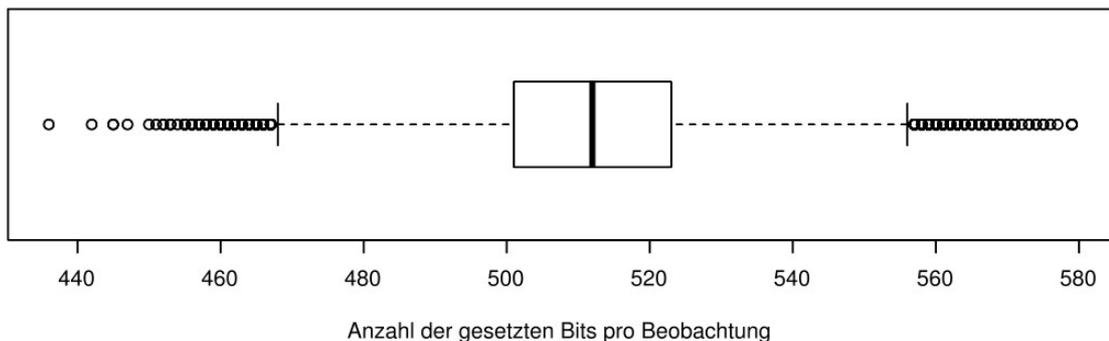
Wir erinnern daran, dass neue Werte für den `nonblocking_pool` aus folgenden Quellen stammen können:

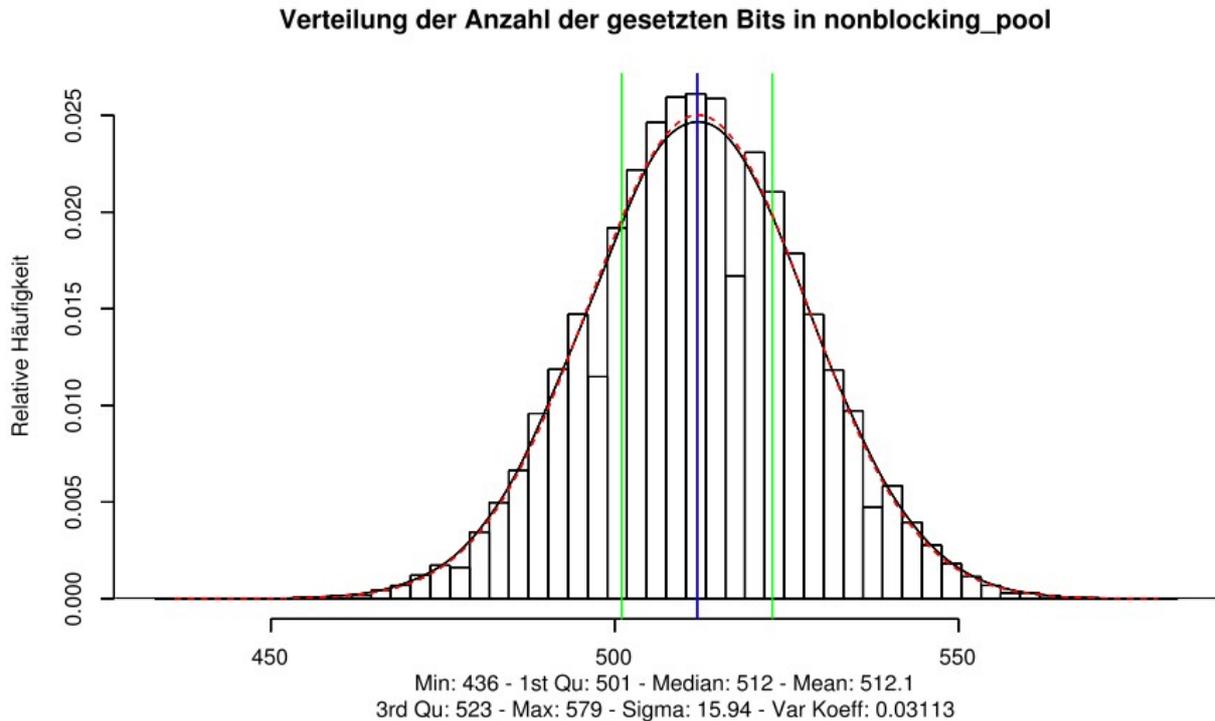
- Übertrag von Daten aus dem `input_pool`, welcher beim Lesen aus `/dev/urandom` angestoßen wird - siehe Abschnitt 2.6, Schritt 1.
- Hinzufügen des SHA-1 Wertes über den gesamten `nonblocking_pool` welcher beim Lesen aus `/dev/urandom` erzeugt wird - siehe Abschnitt 2.6, Schritt 3.iii).
- Hinzufügen von Daten aus dem User-Space, wenn ein Prozess Daten in die Gerätedateien schreibt - siehe Abschnitt 2.5.1.8.

Da das Hinzufügen von erhaltenen Daten unabhängig von der Quelle erfolgt, wurde auf eine gesonderte Betrachtung von ausschließlich Lese- oder Schreibvorgängen auf den Gerätedateien verzichtet.

5.2.3.1 Graphische Veranschaulichung der Testresultate und Interpretation der Ergebnisse

Verteilung der Anzahl der gesetzten Bits in `nonblocking_pool`





Die Verteilungen zeigen die erwarteten Resultate:

- Der Mittelwert und der Median sind fast identisch und liegen bei der Hälfte der Entropie-Pool-Größe in Bits.
- Das Histogramm zeigt eine fast perfekte Normalverteilung. Dennoch sind einige signifikante Einbrüche in den Säulen zu beobachten. Wenn die Säulenbreite aber leicht verändert wird (sowohl bei Vergrößerung, als auch bei Verkleinerung), dann verschwinden diese Einbrüche fast vollständig. Demzufolge ist davon auszugehen, dass die gewählte Säulenzahl die Beobachtungen gerade ungünstig unterteilt. Zu beachten ist, dass die Säulenzahl für alle Graphen gleich groß ist, um eine Vergleichbarkeit herzustellen.
- Die Streuung der Werte um den Mittelwert ist relativ gering, wie im Histogramm und im Box-Whisker-Plot ersichtlich.
- Der Variationskoeffizient zeigt, dass die Verteilung sehr „schmal“ im Verhältnis der Gesamtdaten ist.

Die Ergebnisse widersprechen der Vermutung, dass die Daten im `nonblocking_pool` gleichverteilt sind, nicht.

5.2.4 Test 1 - Gesetzte Bits im Entropie-Pool `blocking_pool`

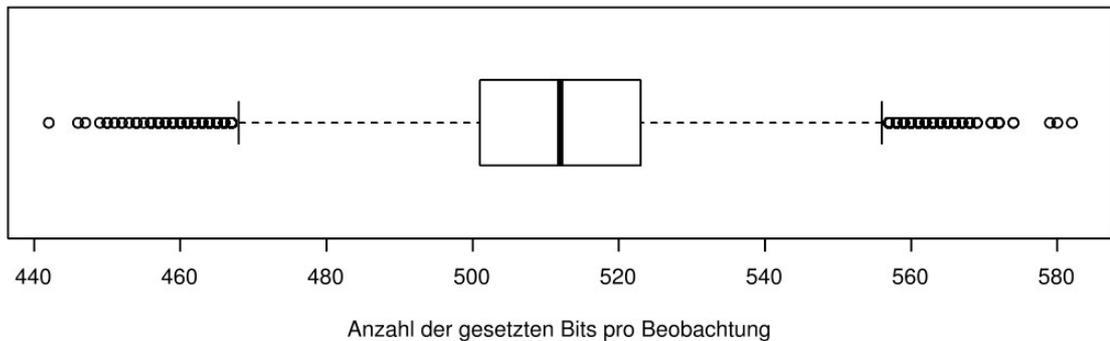
Wir erinnern daran, dass neue Werte für den `blocking_pool` aus folgenden Quellen stammen können:

- Übertrag von Daten aus dem `input_pool`, welcher beim Lesen aus `/dev/random` angestoßen wird – siehe Abschnitt 2.6, Schritt 1.
- Hinzufügen des SHA-1-Wertes über den gesamten `blocking_pool` welcher beim Lesen aus `/dev/random` erzeugt wird – siehe Abschnitt 2.6, Schritt 3.iii).
- Hinzufügen von Daten aus dem User-Space, wenn ein Prozess Daten die Gerätedateien schreibt – siehe Abschnitt 2.5.1.8.

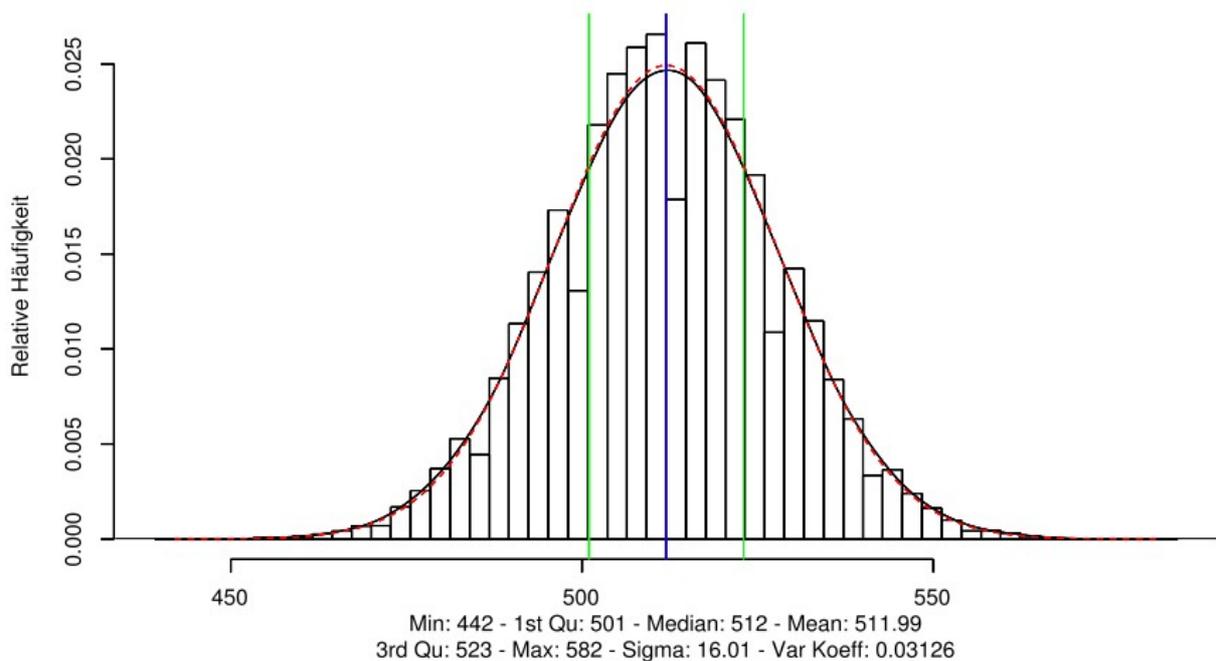
Wie zuvor wurde auf eine gesonderte Betrachtung von ausschließlich Lese- oder Schreibvorgängen auf den Gerätedateien verzichtet.

5.2.4.1 Graphische Veranschaulichung der Testresultate und Interpretation der Ergebnisse

Verteilung der Anzahl der gesetzten Bits in blocking_pool



Verteilung der Anzahl der gesetzten Bits in blocking_pool



Die Verteilungen zeigen die erwarteten Resultate:

- Der Mittelwert und der Median sind fast identisch und liegen bei der Hälfte der Entropie-Pool-Größe in Bits.
- Das Histogramm zeigt eine fast perfekte Normalverteilung.
- Die Streuung der Werte um den Mittelwert ist relativ gering, wie im Histogramm und im Box-Whisker-Plot ersichtlich.
- Der Variationskoeffizient zeigt, dass die Verteilung sehr „schmal“ im Verhältnis der Gesamtdaten ist.

Die Ergebnisse stehen nicht im Widerspruch zur Vermutung, dass die Daten im blocking_pool gleichverteilt sind.

5.2.5 Test 1 zum Systemstart - Vorüberlegungen und Testansatz

5.2.5.1 Vorüberlegungen

Der LRNG bezieht seine Entropie aus Hardware-Ereignissen und deren zeitlichem Eintreffen. Der LRNG geht davon aus, dass das Eintreffen der Hardware-Ereignisse stochastisch unabhängig ist. Diese stochastische Unabhängigkeit muss aber bei dem Startvorgang des Betriebssystems in Zweifel gezogen werden, da:

- das Betriebssystem mit einer fest programmierten Startprozedur initialisiert wird,
- beim Starten üblicherweise ausschließlich Hardware-Ereignisse der Festplattenaktivität gemessen werden, und
- die fest programmierte Startprozedur impliziert, dass die Festplatte in genau vordefinierten Zeitintervallen von genau gleichen Sektoren der Festplatte lesen muss.

Es ist auf Basis dieser Überlegung anzunehmen, dass sich die Entropie-Pools beim Starten ähneln, nicht aber, dass sie gleich sind. Der Grund hierfür ist die hohe Auflösung des Zeitgebers, die Abweichungen beim Ausführen von Code aufgrund von thermischen Schwankungen in der CPU sichtbar macht. Weiterhin ist die Position der Platten und Leseköpfe beim ersten Festplattenzugriff nicht vorhersagbar, wodurch es eine Varianz bei dem ersten Festplattenzugriff gibt.

5.2.5.2 Testansatz

Da in diesem Test die Qualität der Entropie-Pools während des Startvorgangs beobachtet werden soll, wird der Test beendet, wenn der Seed, der beim letzten Herunterfahren gespeichert wurde, in die Entropie-Pools `blocking_pool` und `nonblocking_pool` gemischt wird. Es wird erwartet, dass dieser Test die Notwendigkeit dieser Praxis bestätigt. Wie in dem Überblick bereits beschrieben, werden die Messungen einmal für die gesamten Entropie-Pools diskutiert und einmal für die einzelnen Wörter der Pools.

Die Messungen aller Entropie-Pools für die Testreihe werden zu folgenden zwei Zeitpunkten genommen:

- Nachdem die Initialisierung des Kerns beendet ist und die Abarbeitung der gegebenenfalls vorhandenen `initramfs` beendet wird. Die Messung wird durchgeführt noch bevor `/sbin/init` gestartet wird.
- Unmittelbar vor dem Hinzufügen des Seeds in die beiden Pools, `blocking_pool` und `nonblocking_pool`, wird eine weitere Messung durchgeführt.

Bei der ersten Messung sollten so gut wie keine Hardware-Ereignisse in den `input_pool` eingefügt werden:

- Kein Auftreten von Ereignissen aus Eingabegeräten.
- Das Laden der Datei mit dem Kern und der `initramfs` von der Festplatte wird vom Bootloader durchgeführt. Damit tritt diese Festplattenaktivität noch vor der Initialisierung des LRNG auf und wird nicht als Ereignis in den `input_pool` aufgenommen.
- Es werden gegebenenfalls einige Festplatten-Ereignisse durch die `initramfs` bedingt, wenn die Platten auf Vorhandensein geprüft werden.
- Weder das Mounten der `initramfs` noch das Laden von Programmen aus der `initramfs` erzeugen Ereignisse für den LRNG, da die `initramfs` im Speicher liegt.
- Das Mounten der `root`-Partition erzeugt einige Festplatten-Ereignisse.
- Das Laden des `SystemTap`-Kernel-Moduls, des Hilfs-Shell-Skripts, der Programme `staprun` und `stapio` erzeugen einige Festplattenaktivität, welche vom LRNG genutzt wird.

Auch die Pools `blocking_pool` und `nonblocking_pool` sollten kaum Änderungen ihrer Initialwerte haben, da keine User-Space-Programme Zufallszahlen angefordert haben. Kerninterne Anforderungen für Zufallszahlen aus dem `nonblocking_pool` sollten

stattgefunden haben, zum Beispiel die Generierung von Initial-Sequence-Numbers für TCP oder UUIDs.

Bei der zweiten Messung sind bereits sehr viele Festplattenereignisse eingetreten. Abschnitt 6.1.2 zeigt diese Aktivität sehr klar, da die Entropieschätzung vom `input_pool` während der ersten 60 Sekunden des Startens des User-Space stark schwankt.

Die Analyse der Testresultate erfolgt zu jedem Messzeitpunkt für jeden Entropie-Pool. Damit gibt es insgesamt sechs Analysen für die drei Entropie-Pools, welche zu 2 verschiedenen Zeitpunkten durchgeführt wurden.

Wie bereits beschrieben, ist jede Analyse zweigeteilt:

- Analog zu 5.2.2, 5.2.3, und 5.2.4 wird jeweils die Anzahl der gesetzten Bits des gesamten Entropie-Pools betrachtet und graphisch veranschaulicht.
- Im zweiten Teil wird für jeden Pool die Verteilung der Anzahl der gesetzten Bits pro 32-Bit-Wort betrachtet. Es wird also jedes Wort des Entropie-Pools über die gesamte Stichprobe separat von den anderen Wörtern betrachtet. Selbstverständlich erhält man auf diese Weise zu jedem Beobachtungszeitpunkt nicht eine Bit-Summe, sondern für jedes Wort des Entropie-Pools eine Bit-Summe. Auch hier werden die Messergebnisse graphisch veranschaulicht.

Des Weiteren wird die normalisierte Standardabweichung für jeden dieser Werte eines

$$\sigma_{norm} = \frac{|(X - M_{Wort})|}{\sigma_{Wort}}$$

Worts wie folgt berechnet

wobei X die Anzahl der gesetzten Bits des betrachteten Worts ist, M_{Wort} der Mittelwert der Bitwerte und σ_{Wort} die Standardabweichung dieser Werte ist. Den Grund hierfür liefern die angenommenen Ausreißer, die durch die Standardabweichung besser sichtbar sein sollen.

Wenn die Ausprägungen der Wörter im Entropie-Pool einer Gleichverteilung folgen, dann müssen die gemessenen Werte für die Anzahl der gesetzten Bits einer Normalverteilung folgen, welche sich um den Mittelwert M_{Wort} zentriert. Dies bedeutet, dass eine normierte Standardabweichung von größer 2 zu einer näheren Betrachtung der Werte führen sollte – bei einer Normalverteilung tritt eine Standardabweichung von größer 2 nur in 4% der Fälle auf.

Die Skripte und Dateien zu diesem Test sind im Verzeichnis `corr_pools_boot` zu finden, und wie folgt gegliedert:

- Das SystemTap-Skript `corr_pools_boot.stp` wurde erstellt. Die Anzahl der getesteten Startvorgänge kann mit der Variable `num_samplings` angepasst werden. Das Skript misst sofort²¹ die drei Entropie-Pools und schreibt die Werte wortweise aus. Weiterhin liest es die Pools aus, wenn Daten entweder über `/dev/random` oder `/dev/urandom` geschrieben werden. Wiederum werden alle drei Pools gelesen und deren Inhalt wortweise angezeigt.
- Es wurde ein Bash-Skript erzeugt, das das SystemTap-Kernel-Modul lädt. Dieses Skript ist ein Ersatz für `/sbin/init`.
- Ein weiteres Bash-Skript speichert die erzeugten Daten auf die Festplatte, nachdem der Test beendet wurde und initiiert dann den Neustart des Systems.
- Ein `upstart`-Job wird bereitgestellt, welcher das zweite Bash-Skript am Ende der User-Space-Startprozedur aufruft.
- Die Testumgebung wird mit dem Bash-Skript `gendata.sh` initialisiert.

²¹ Die Implementation des Skripts misst die Entropie-Pools zum Zeitpunkt, zu dem ein neues Ereignis auftritt. Das Skript stellt sicher, dass die Pools noch vor dem Hinzufügen des Ereignisses zum entsprechenden Pool ausgelesen werden.

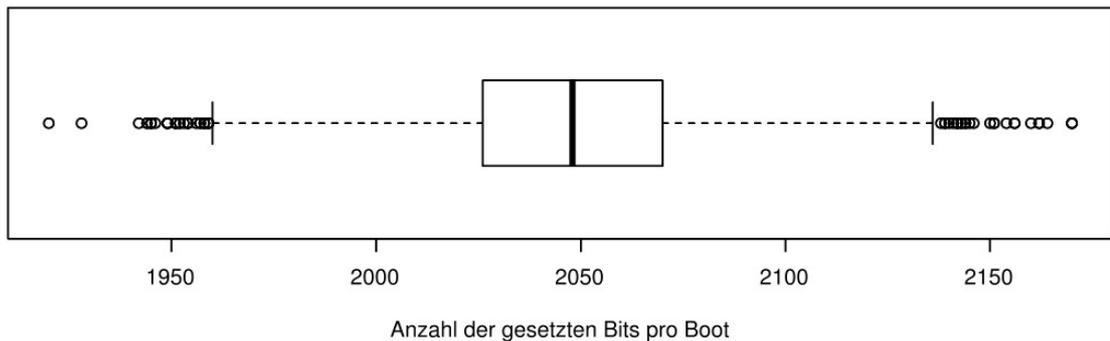
- Das R-Project Programm `corr_pools_boot.r` liest alle Testreihen in eine Matrix ein, um Analysen durchzuführen.

Als Stichprobenzahl wird $S=10.000$ gewählt.

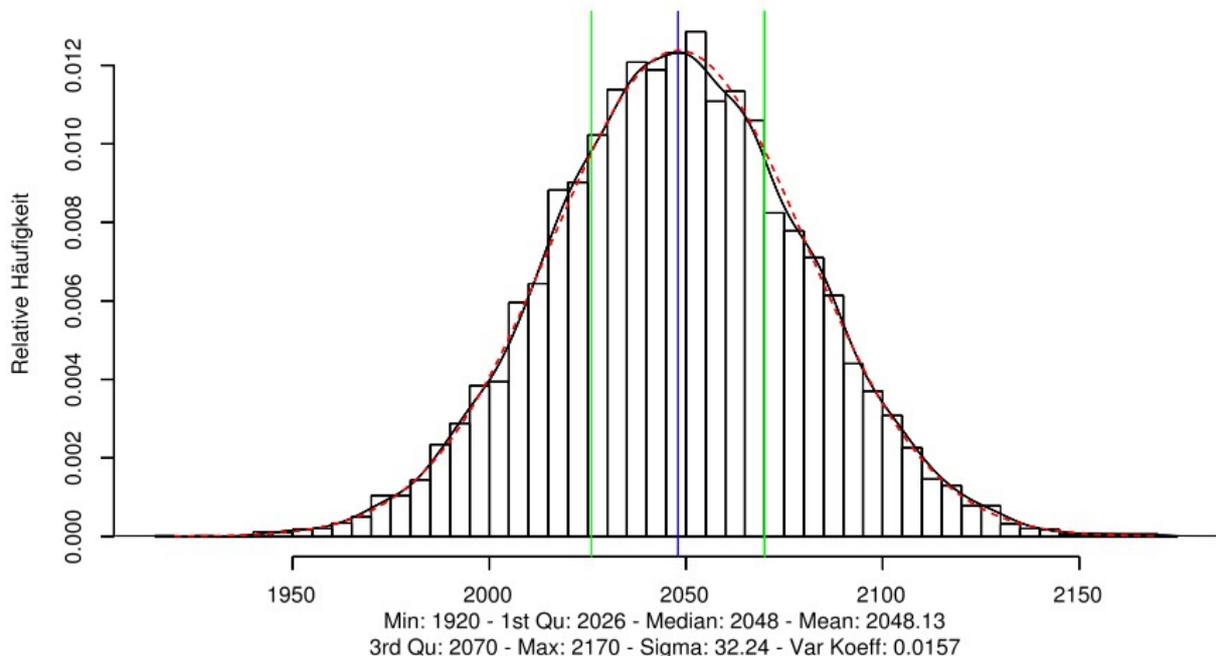
5.2.6 Test 1 - Gesetzte Bits im Entropie-Pool `input_pool` zum Systemstart

5.2.6.1 Graphische Veranschaulichung der Testresultate und Interpretation der Ergebnisse vor Start des User-Space

Verteilung der Anzahl der gesetzten Bits in `input_pool`



Verteilung der Anzahl der gesetzten Bits in `input_pool`

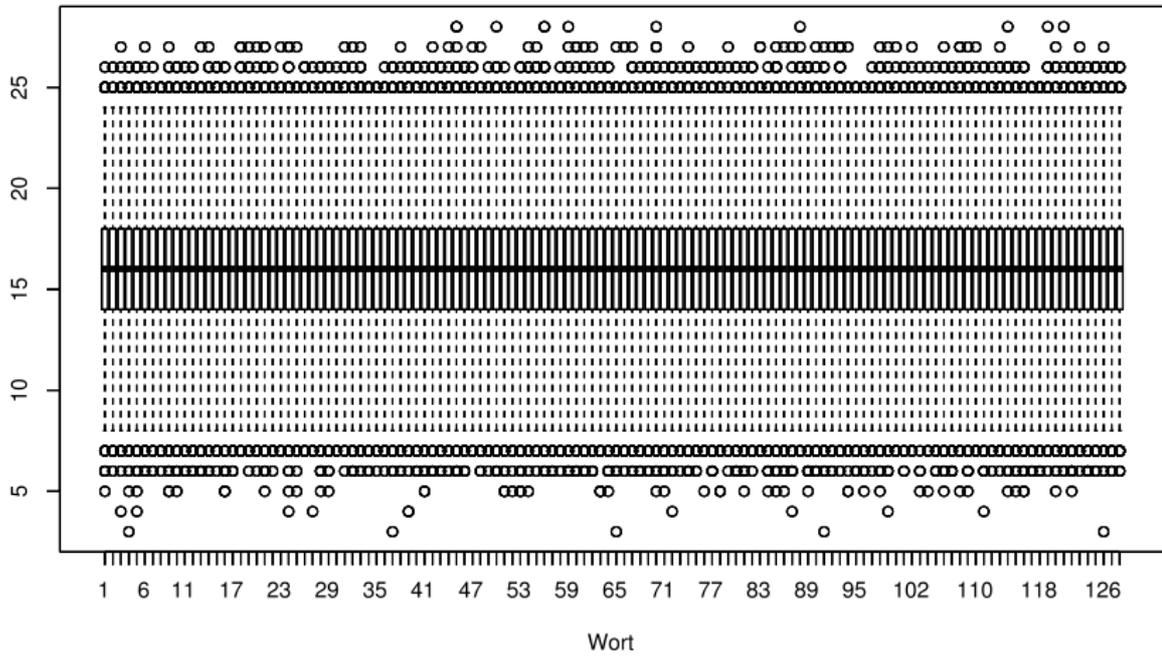


Für den zweiten Teil der Analyse, der Betrachtung der Bit-Summen für jedes einzelne Wort, erinnern wir daran, dass der `input_pool` aus

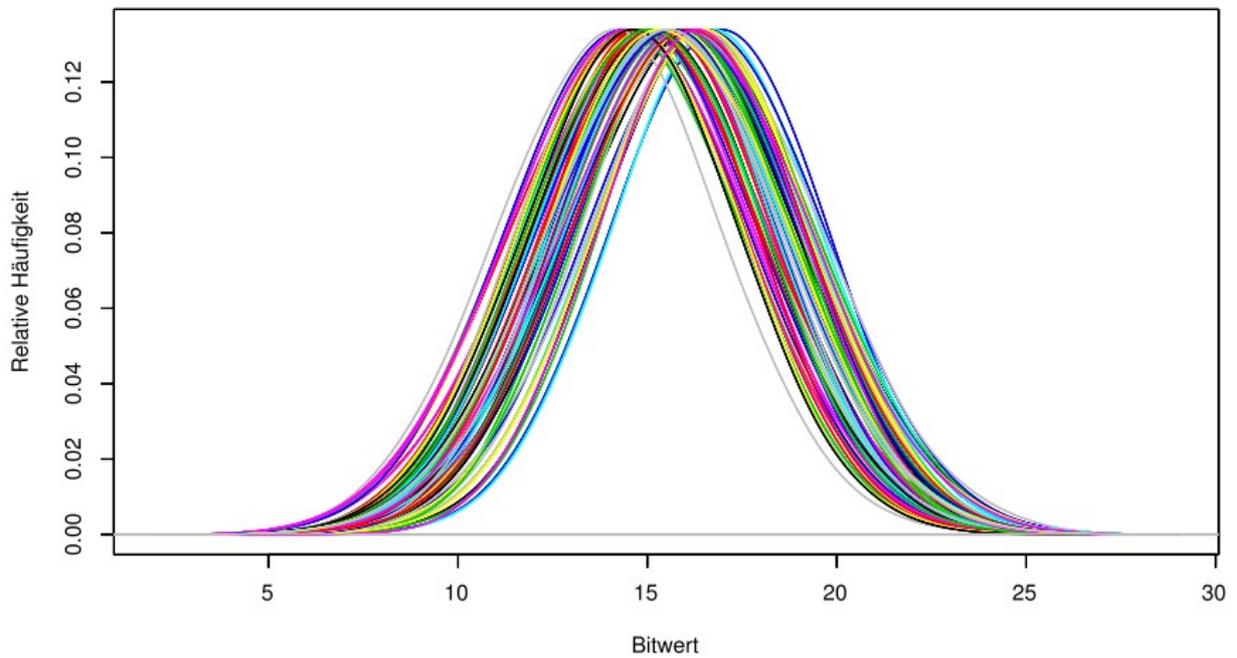
`INPUT_POOL_WORDS = 128`

32-Bit-Wörtern besteht (siehe Abschnitt 2.3). Deshalb erhält man hier in 128 Messreihen.

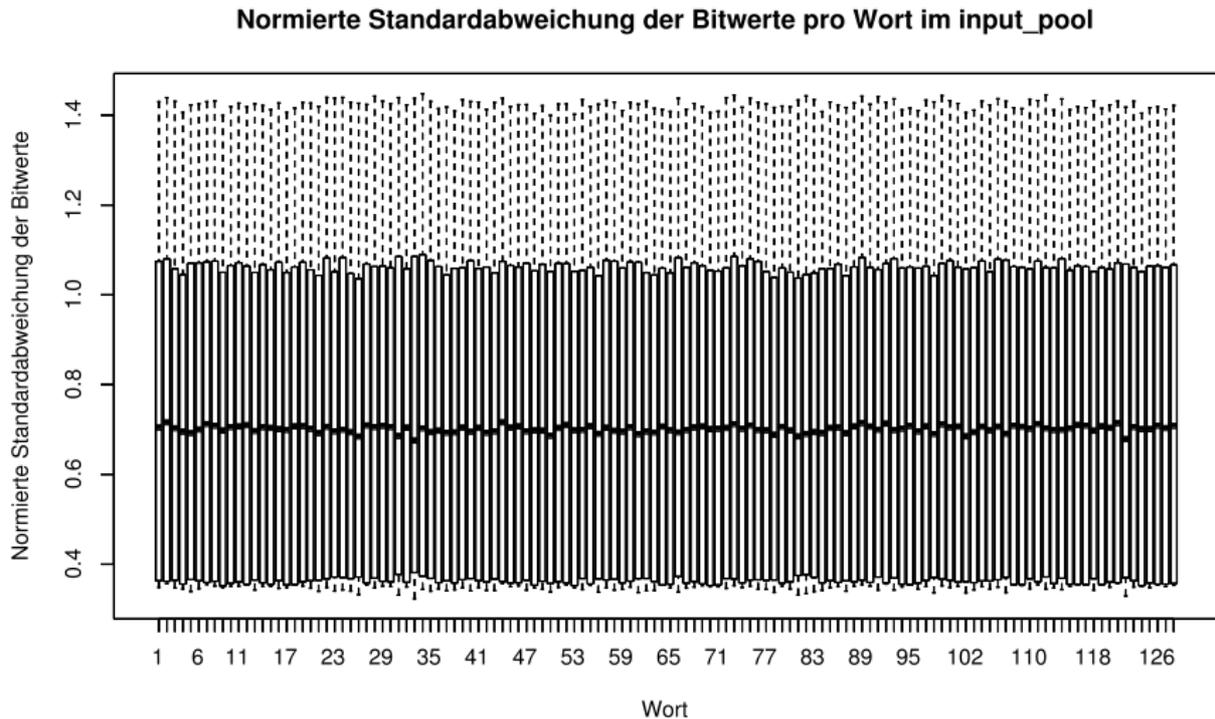
Verteilung der Anzahl der gesetzten Bits pro Wort in input_pool



Verteilung der Bitsummen pro Wort in input_pool



Abschließend folgt ein weiterer Box-Whisker-Plot zur Darstellung der normierten Standardabweichung der Bit-Summen pro Wort.



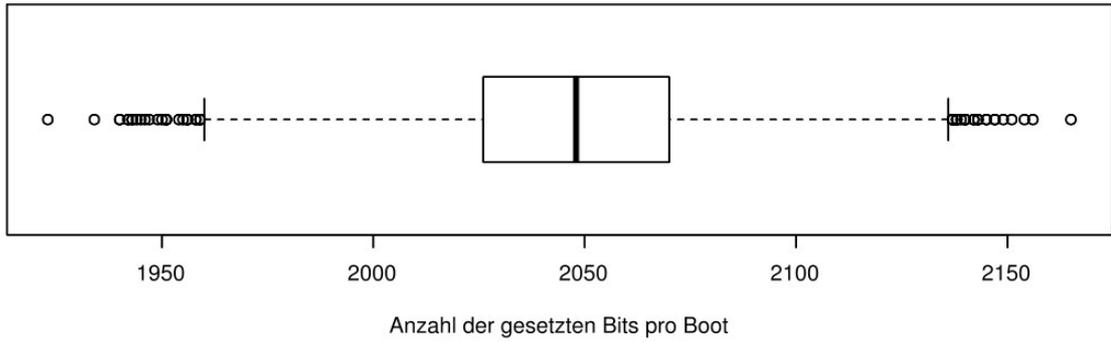
Die Verteilung der Bit-Summen des Entropie-Pools zeigt fast eine identische Verteilung wie in der Analyse in Abschnitt 5.2.2. Eine Ausnahme gibt es aber: das Histogramm zeigt einige „Löcher“ in denen die Beobachtungen für einen kleinen Abschnitt einbrechen. Dennoch sind diese Löcher nicht in der Dichte sichtbar. **Obwohl die Ergebnisse für die Dichte nicht im Widerspruch zu der Annahme, dass die Daten in input_pool gleichverteilt sind, stehen, gibt das Histogramm Hinweise darauf, dass die Daten im input_pool doch nicht gleichverteilt sind.**

Bei der wortweisen Analyse der Entropie-Pool sind ebenfalls Normalverteilungen zu sehen, welche um den Mittelwert/Median schwanken. Dabei zeigt aber der Graph mit den übereinander gelegten wortweisen Dichteverteilungen eine erheblich stärkere Streuung als die Wiederholung der Beobachtungen des gesamten Entropie-Pools. **Dies lässt den Schluss zu, dass die Verteilung der Bit-Summen in den Wörtern zwar einer Normalverteilung ähneln, aber die Verteilung der Bits innerhalb der einzelnen Wörter eine größere Abweichung von einer Gleichverteilung aufweisen.** Diese Abweichungen von der Gleichverteilung bewegen sich jedoch noch in einem vollkommen unkritischen Bereich (siehe dazu auch die Vorbemerkungen in Abschnitt 5.1.

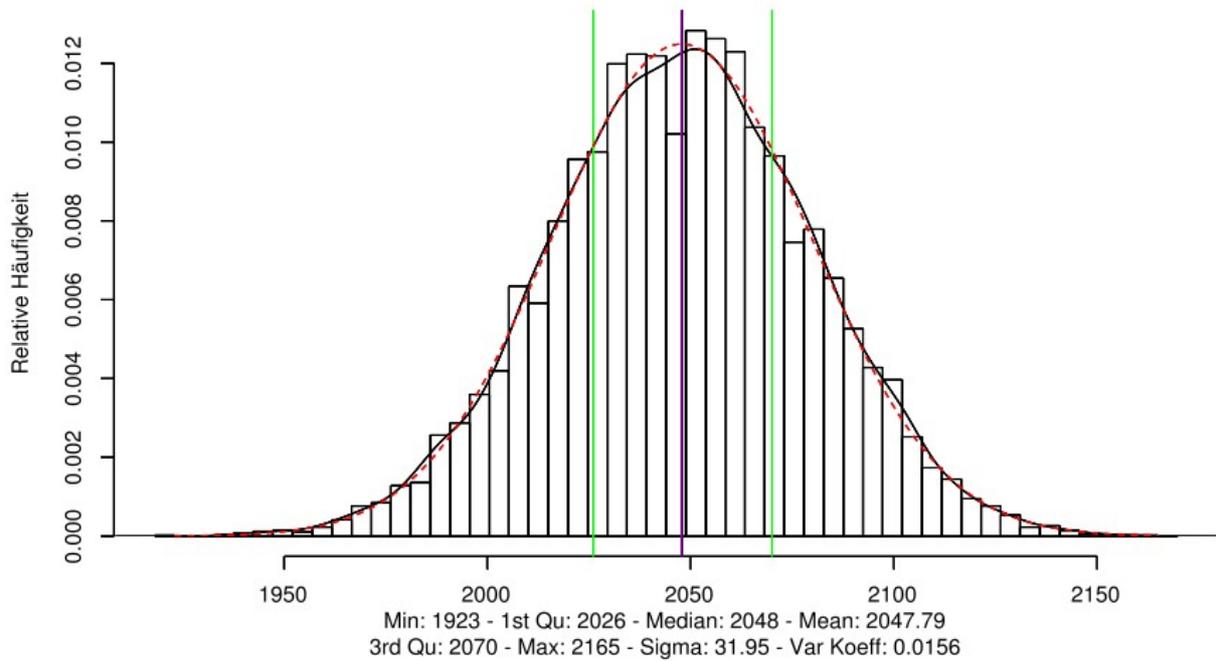
Der Graph mit der normierten Standardabweichung zeigt eine breite, aber gleichmäßige Streuung der Werte.

5.2.6.2 Graphische Veranschaulichung der Testresultate und Interpretation der Ergebnisse vor Hinzufügen des Seeds

Verteilung der Anzahl der gesetzten Bits in input_pool

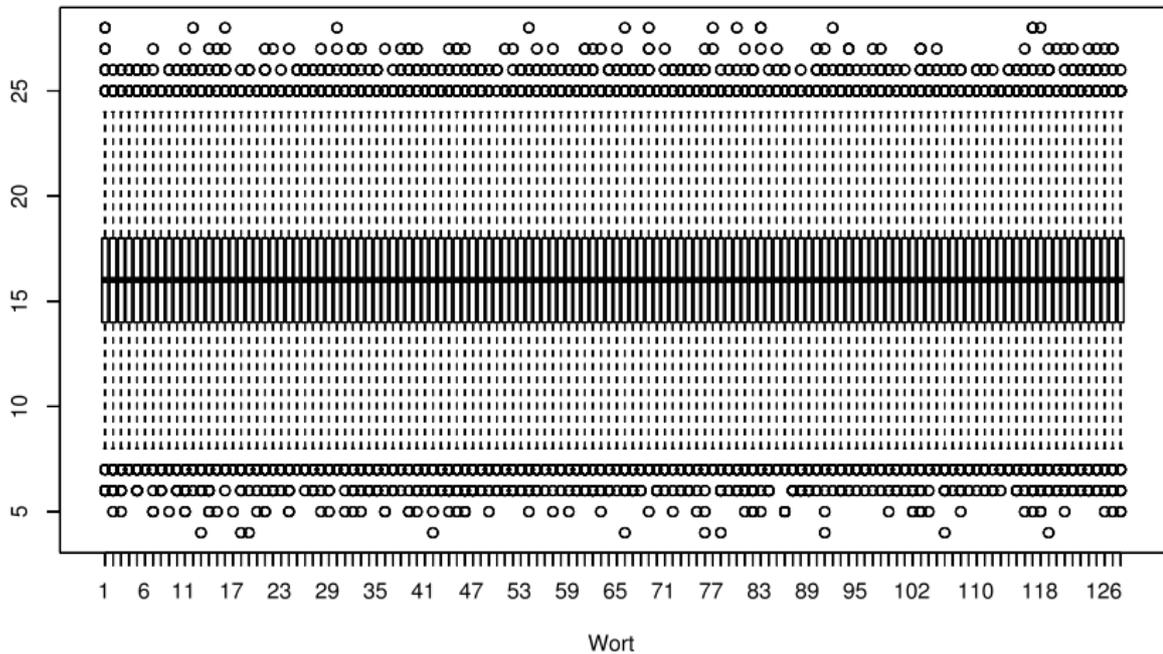


Verteilung der Anzahl der gesetzten Bits in input_pool

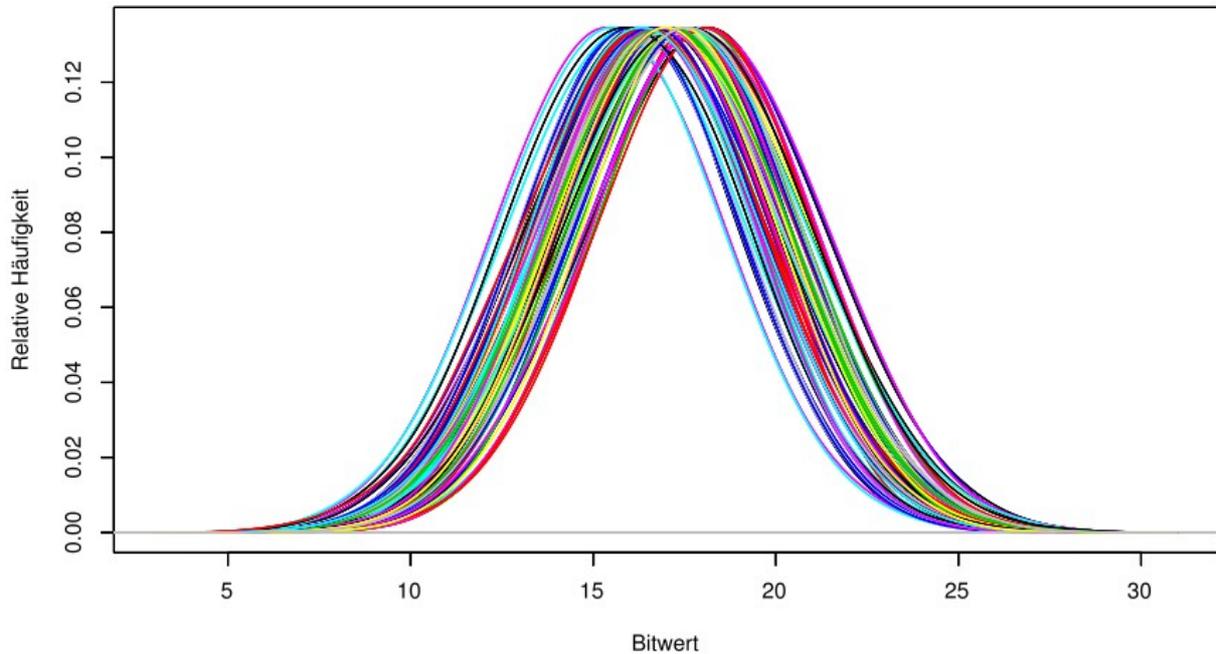


Wieder folgt jetzt die Betrachtung der Bit-Summen für jedes einzelne Wort.

Verteilung der Anzahl der gesetzten Bits pro Wort in input_pool

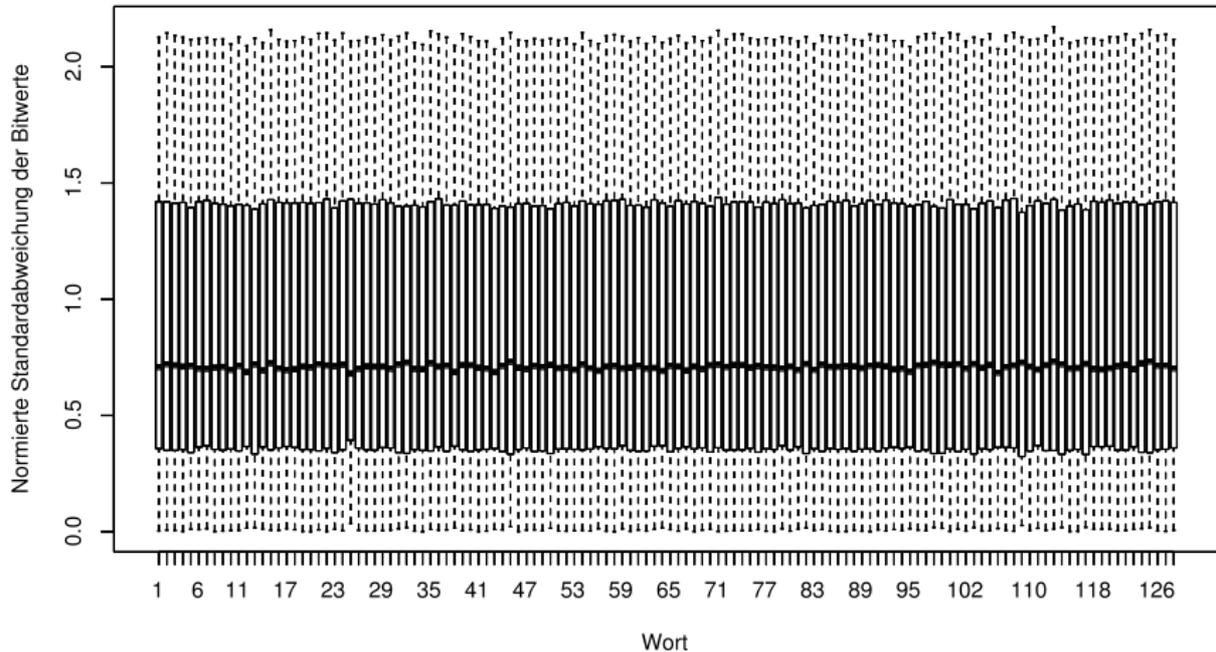


Verteilung der Bitsummen pro Wort in input_pool



Wie zuvor wird abschließend die normierte Standardabweichung der Bit-Summen pro Wort mittels eines Box-Whisker-Plots dargestellt.

Normierte Standardabweichung der Bitwerte pro Wort im input_pool



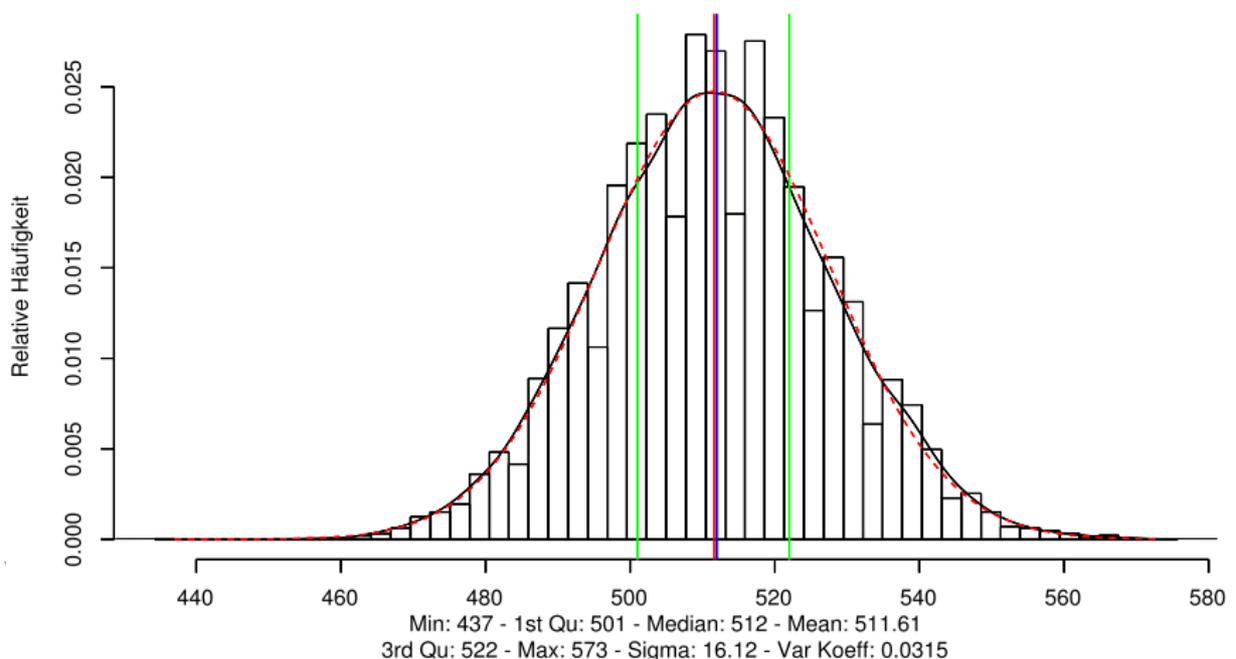
Bei der Wiederholung der Messung des Entropie-Pools kurz vor dem Einbringen des Seeds zeigen sich kaum signifikanten Änderungen gegenüber der Messung kurz nach dem Start des Kerns: Die Dichte und das Histogramm der Daten im gesamten Pool zeigen in beiden Fällen eine gute Angleichung an eine Normalverteilung. Dennoch ist die Verteilung noch nicht so gut wie bei den Beobachtungen des Entropie-Pools im laufenden System.

Die wortweise Betrachtung der Verteilung der Daten in dem Pool zeigt hingegen keine Auffälligkeiten.

5.2.7 Test 1 - Gesetzte Bits im Entropie-Pool nonblocking_pool zum Systemstart

5.2.7.1 Graphische Veranschaulichung der Testresultate und Interpretation der Ergebnisse vor Start des User-Space

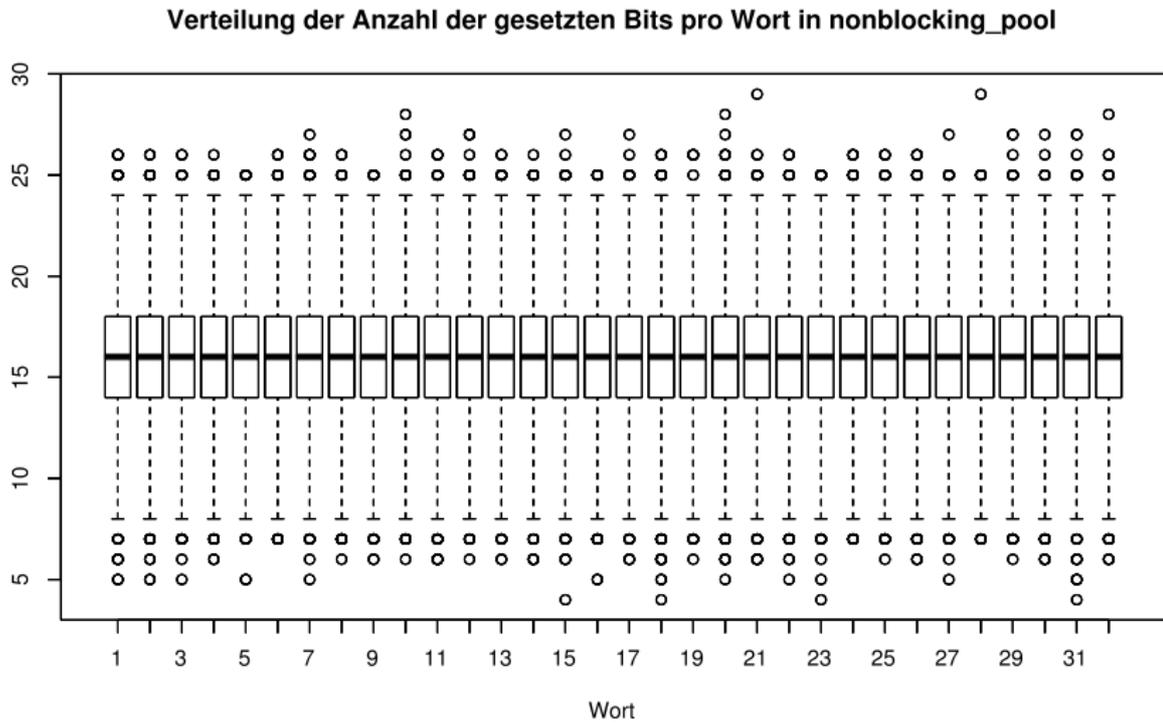
Verteilung der Anzahl der gesetzten Bits in nonblocking_pool



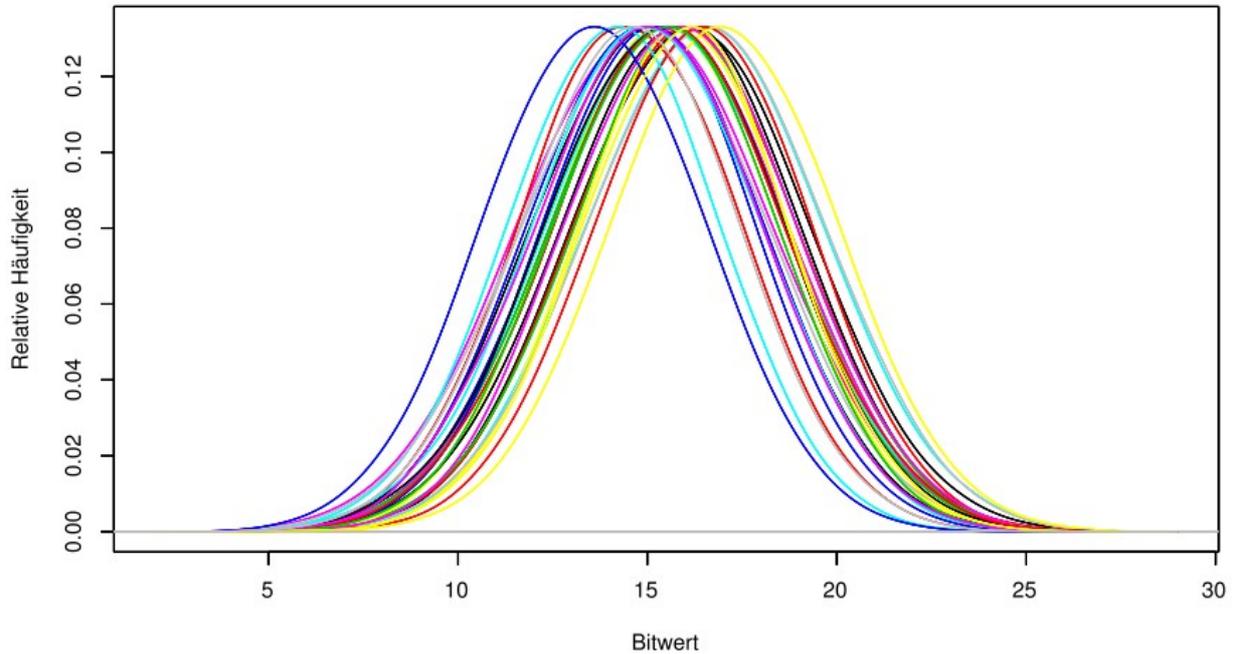
Im zweiten Teil der Analyse werden die Bit-Summen der einzelnen Wörter betrachtet. Im Gegensatz zum input_pool enthalten die beiden sekundären Pools blocking_pool und nonblocking_pool genau

OUTPUT_POOL_WORDS = 32

32-Bit-Wörter. Dadurch ergibt sich diese Anzahl an Messreihen und Graphen in den folgenden Diagrammen.

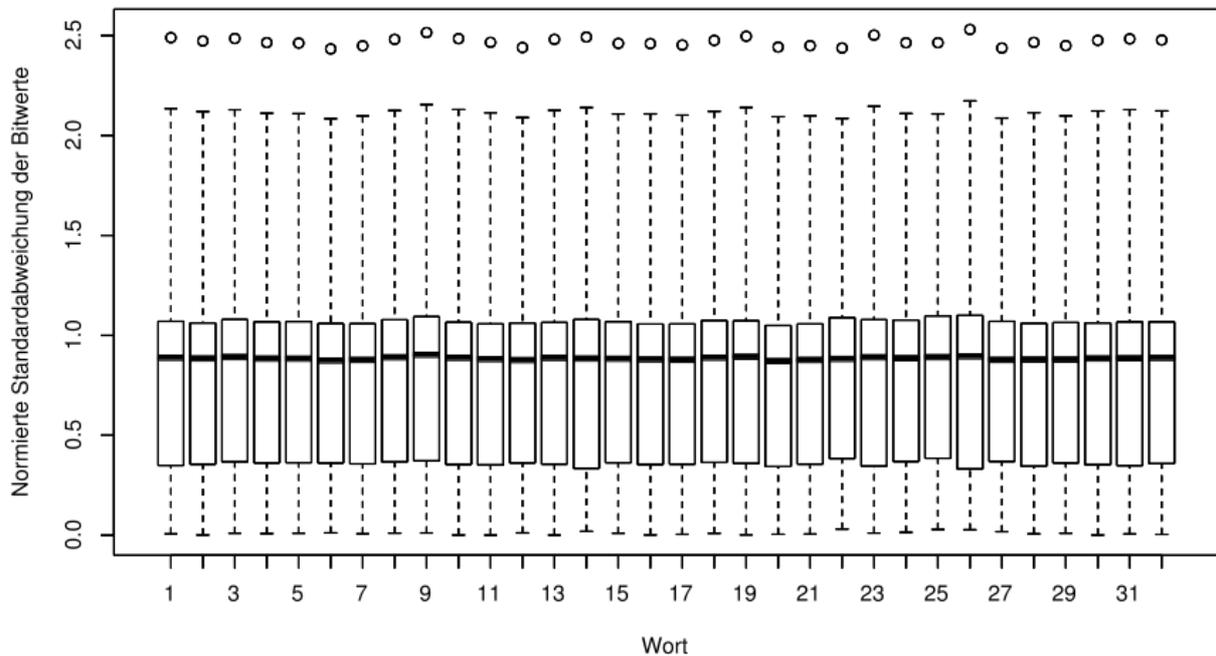


Verteilung der Bitsummen pro Wort in nonblocking_pool



Auch hier folgt noch ein Box-Whisker-Plot zur Darstellung der normierten Standardabweichung der Bit-Summen für jedes Wort.

Normierte Standardabweichung der Bitwerte pro Wort im nonblocking_pool

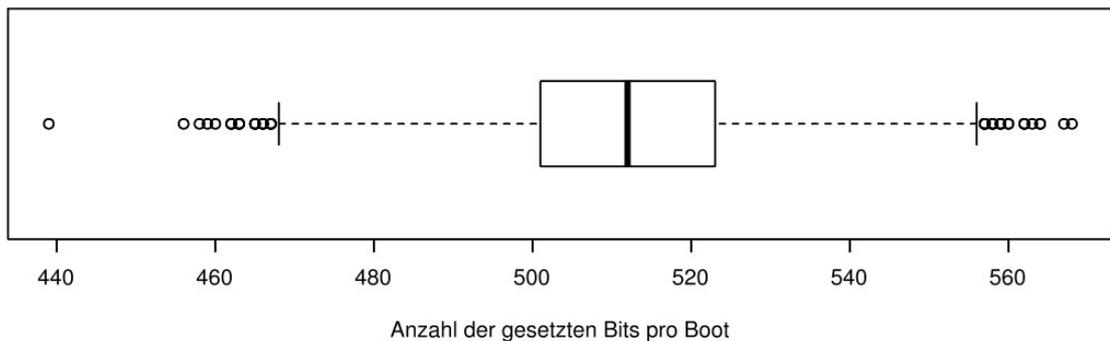


Ähnlich wie in Abschnitt 5.2.3 zeigt die Dichte der Bit-Summen des Entropie-Pools kaum Hinweise, die gegen die vermutete Gleichverteilung der Daten sprechen. Lediglich das erste Histogramm zeigt einige Auffälligkeiten, die als bedenklich eingestuft werden müssen.

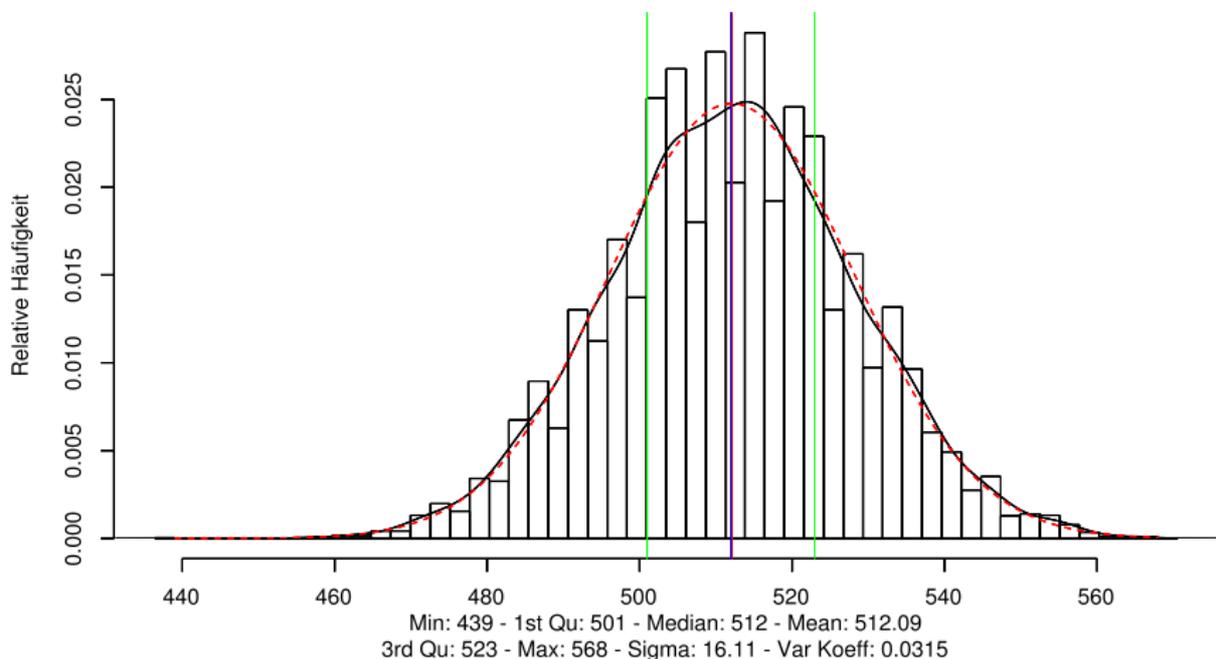
Die wortweise Analyse zeigt die erwarteten Normalverteilungen, die um den Mittelwert/Median schwanken. Die normierten Standardabweichungen der Bit-Summen zeigt eine breite, aber gleichmäßige Streuung der Werte. **Dementsprechend liefert dieser Test keinen Hinweis auf eine eventuelle Ungleichverteilung.**

5.2.7.2 Graphische Veranschaulichung der Testresultate und Interpretation der Ergebnisse vor Hinzufügen des Seeds

Verteilung der Anzahl der gesetzten Bits in nonblocking_pool

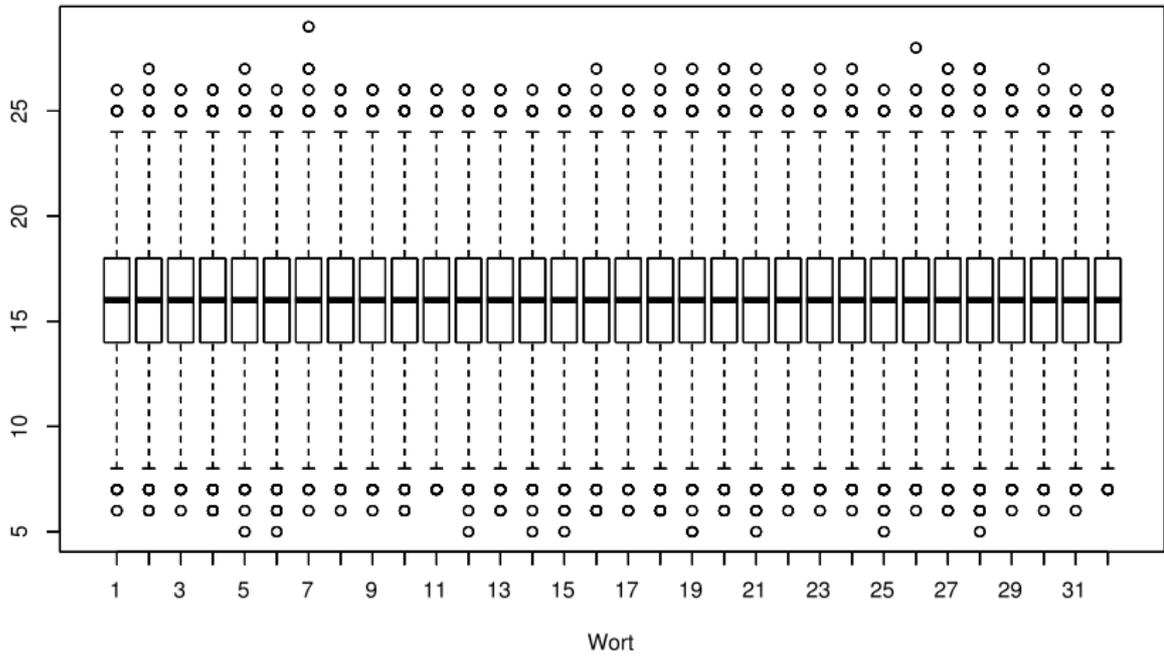


Verteilung der Anzahl der gesetzten Bits in nonblocking_pool

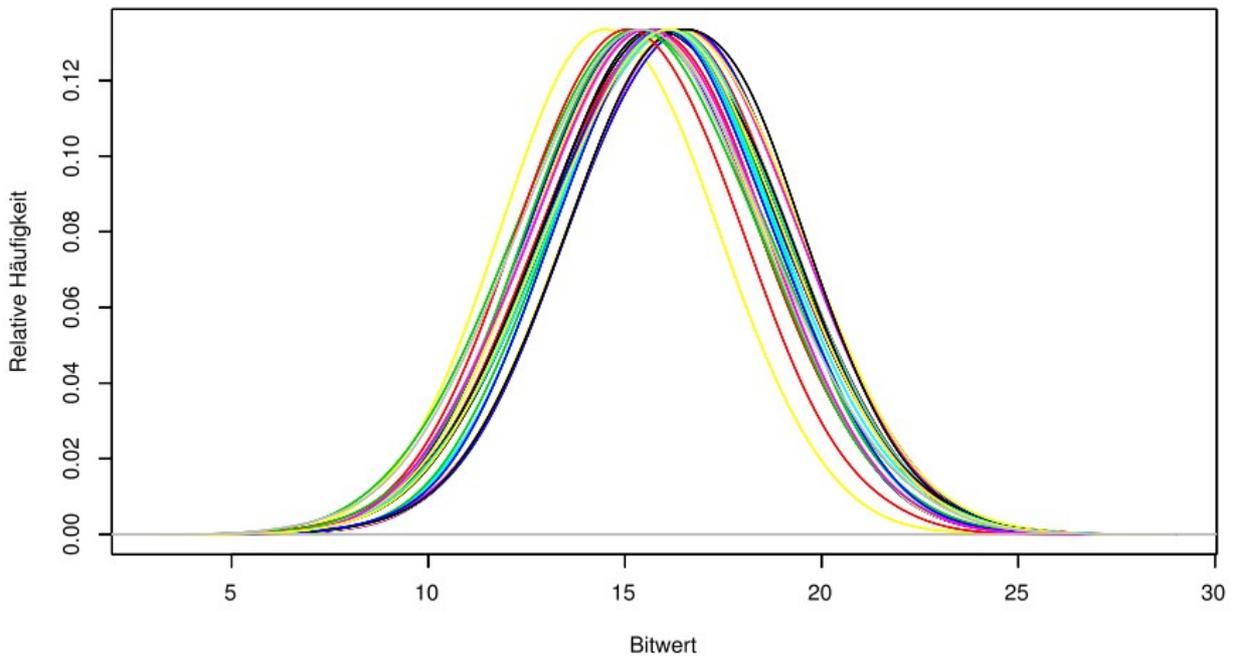


Wie zuvor schließt sich hier der zweite Teil der Analyse mit der Darstellung der Verteilungen der Anzahl der gesetzten Bits pro Wort an.

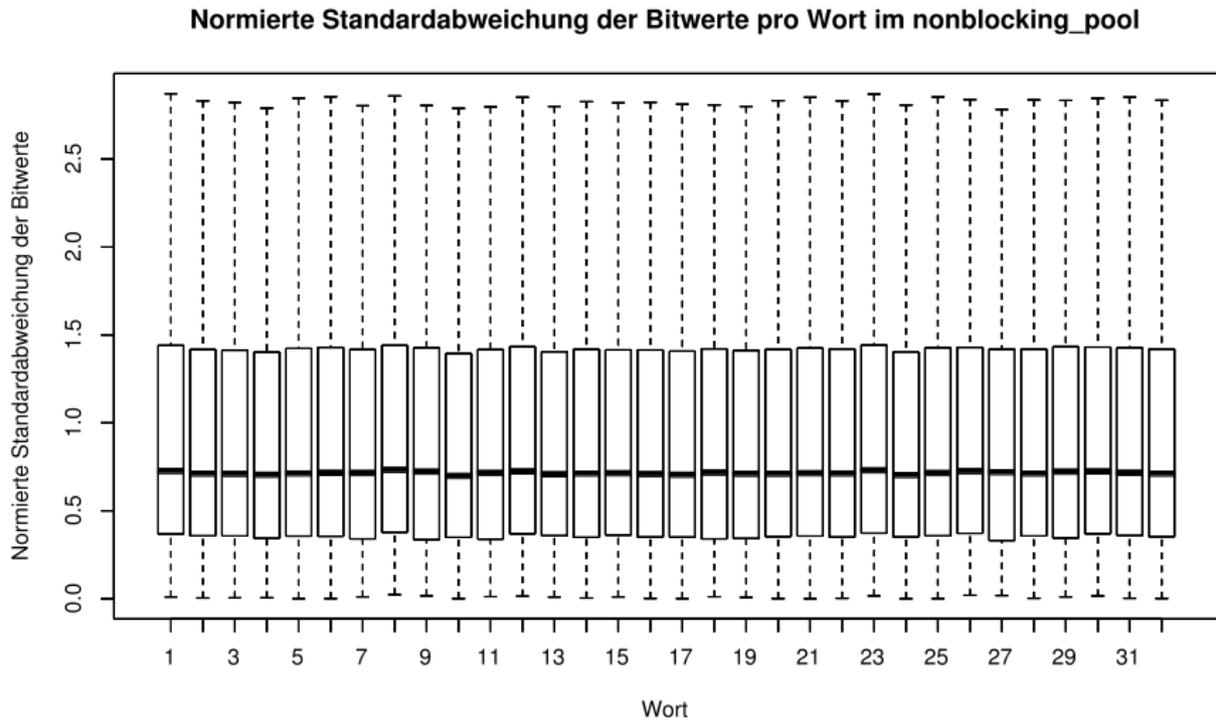
Verteilung der Anzahl der gesetzten Bits pro Wort in nonblocking_pool



Verteilung der Bitsummen pro Wort in nonblocking_pool



Der folgende Box-Whisker-Plot zeigt noch die normierten Standardabweichungen der Bit-Summen für jedes Wort.

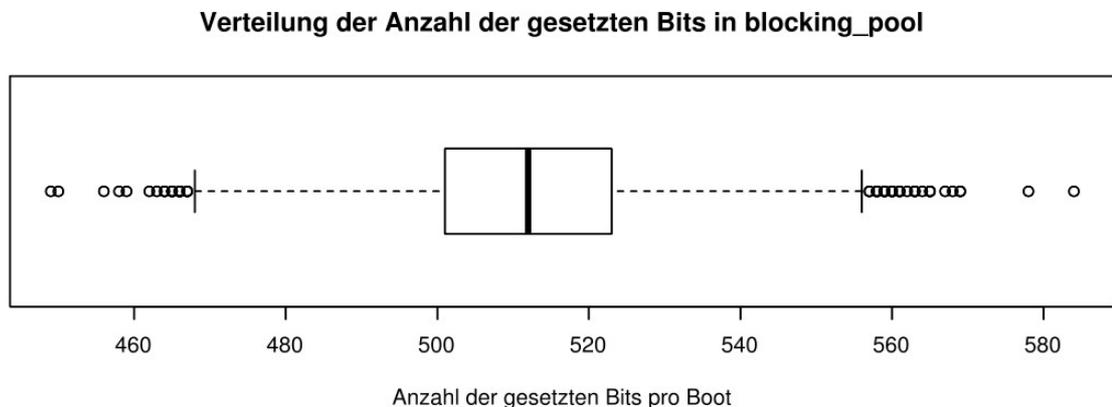


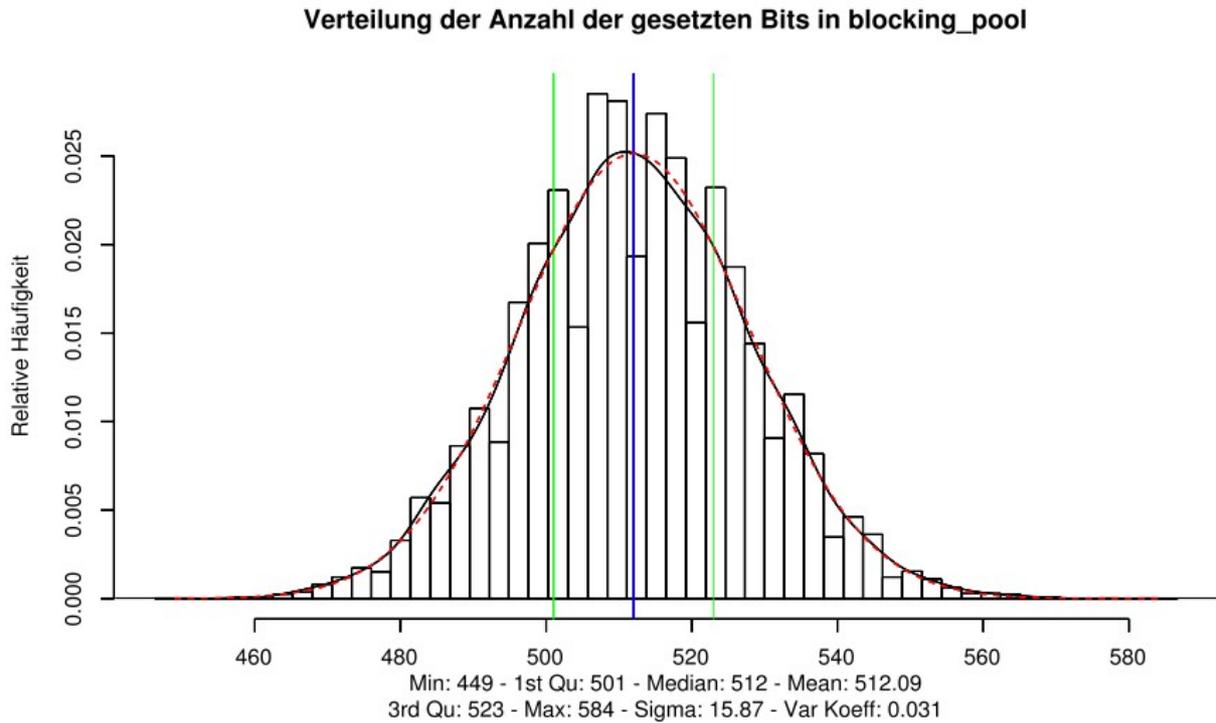
Interessanterweise gibt es im Histogramm größere Auffälligkeiten in dem es große Einbrüche in bei einigen Häufigkeiten im Histogramm gibt. Diese Auffälligkeiten nehmen beim Startvorgang noch zu **und unterstützen die These einer Gleichverteilung nicht.**

Es ist demnach wichtig, dass der Seed eingebracht wird, um den nonblocking_pool weiter zu durchmischen.

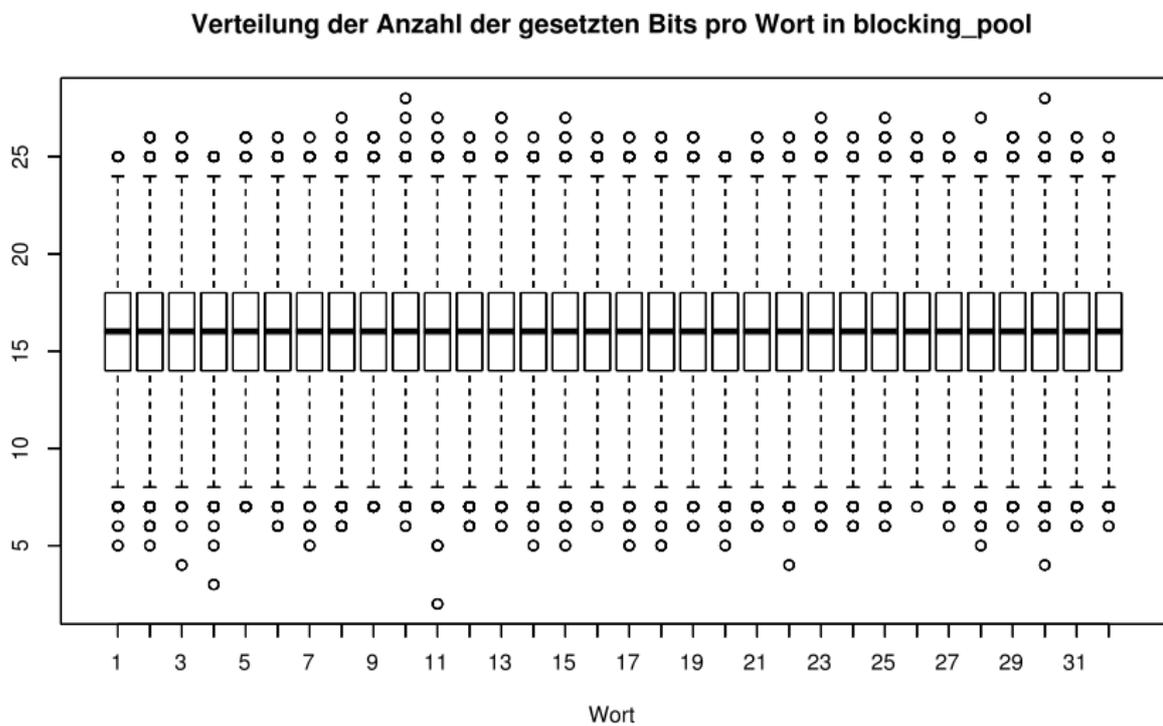
5.2.8 Test 1 - Gesetzte Bits im Entropie-Pool blocking_pool zum Systemstart

5.2.8.1 Graphische Veranschaulichung der Testresultate und Interpretation der Ergebnisse vor Start des User-Space

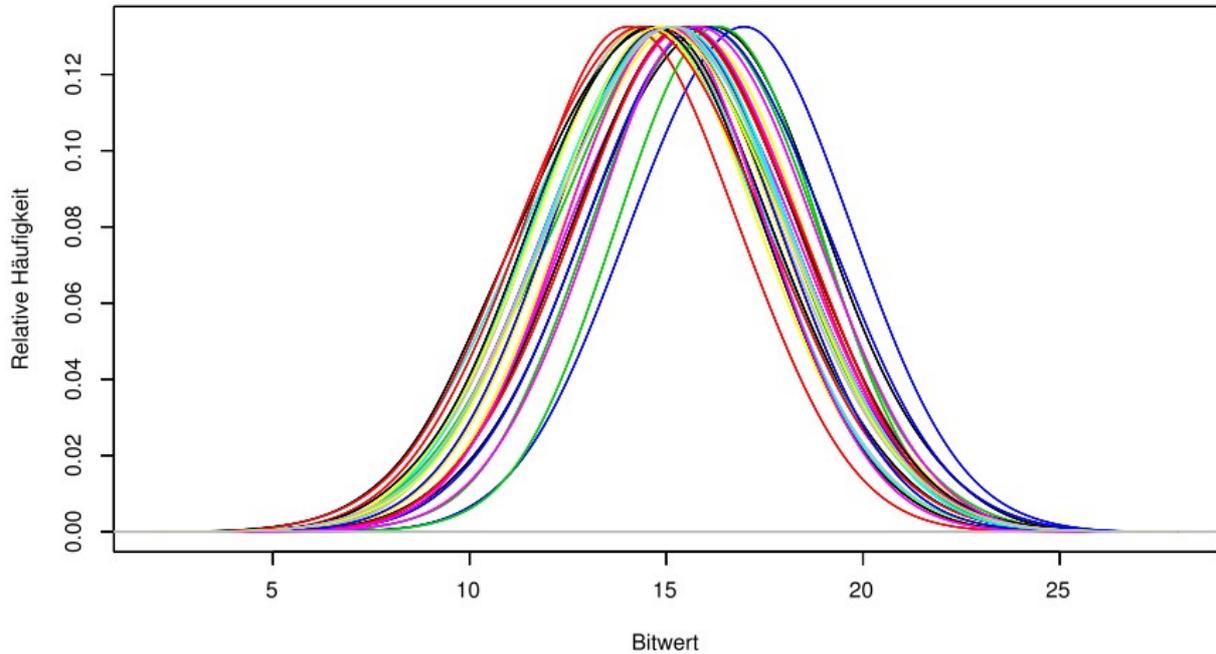




Wie in Abschnitt 5.2.7.1 werden im zweiten Teil dieser Analyse 32 Messreihen für die einzelnen 32-Bit-Wörter des `blocking_pool` betrachtet.

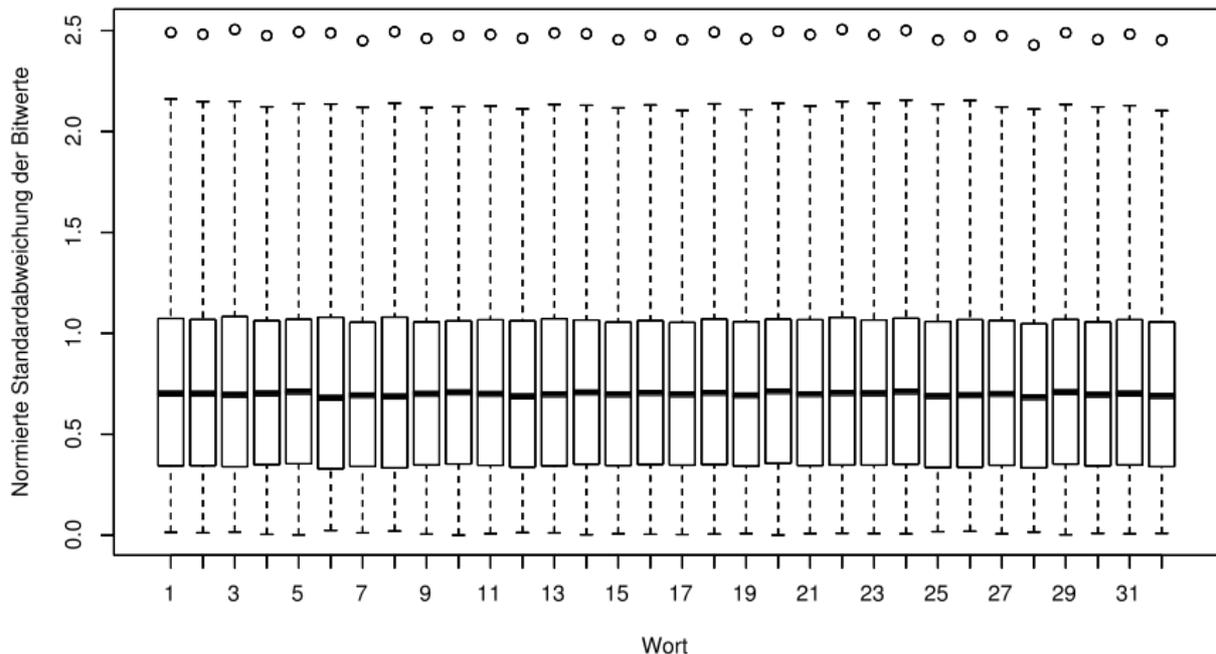


Verteilung der Bitsummen pro Wort in blocking_pool



Auch hier zeigt ein weiterer Box-Whisker-Plot die normierte Standardabweichung der Bit-Summen pro Wort.

Normierte Standardabweichung der Bitwerte pro Wort im blocking_pool



Wie in Abschnitt 5.2.4 zeigt die Dichte der Bit-Summen des Entropie-Pools kaum Hinweise, die gegen die vermutete Gleichverteilung der Daten im blocking_pool sprechen. Dennoch, das erste Histogramm zeigt einige Auffälligkeiten, die als bedenklich eingestuft werden müssen.

Die wortweise Analyse zeigt die erwarteten Normalverteilungen, die um den Mittelwert/Median schwanken. Die normierten Standardabweichungen der Bit-Summen zeigt eine breite, aber gleichmäßige Streuung der Werte. **Dementsprechend liefert dieser Test keinen Hinweis auf eine eventuelle Ungleichverteilung.**

5.2.8.2 Graphische Veranschaulichung der Testresultate und Interpretation der Ergebnisse vor Hinzufügen des Seeds

Zufallszahlen aus dem `blocking_pool` können im Gegensatz zum `nonblocking_pool` nur vom User-Space via `/dev/random` ausgelesen. Der Startvorgang bis zum Verarbeiten des Seeds führt aber keine Anfragen an `/dev/random` durch. Demzufolge wurde der Pool auch nicht verändert. Somit sind die Messergebnisse für den `blocking_pool` vor dem Hinzufügen des Seeds identisch mit den Ergebnissen im vorangegangenen Abschnitt 5.2.8.1.

Aufgrund der gefundenen Auffälligkeiten ist es als wichtig anzusehen, dass der Seed eingebracht wird, um den `blocking_pool` weiter zu durchmischen.

5.3 Test 2 - Gesetzte Bits für jede Bitposition

In diesem Test wird, wie im Überblick in 5.1 dargelegt, zu jedem Beobachtungszeitpunkt die Anzahl der gesetzten Bits für jede Bitposition bestimmt.

5.3.1 Test 2 im laufenden Betrieb - Testansatz

Ausgehend von diesen Anfangsüberlegungen wurde folgender Test erstellt, welcher im Verzeichnis `entropy_pool_distribution` zu finden ist:

- Ein SystemTap-Skript `entropy_pool_distribution.stp` wurde erstellt, in welchem die Stichprobenanzahl mit der Variable `num_samplings` angepasst werden kann. Dieses Skript berechnet die Anzahl der gesetzten Bits pro Bitposition.
- Das SystemTap-Skript wird mit dem Bash-Skript `gendata.sh` aufgerufen.
- Ein R-Project-Analyseprogramm `entropy_pool_distribution.r` erstellt die folgenden Graphiken aus der mittels dem SystemTap-Skript erzeugten Datenreihe.

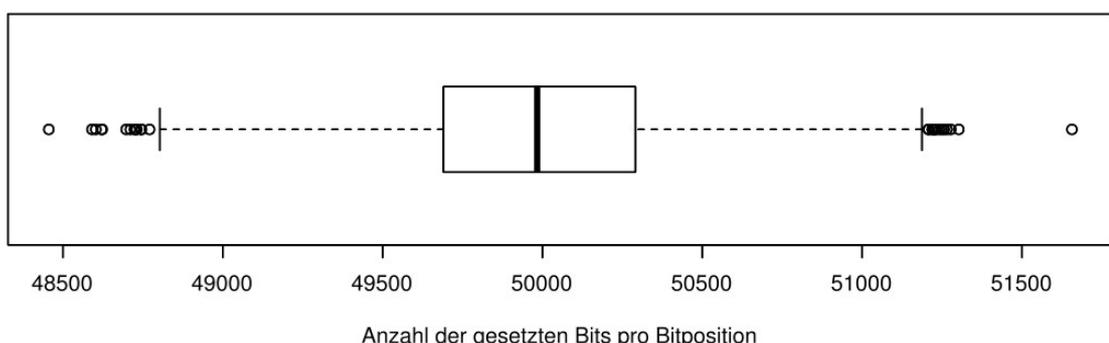
Als Stichprobengröße wurde $S=100.000$ gewählt.

Wie in Abschnitt 5.1 beschrieben, erwartet man, dass jedes Bit mit einer Wahrscheinlichkeit von etwa 50% gesetzt ist. Insbesondere rechnen wir also damit, dass die gemessenen Bit-Summen der einzelnen Bitpositionen um $S/2=50000$ normalverteilt sind.

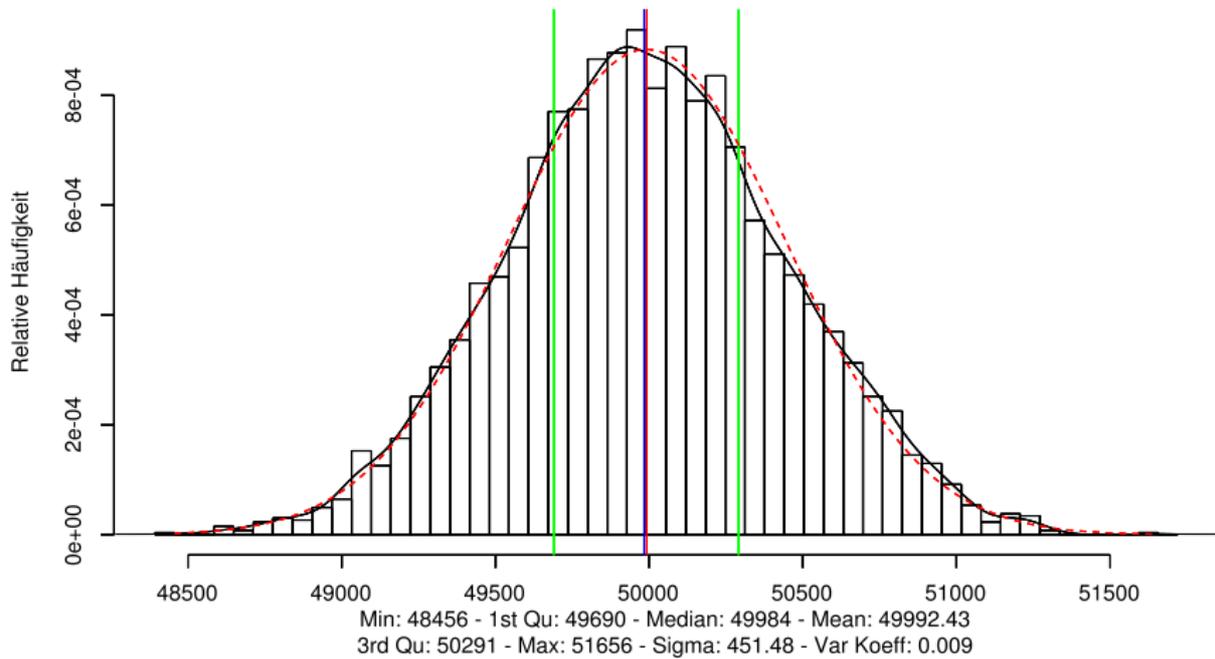
5.3.2 Test 2 - Gesetzte Bits für jede Bitposition im Entropie-Pool `input_pool`

5.3.2.1 Graphische Veranschaulichung der Testresultate und Interpretation der Ergebnisse

Verteilung der Anzahl der gesetzten Bits pro Bitposition in `input_pool`

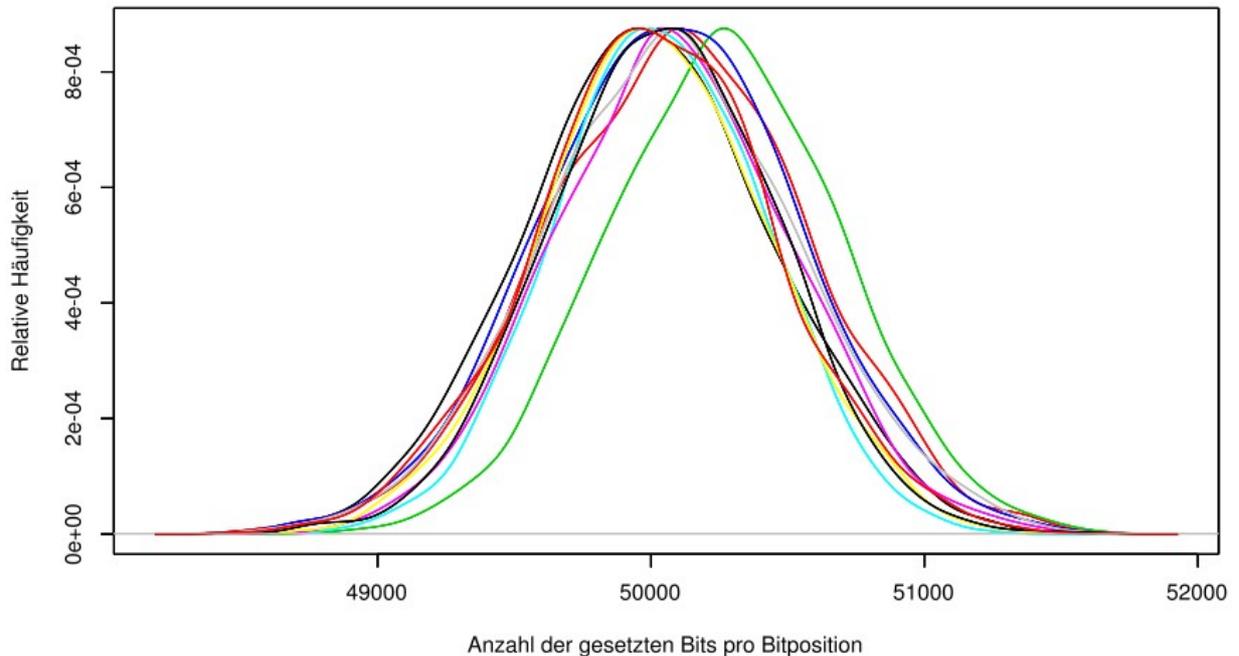


Verteilung der Anzahl der gesetzten Bits pro Bitposition in input_pool



Das Verhalten der Verteilung über die Zeit wurde mit 10 Testreihen mit mit der oben angegebenen Stichprobengröße durchgeführt. Die schwarze Kurve entspricht der ersten Testreihe, die obigem Box-Whisker-Plot und dem zugehörigen Histogramm zugrunde liegt.

Verteilung der Anzahl der gesetzten Bits pro Bitposition in input_pool



Die Verteilungen der Veränderungen zeigen die erwarteten Resultate:

- Der Mittelwert und der Median sind fast identisch.

- Das Histogramm zeigt eine fast perfekte Normalverteilung.
- Die Streuung der Werte um den Mittelwert ist relativ gering, wie im Histogramm und im Box-Whisker-Plot ersichtlich.
- Die Veränderung der Verteilung beim Wiederholen des Tests ist ebenfalls sehr gering.
- Der Variationskoeffizient zeigt, dass die Verteilung sehr „schmal“ im Verhältnis der Gesamtdaten ist.

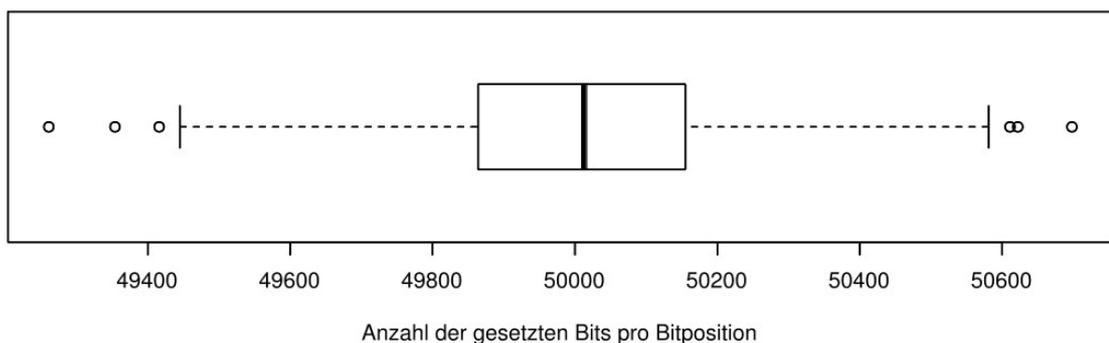
Die Ergebnisse entsprechen den Erwartungen und widersprechen weder einer Gleichverteilung der Daten des Pools noch der Annahme, dass jedes Bit mit einer Wahrscheinlichkeit von etwa 50% gesetzt ist, nicht.

5.3.3 Test 2 - Gesetzte Bits für jede Bitposition im Entropie-Pool `nonblocking_pool`

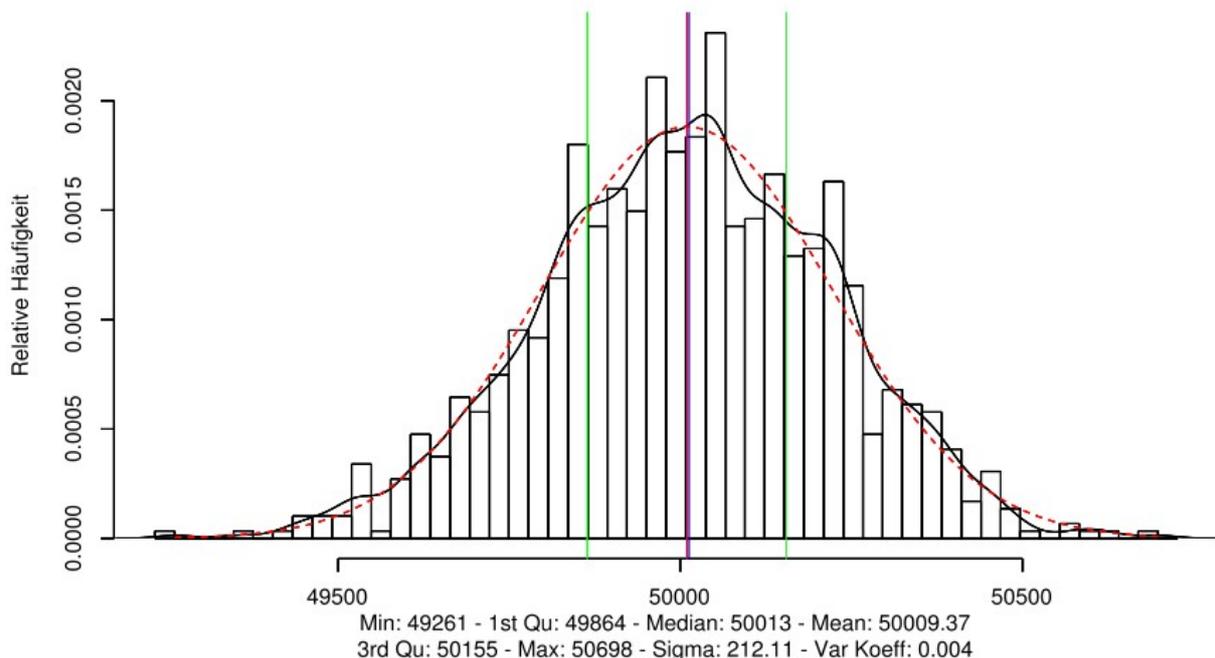
Wie in Abschnitt 5.2.3 dargelegt, erachten wir eine gesonderte Betrachtung von ausschließlich Lese- oder Schreibvorgängen auf den Gerätedateien nicht als notwendig.

5.3.3.1 Graphische Veranschaulichung der Testresultate und Interpretation der Ergebnisse

Verteilung der Anzahl der gesetzten Bits pro Bitposition in `nonblocking_pool`



Verteilung der Anzahl der gesetzten Bits pro Bitposition in `nonblocking_pool`



Die Verteilungen zeigen die folgenden Ergebnisse:

- Der Mittelwert und der Median sind fast identisch.
- Das Histogramm zeigt eine Deformation der Normalverteilung. Die Ursache dieser leichten Deformierung ist unklar. Dennoch sollte man sich vor einer Interpretation die Breite der Verteilung im Verhältnis zur theoretischen Breite der Beobachtungen vor Augen führen: der Variationskoeffizient zeigt, dass die Verteilung äußerst „schmal“ ist und damit kann diese leichte Deformation als nicht signifikant betrachtet werden.
- Die Streuung der Werte um den Mittelwert ist relativ gering, wie im Histogramm und im Box-Whisker-Plot ersichtlich.
- Der Variationskoeffizient zeigt, dass die Verteilung sehr „schmal“ im Verhältnis der Gesamtdaten ist.

Die Ergebnisse liegen trotz der angesprochenen leichten Deformation noch im Rahmen der Erwartungen und widersprechen weder einer Gleichverteilung der Daten des Pools noch der Annahme, dass jedes Bit mit einer Wahrscheinlichkeit von etwa 50% gesetzt ist, nicht.

Zu beachten ist: Damit die Beispieldaten gesammelt werden konnten, wurde der Befehl

```
cat /dev/urandom
```

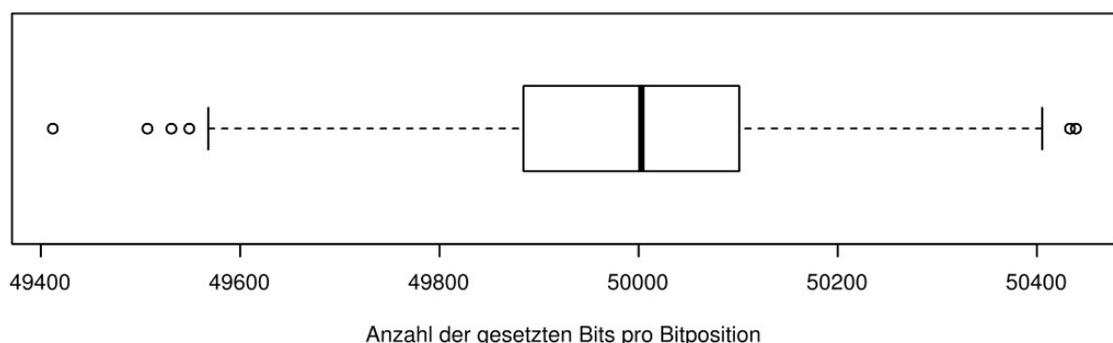
parallel mit der Messung ausgeführt. Dabei liest dieser Befehl ununterbrochen Daten aus `nonblocking_pool`. Er wurde verwendet, um Ergebniswerte in einer angemessenen Zeit zu erzeugen. Mit diesem Befehl wurde somit in erster Linie der deterministische Teil des Entropie-Pools gemessen, da kaum andere Daten aus einer Rauschquelle in `nonblocking_pool` eingehen.

5.3.4 Test 2 - Gesetzte Bits für jede Bitposition im Entropie-Pool `blocking_pool`

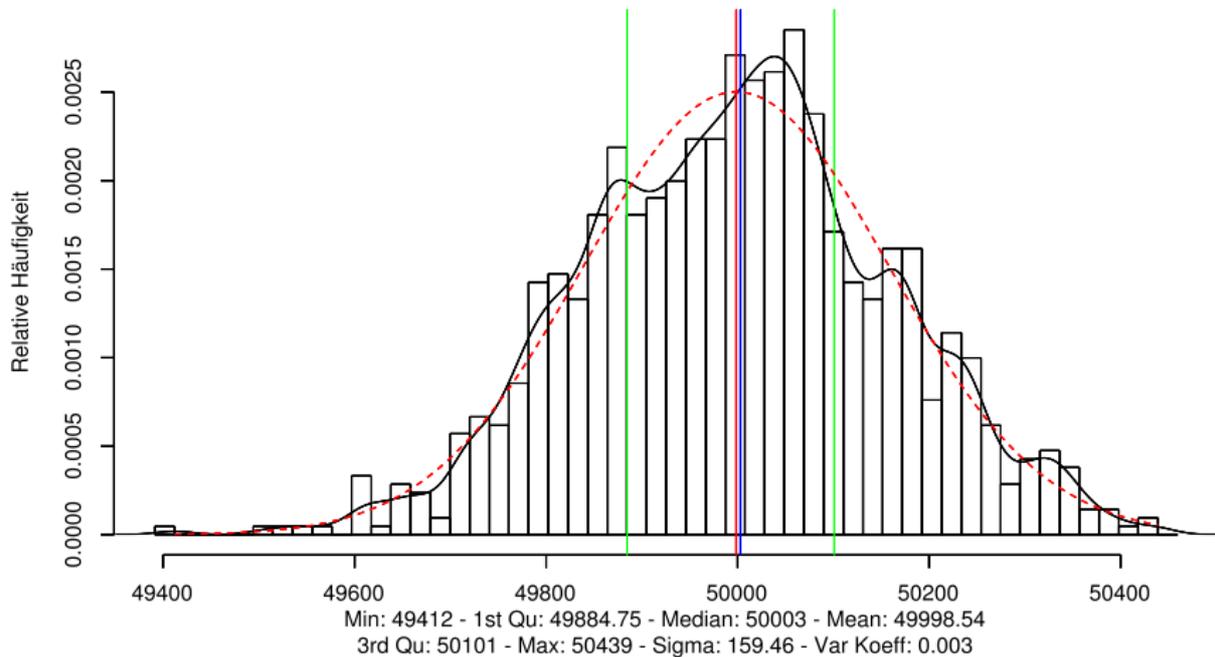
Wie in 5.2.4 verzichten wir hier auf eine gesonderte Betrachtung von ausschließlich Lese- oder Schreibvorgängen auf den Gerätedateien.

5.3.4.1 Graphische Veranschaulichung der Testresultate und Interpretation der Ergebnisse

Verteilung der Anzahl der gesetzten Bits pro Bitposition in `blocking_pool`



Verteilung der Anzahl der gesetzten Bits pro Bitposition in blocking_pool



Die Verteilungen zeigen größtenteils die erwarteten Resultate:

- Der Mittelwert und der Median sind fast identisch und liegen bei der Hälfte der Stichprobengröße.
- Das Histogramm zeigt eine leicht deformierte Normalverteilung. Die Ursache dieser leichten Deformation ist unklar. Dennoch sollte man sich vor einer Interpretation die Breite der Verteilung im Verhältnis zur theoretischen Breite der Beobachtungen vor Augen führen: der Variationskoeffizient zeigt, dass die Verteilung äußerst „schmal“ ist und damit kann diese leichte Deformation als nicht signifikant betrachtet werden.
- Die Streuung der Werte um den Mittelwert ist relativ gering, wie im Histogramm und im Box-Whisker-Plot ersichtlich.
- Der Variationskoeffizient zeigt, dass die Verteilung sehr „schmal“ im Verhältnis der Gesamtdaten ist.
- Das Histogramm zeigt eine Deformation der Verteilung, welche von der Normalverteilung abweicht. Eine Erklärung der Deformation kann nicht gefunden werden.

Die Ergebnisse liegen trotz der angesprochenen leichten Deformation noch im Rahmen der Erwartungen und widersprechen einer Gleichverteilung der Daten des Pools nicht. Ebenso wurde kein Widerspruch zur Annahme, dass jedes Bit mit einer Wahrscheinlichkeit von etwa 50% gesetzt ist, gefunden.

5.3.5 Test 2 zum Systemstart - Vorüberlegungen und Testansatz

Es gelten die gleichen Vorüberlegungen wie in Abschnitt 5.2.5.1. Zudem werden die gleichen Messdaten wie in Abschnitt 5.2.5.2 verwendet. Die für den Test relevanten Dateien finden sich in dem `corr_pool_boot`, insbesondere

- Das R-Project Programm `corr_pools_boot.r` liest alle Testreihen, welche durch den in 5.2.5 definierten Test gewonnen wurden, in eine Matrix ein, um Analysen durchführen zu können.
- Die gewählte Stichprobengröße ergibt sich aus wie in Abschnitt 5.2.5.2.

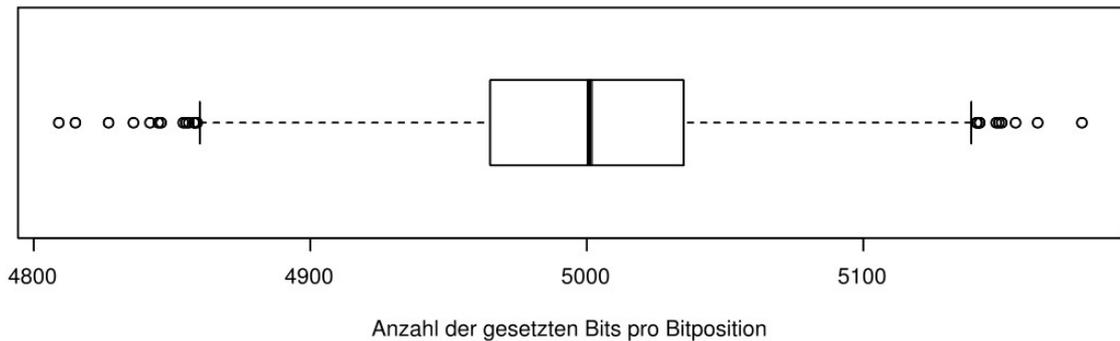
Die folgenden Unterabschnitte beinhalten die vorher genannten sechs Analysen.

5.3.6 Test 2 - Gesetzte Bits für jede Bitposition im Entropie-Pool input_pool zum Systemstart

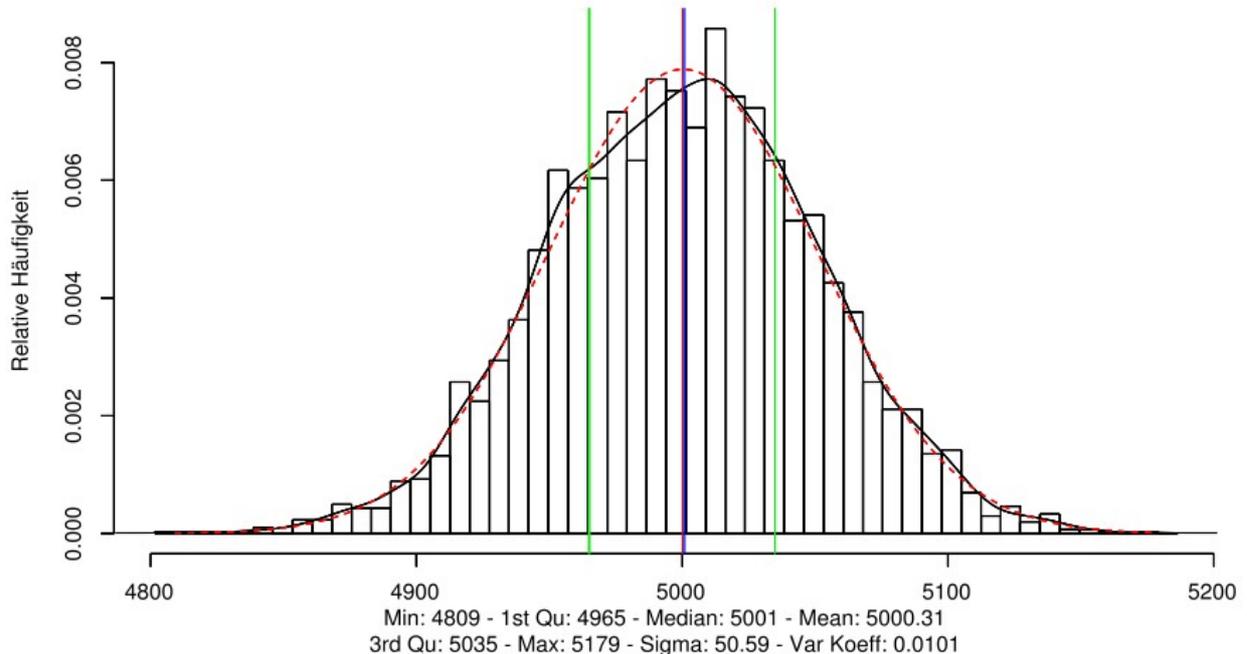
5.3.6.1 Graphische Veranschaulichung der Testresultate und Interpretation der Ergebnisse vor Start des User-Space

Der erste Teil der Analyse zeigt die Verteilung der Bit-Summen der Bitpositionen innerhalb des gesamten Entropie-Pools.

Verteilung der Anzahl der gesetzten Bits in input_pool

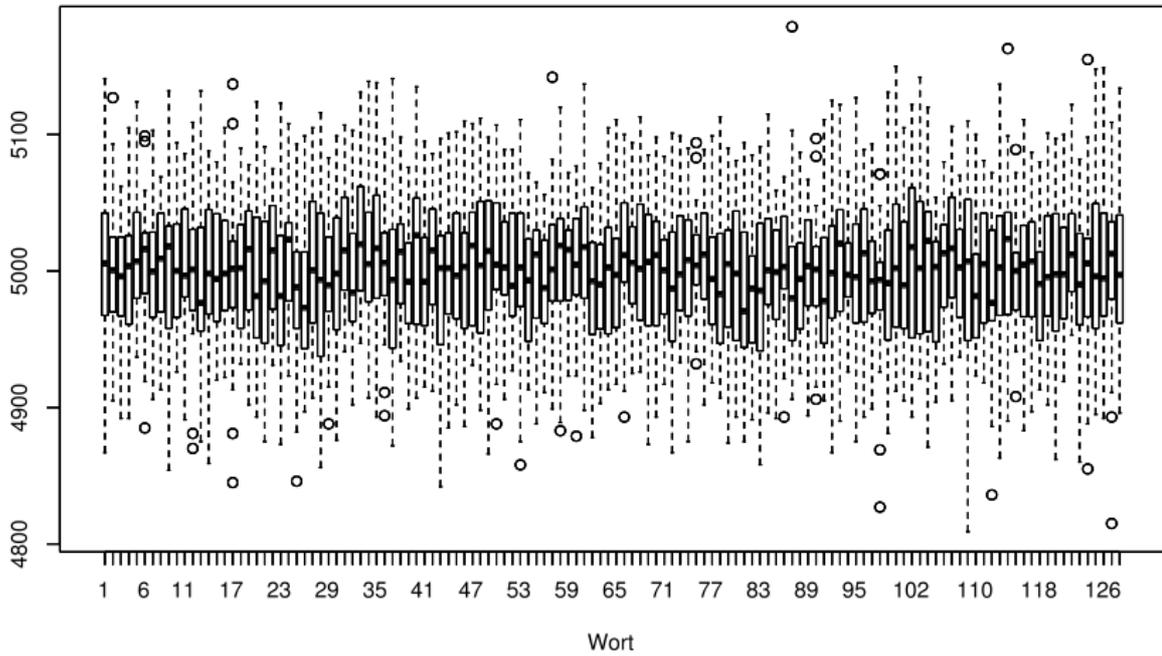


Verteilung der Anzahl der gesetzten Bits in input_pool

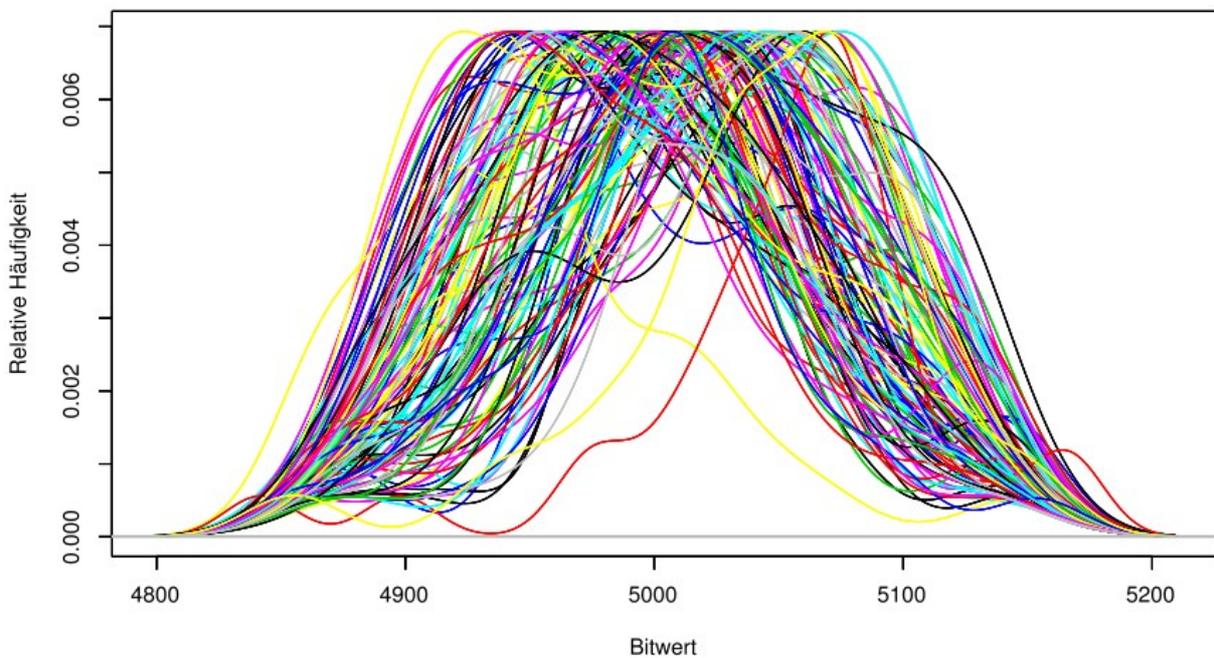


Der zweite Teil der Analyse diskutiert die Verteilungen der Bit-Summen der Bitpositionen für die einzelnen Wörter. Wie bereits beschrieben, enthält input_pool 128 Wörter à 32 Bit, weshalb 128 Messreihen betrachtet werden.

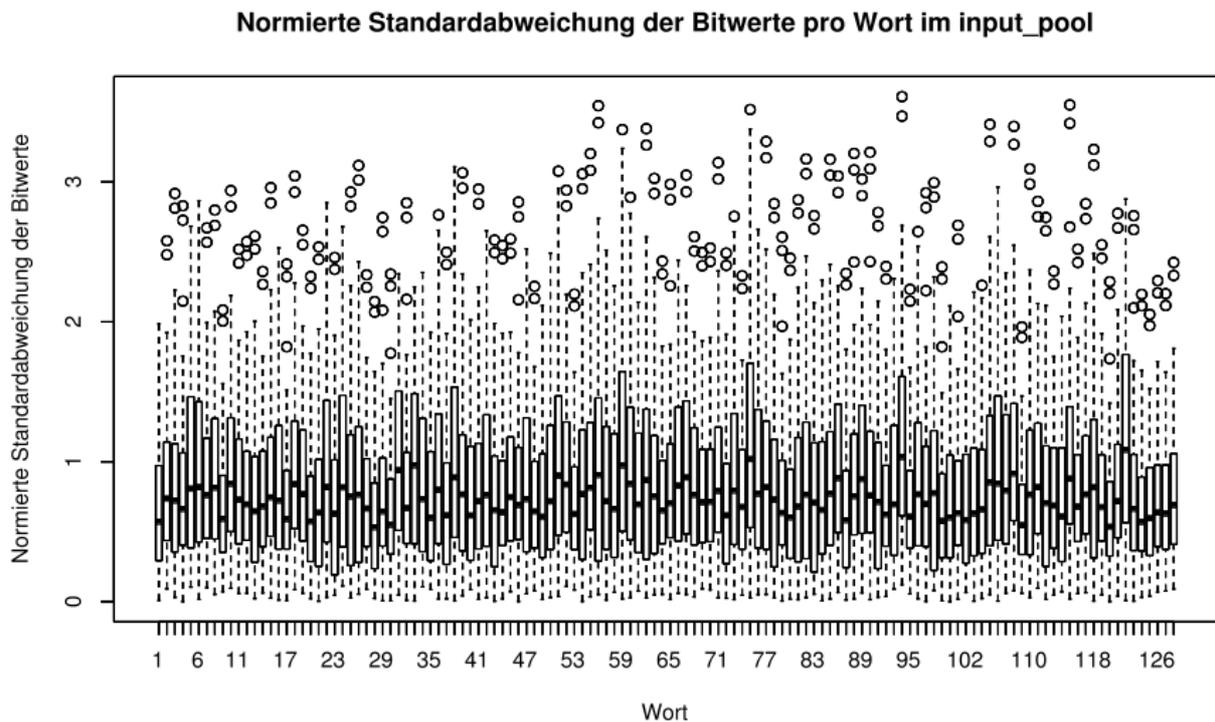
Verteilung der Anzahl der gesetzten Bits pro Wort in input_pool



Verteilung der Bitsummen pro Wort in input_pool



Abschließend zeigt ein weiterer Box-Whisker-Plot die normierte Standardabweichung der Bit-Summen der einzelnen Bitpositionen pro Wort.



Die Dichte und das Histogramm zeigen eine leicht deformierte Verteilung an der Spitze. Der vermutete Grund ist die Ähnlichkeit und die geringe Anzahl der Ereignisse, die beim Startvorgang erzeugt werden.

Die Verteilung der Bit-Summen der einzelnen Bitpositionen innerhalb des Entropie-Pools zeigt Hinweise auf Widersprüche bezüglich der Annahme einer Gleichverteilung der Daten und zur Annahme, dass jedes Bit mit einer Wahrscheinlichkeit von etwa 50% gesetzt ist.

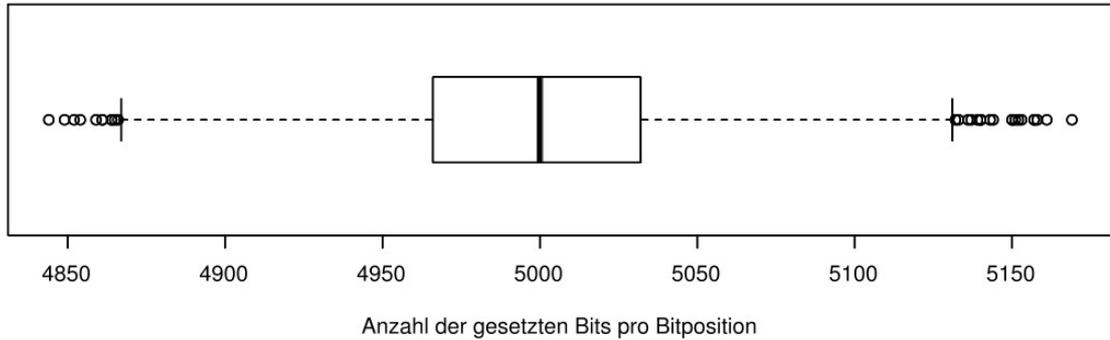
Interessant ist die relativ starke Schwankungsbreite, wenn man die Dichteverteilung aller Wörter übereinander legt. In diesem Diagramm ist auch zu sehen, dass die Verteilung innerhalb einzelner Wörter teilweise erheblich von einer Normalverteilung abweicht, indem sie beispielsweise zwei lokale Maxima besitzt.

Dieser Unterschied ist auch an den normierten Standardabweichungen zu beobachten: Die Box-Plots und der Median der verschiedenen Wörter differieren stark. Es zeigen sich aber keine signifikanten Ausreißer. Alle Box-Whisker Plots der Standardabweichungen haben einen Bias in Richtung Null. Das bedeutet, dass die Whisker Richtung Null kleiner sind, als die Whisker in die andere Richtung. Weiterhin ist der Medianwert in der Box in Richtung Null verschoben. Dieses Ergebnis ist bei einer Normalverteilung zu erwarten.

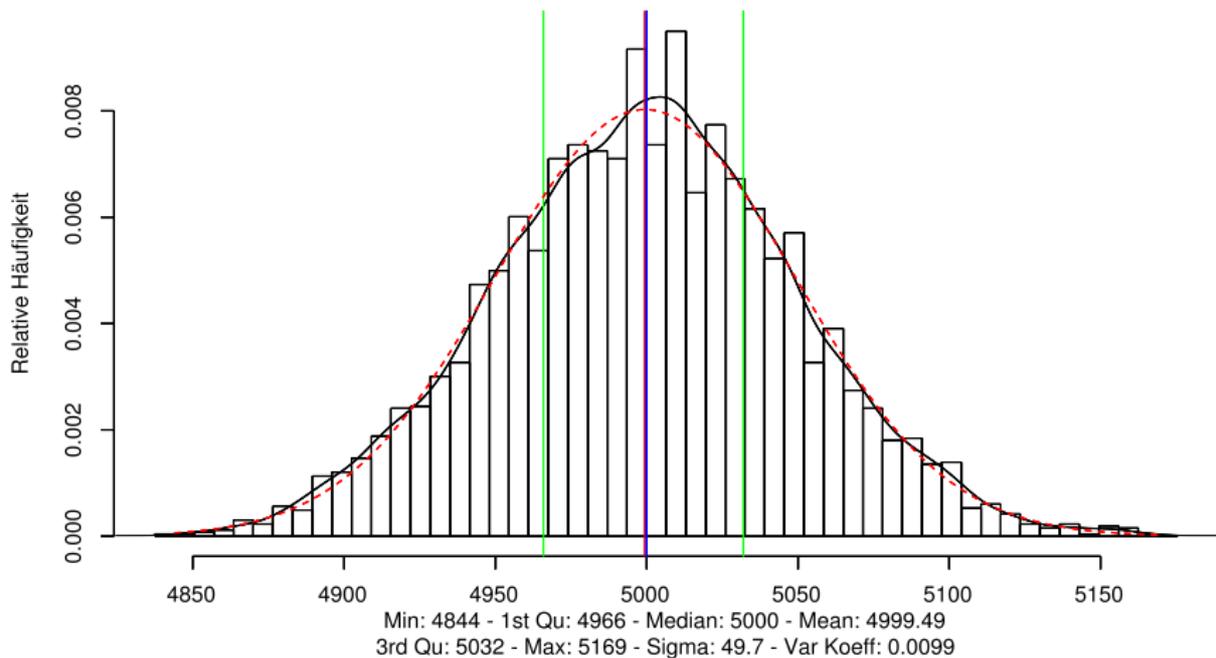
Somit scheinen die wortweisen Verteilungen der Anzahl der gesetzten Bits pro Bitposition nicht immer einer Normalverteilung zu entsprechen. Die starken Abweichungen werden jedoch bei der Gesamtbetrachtung des Entropie-Pools reduziert und haben keinen nennenswerten Einfluss auf die Gesamtverteilung im Entropie-Pool. Hinweise gegen die Annahme, dass jedes Bit in etwa 50% der Fälle gesetzt ist, wurden nicht gefunden.

5.3.6.2 Graphische Veranschaulichung der Testresultate und Interpretation der Ergebnisse vor Hinzufügen des Seeds

Verteilung der Anzahl der gesetzten Bits in input_pool

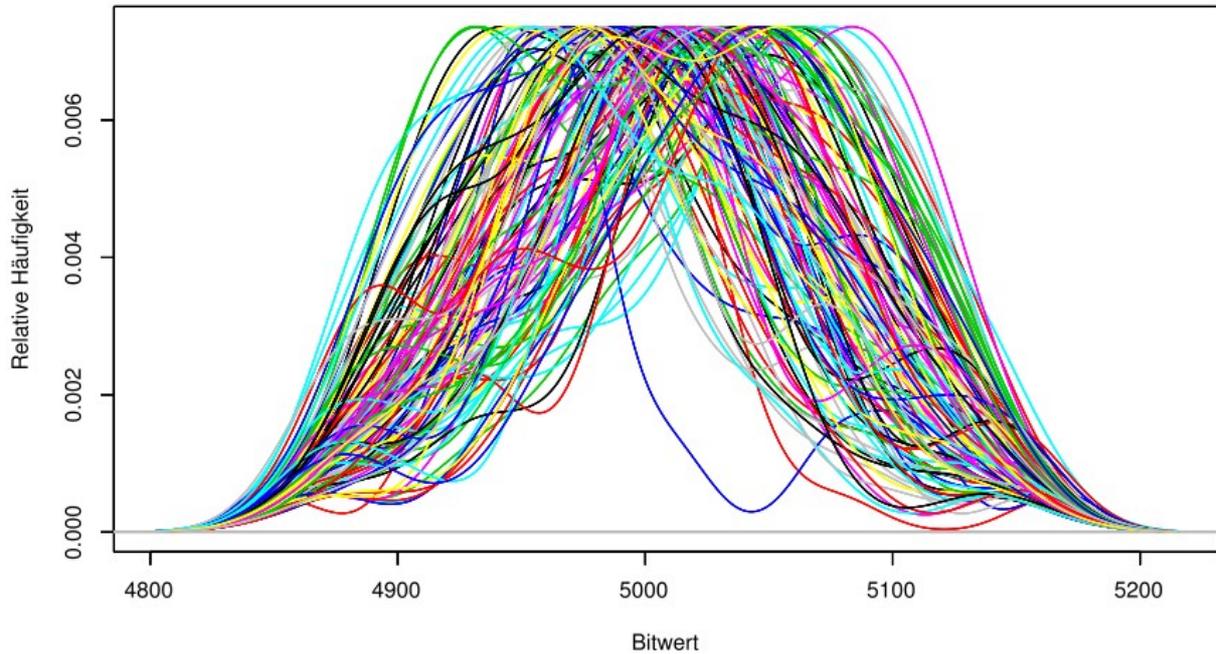


Verteilung der Anzahl der gesetzten Bits in input_pool



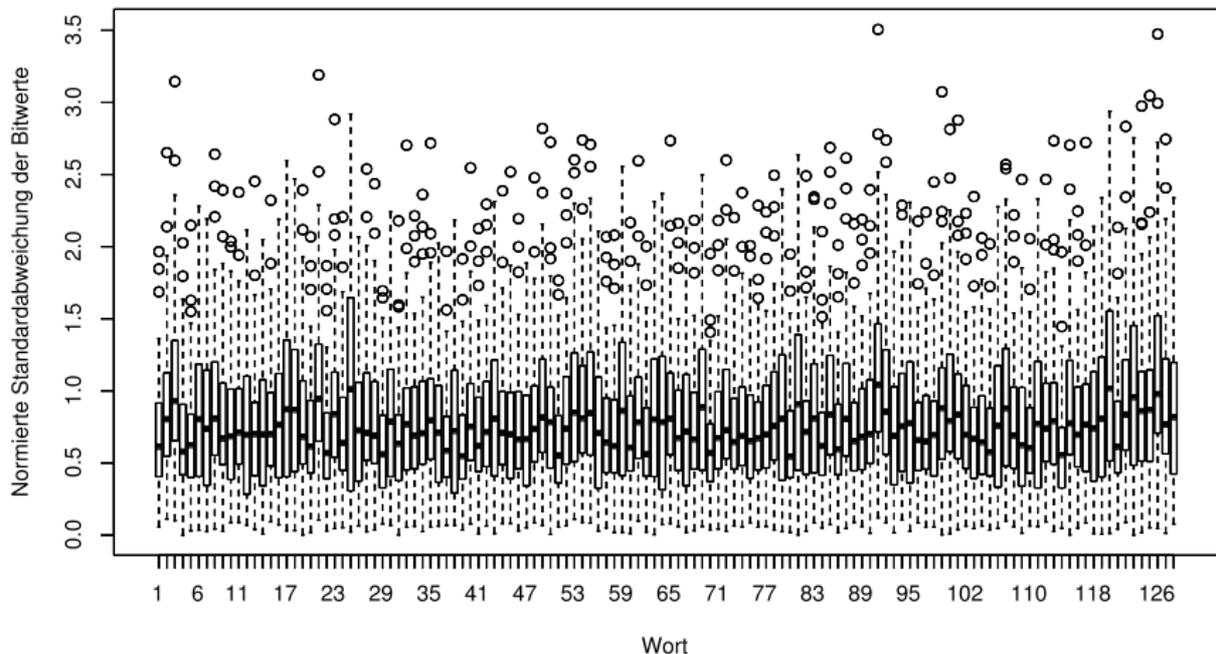
Auch hier folgt im zweiten Teil die wortweise Betrachtung der Bit-Summen einzelner Bitpositionen.

Verteilung der Bitsummen pro Wort in input_pool



Abschließend folgt wieder ein Box-Whisker-Plot zur Darstellung der normierten Standardabweichungen der Bit-Summen der Bitpositionen für jedes Wort.

Normierte Standardabweichung der Bitwerte pro Wort im input_pool



Im Gegensatz zu der Messung des Entropie-Pools zum Systemstart sind jetzt erheblich mehr Hardware-Ereignisse in den Pool eingegangen. Mit jedem Ereignis werden potentielle Schwächen, resultierend aus möglichen Ähnlichkeiten der Ereignisse, immer geringer. Dies ist in den Verteilungen zu sehen, da die Verteilungen in diesem Abschnitt etwas besser einer

Normalverteilung gleichen als jene Verteilungen, welche den Zustand des Pools vor dem Start des User-Space zeigen.

Wie erwartet gleicht die Verteilung der Bit-Summen der einzelnen Bitpositionen innerhalb des Entropie-Pools einer Normalverteilung erheblich besser, als jene Verteilung basierend auf den Daten vor dem Start des User-Space.

Wie zu erwarten war, ähneln die Diagramme denen des vorherigen Abschnitts.

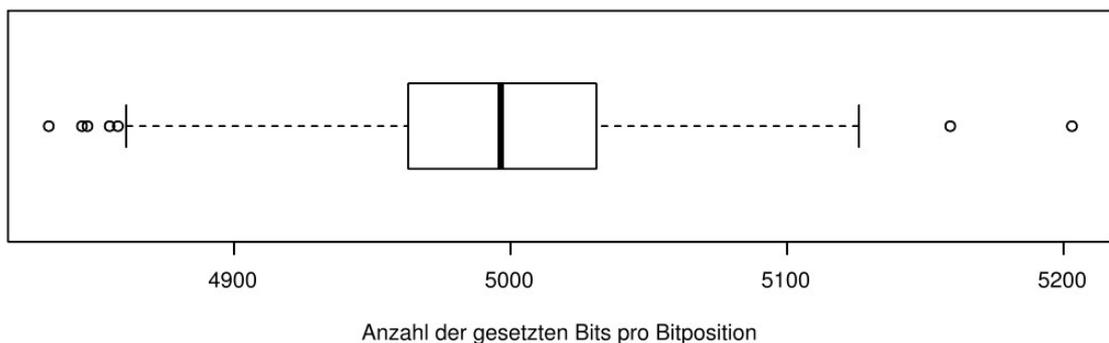
Es bleibt noch festzustellen, dass die wortweise Verteilung der Bit-Summen pro Bitposition nicht immer einer Normalverteilung entspricht. Die starken Abweichungen werden jedoch bei der Gesamtbetrachtung des Entropie-Pools reduziert und haben keinen nennenswerten Einfluss auf die Gesamtverteilung im Entropie-Pool. **Es wurden keine Hinweise gegen die Annahme, dass jedes Bit mit einer Wahrscheinlichkeit von etwa 50% gesetzt ist, gefunden.**

5.3.7 Test 2 - Gesetzte Bits für jede Bitposition im Entropie-Pool `nonblocking_pool` zum Systemstart

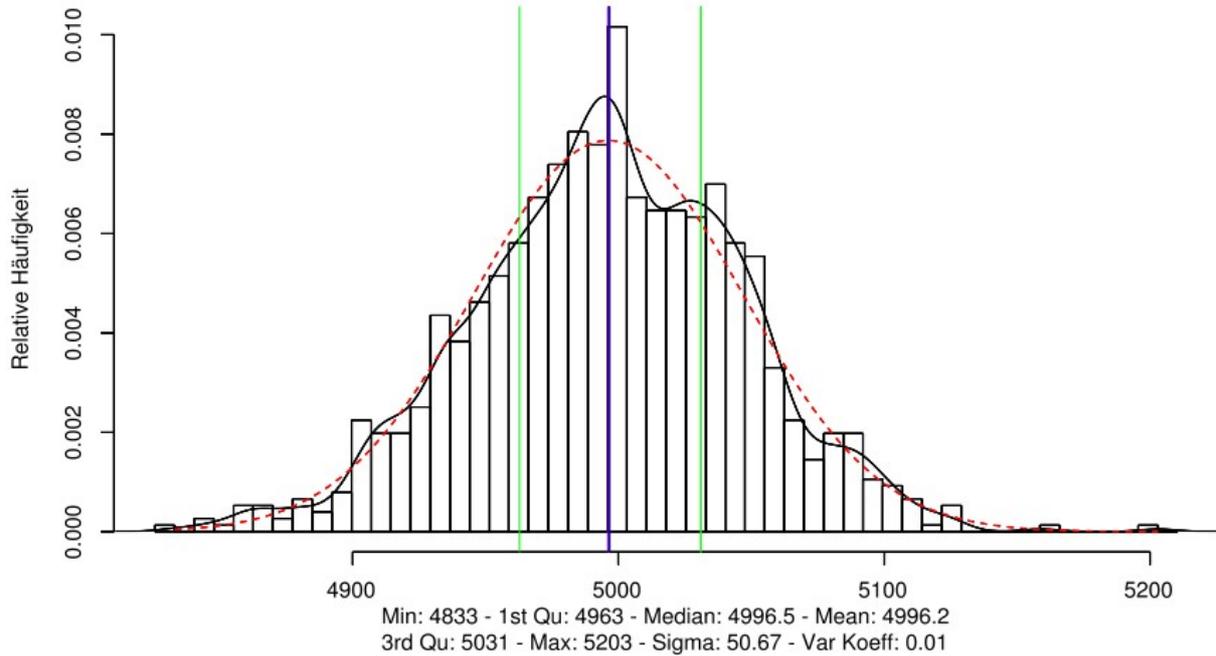
5.3.7.1 Graphische Veranschaulichung der Testresultate und Interpretation der Ergebnisse vor Start des User-Space

Wieder ist der erste Teil der Analyse der Verteilung der Bit-Summen der einzelnen Bitpositionen innerhalb des gesamten Entropie-Pools gewidmet.

Verteilung der Anzahl der gesetzten Bits in `nonblocking_pool`

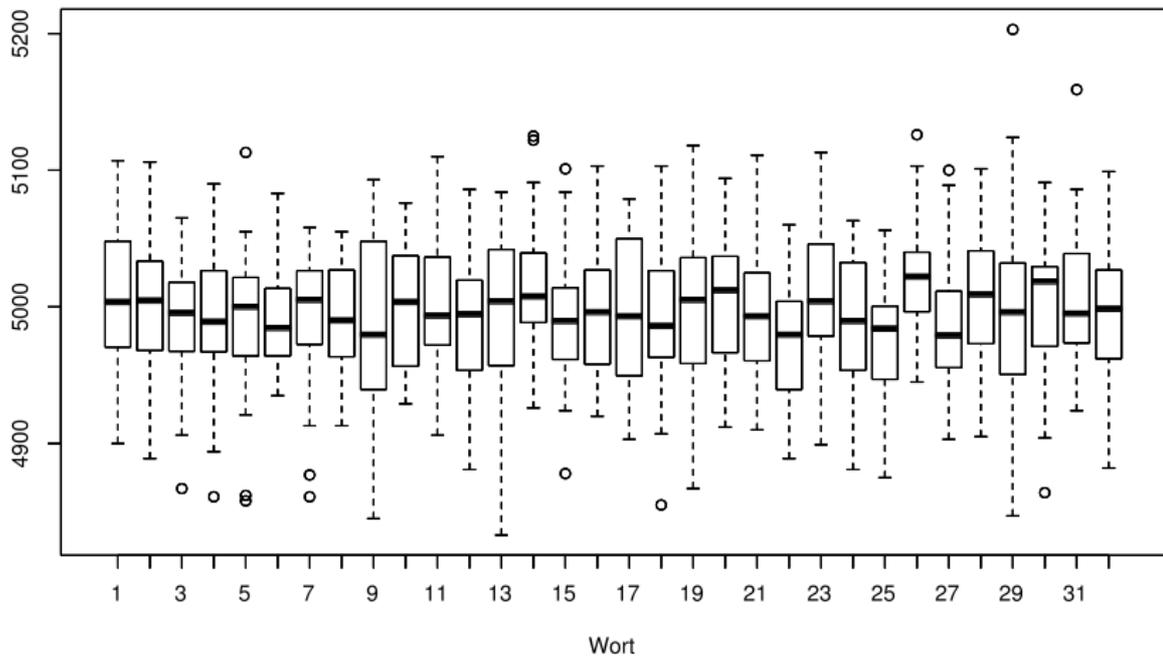


Verteilung der Anzahl der gesetzten Bits in nonblocking_pool

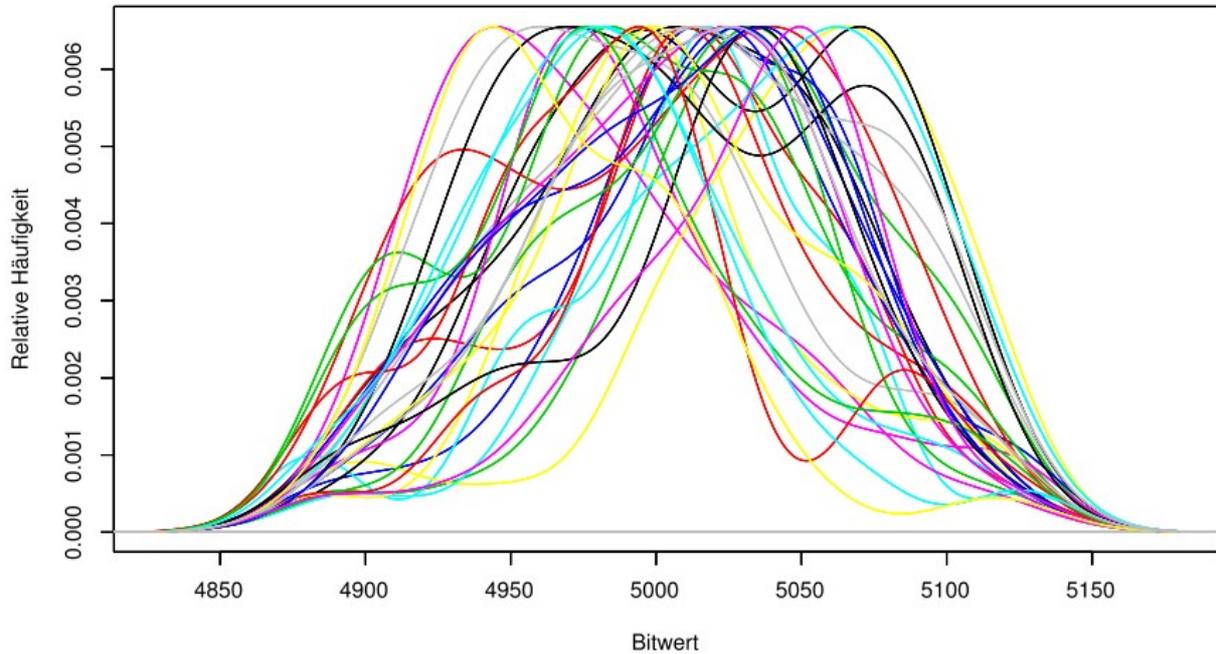


Wir erinnern hier nochmals daran, dass im zweiten Teil der Analyse 32 Verteilungen der Bit-Summen der Bitpositionen für jedes Wort betrachtet werden.

Verteilung der Anzahl der gesetzten Bits pro Wort in nonblocking_pool

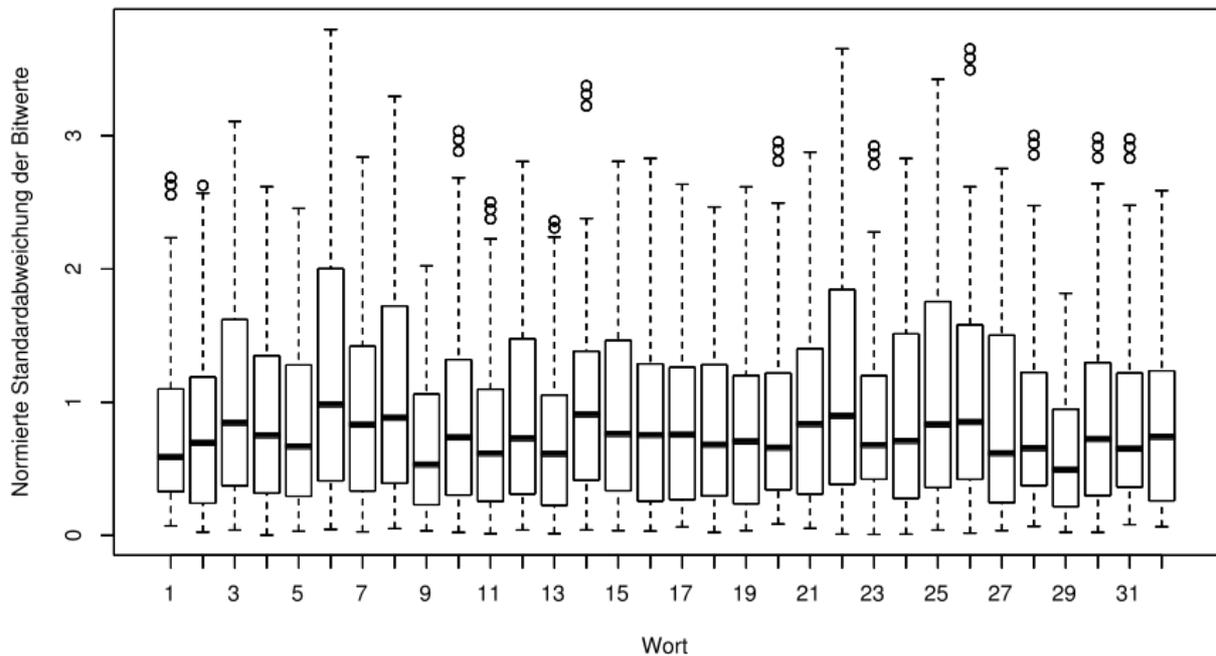


Verteilung der Bitsummen pro Wort in nonblocking_pool



Ein weiterer Box-Whisker-Plot zeigt noch die normierten Standardabweichungen der Bit-Summen für jedes Wort.

Normierte Standardabweichung der Bitwerte pro Wort im nonblocking_pool

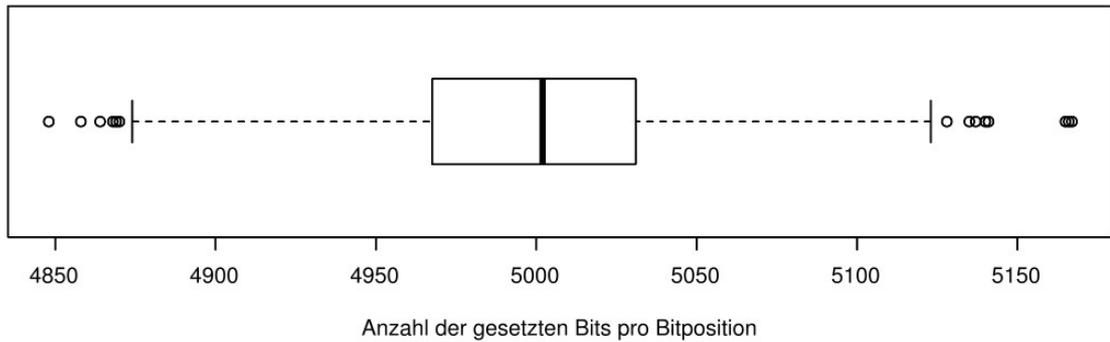


Die Verteilung der Bit-Summen für jede Bitposition innerhalb des gesamten Pools weicht nur leicht von einer Normalverteilung ab. Die wortweise Betrachtung zeigt teils deutliche Abweichungen von der Normalverteilung. Das könnte ein Hinweis auf eine etwas ungleiche, nicht jedoch auf eine kritische Abweichung von einer

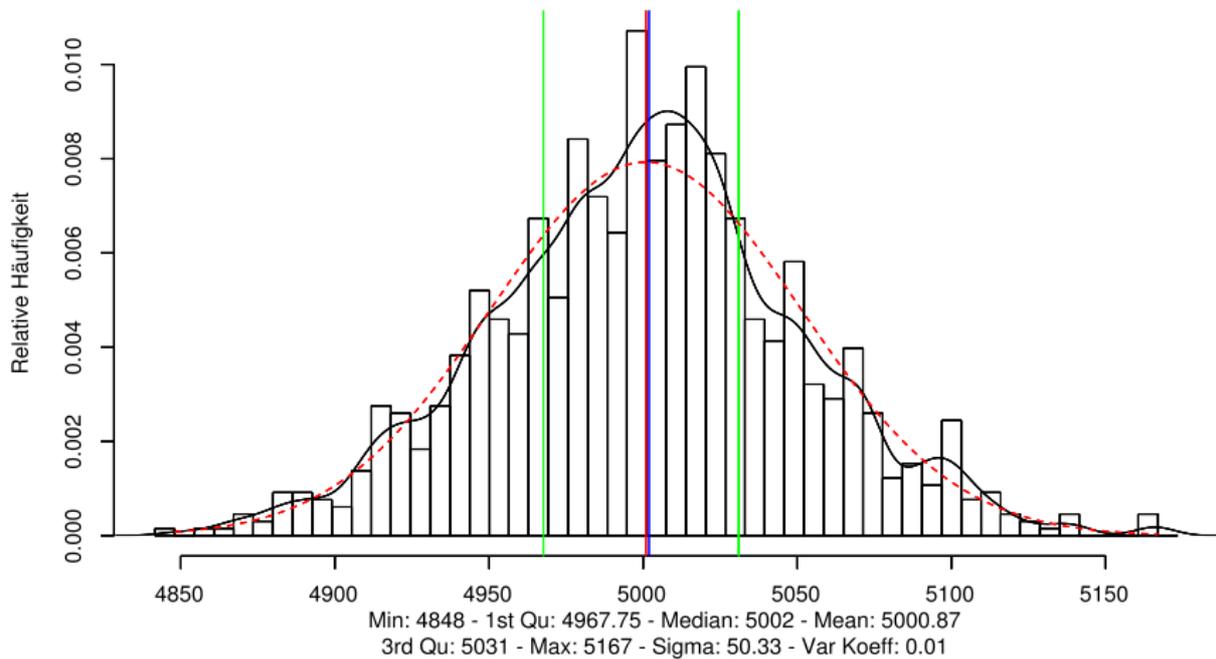
Gleichverteilung der Daten des Pools sein. Die einzelnen Bits scheinen auch hier mit einer Wahrscheinlichkeit im Bereich von 50% gesetzt zu sein.

5.3.7.2 Graphische Veranschaulichung der Testresultate und Interpretation der Ergebnisse vor Hinzufügen des Seeds

Verteilung der Anzahl der gesetzten Bits in nonblocking_pool

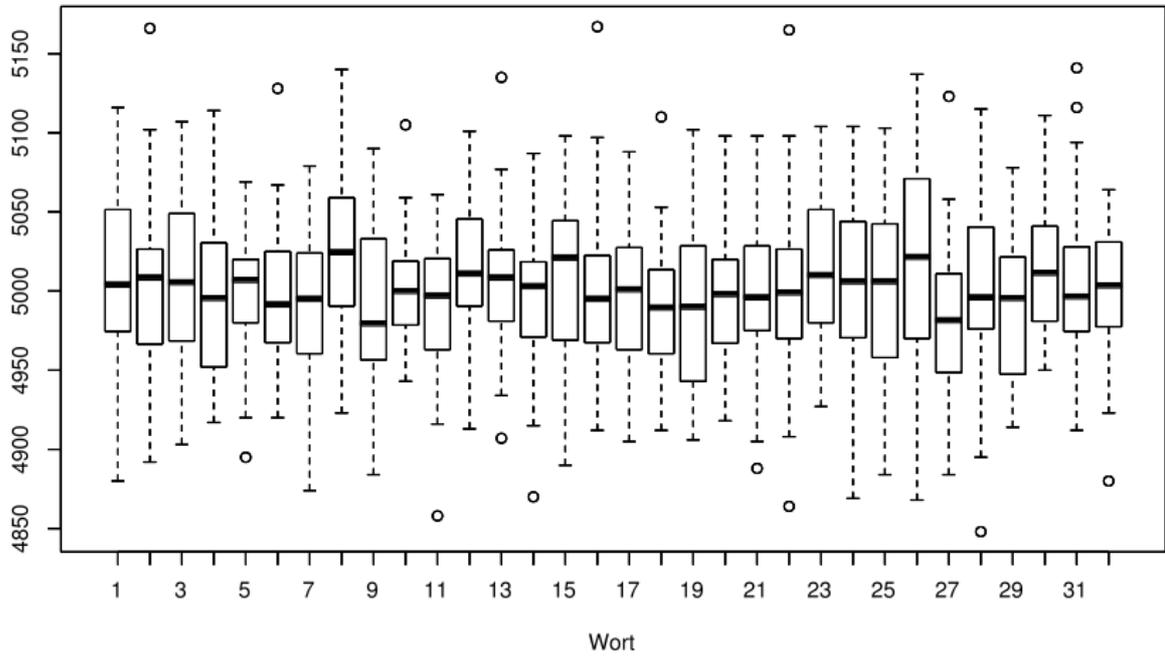


Verteilung der Anzahl der gesetzten Bits in nonblocking_pool

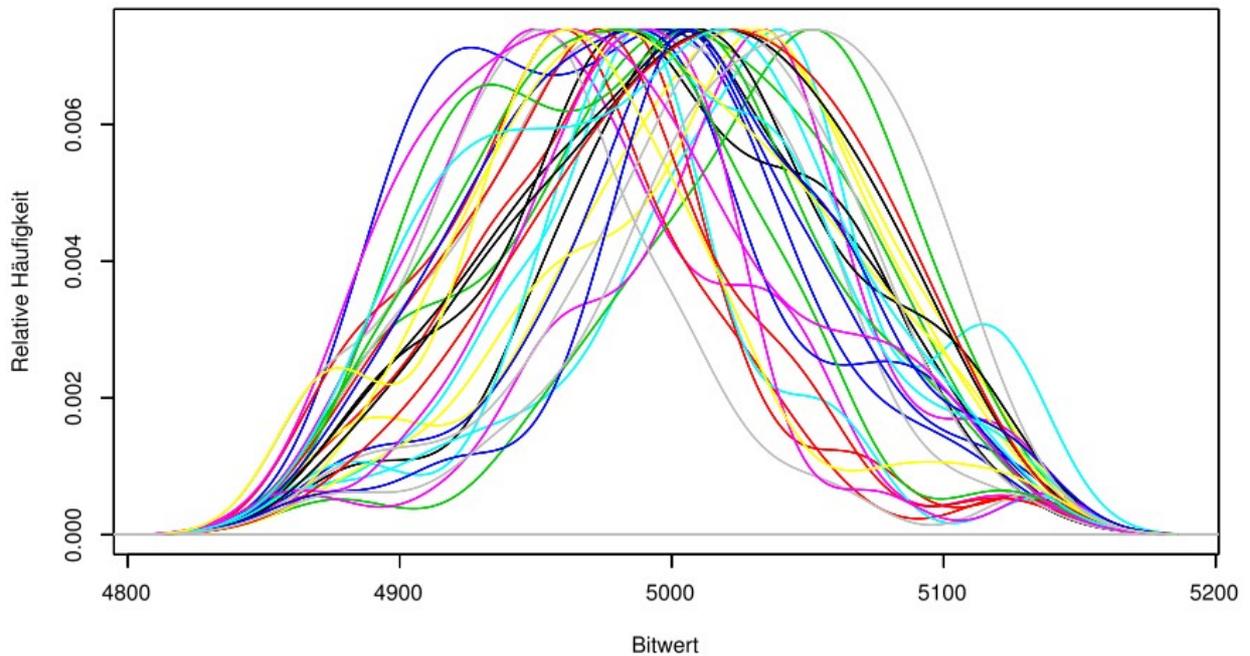


Der zweite Teil der Analyse diskutiert die Verteilungen der Bit-Summen pro Wort, also 32 Messreihen.

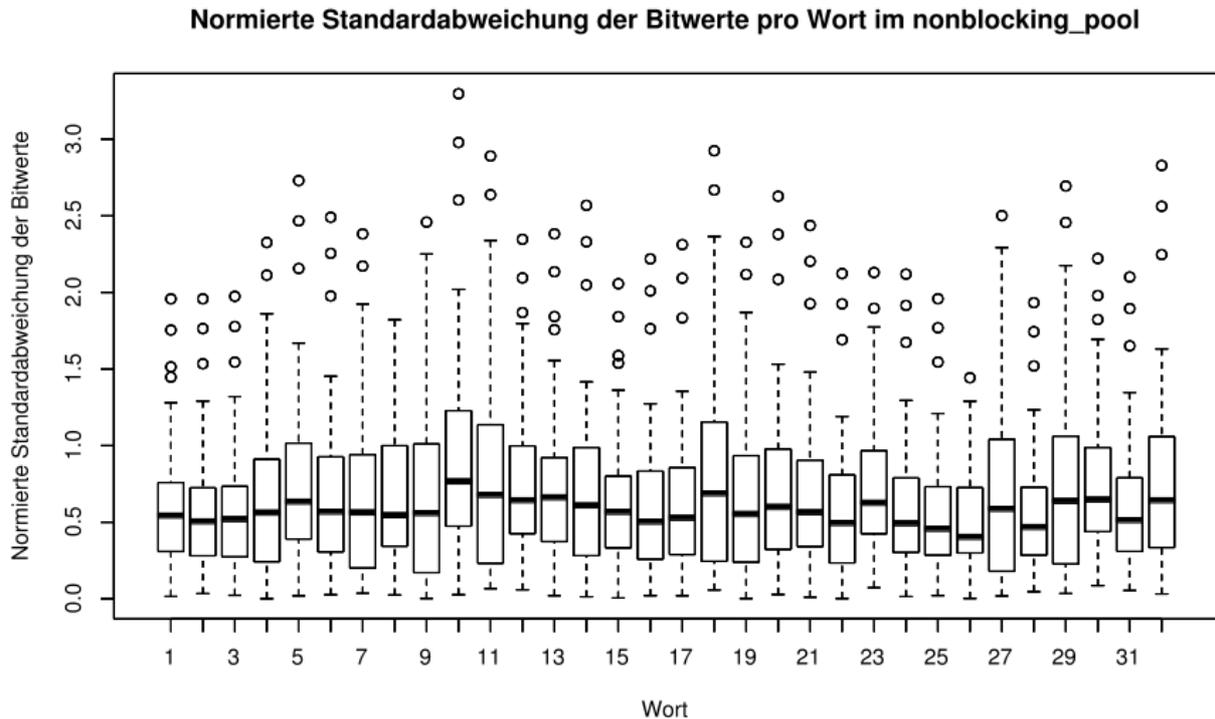
Verteilung der Anzahl der gesetzten Bits pro Wort in nonblocking_pool



Verteilung der Bitsummen pro Wort in nonblocking_pool



Abschließend folgt eine Darstellung der normierten Standardabweichungen der Bit-Summen pro Wort mittels eines Box-Whisker-Plots.



Die Verteilung der Bit-Summen der einzelnen Bitpositionen des nonblocking_pools zeigt eine deformierte Normalverteilung. Diese Deformierung könnte ihre Ursache, neben der zu kleinen Stichprobengröße, auch in den sich ähnelnden Ereignissen beim Startvorgang haben. Gegebenenfalls sollte dieser Test mit einer größeren Stichprobe wiederholt werden. Interpretiert man die Deformierung als Zeichen auf eine Ungleichverteilung der Pool-Daten, so ist es aufgrund der leichten Ausprägung praktisch als unkritisch anzusehen. Allerdings liefert es einen weiteren Hinweis auf die Notwendigkeit zum Seeding.

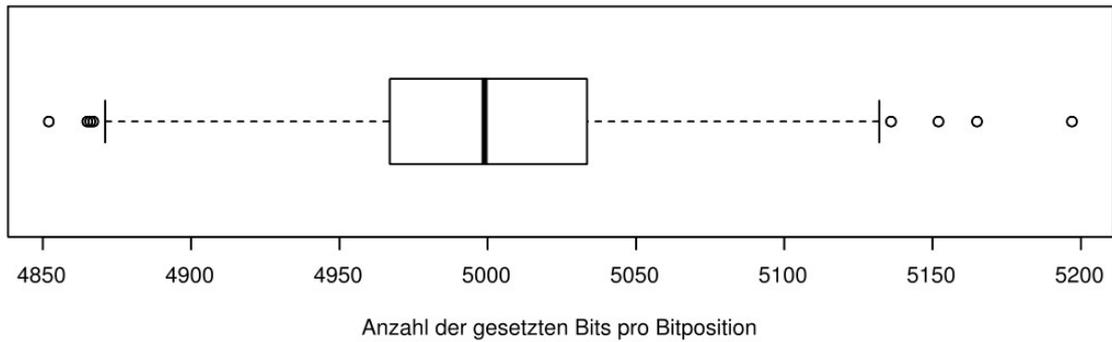
Die wortweise Betrachtung zeigt wieder teils erheblich deformierte Normalverteilungen, sowie eine vergrößerte normierte Standardabweichung. Bezüglich einer Interpretation verweisen wir auf Abschnitt 5.3.7.1. **Ein Hinweis gegen eine Wahrscheinlichkeit nahe 50% für das Gesetzsein der einzelnen Bits wurde nicht gefunden.**

5.3.8 Test 2 - Gesetzte Bits für jede Bitposition im Entropie-Pool blocking_pool zum Systemstart

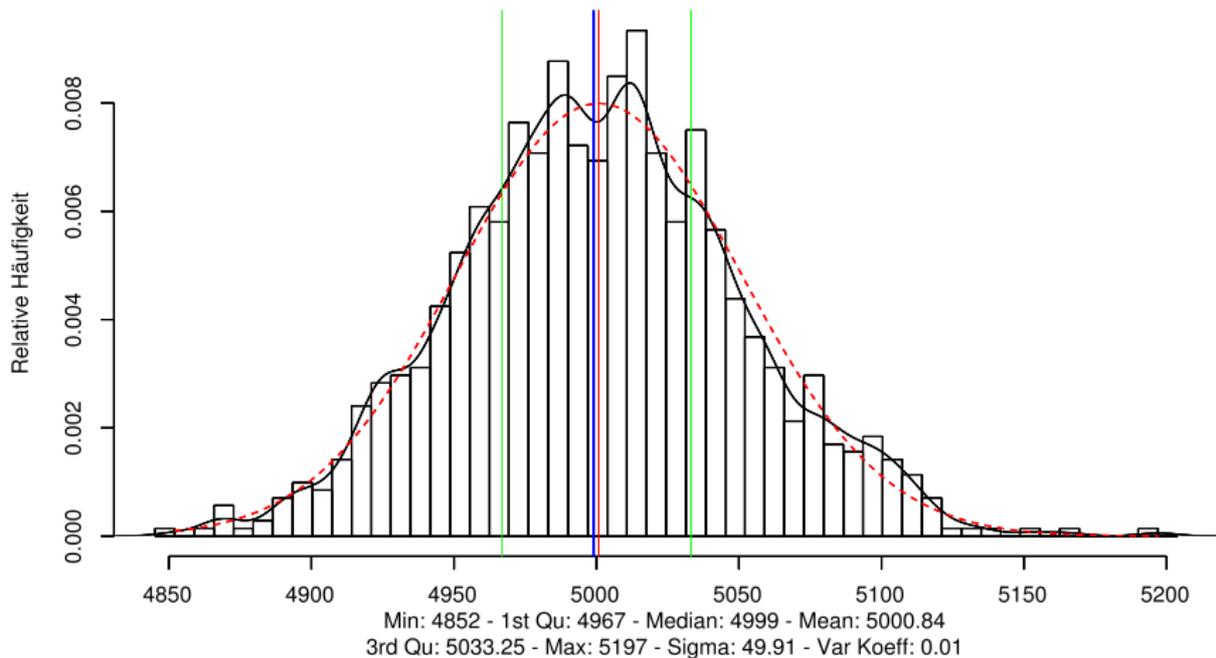
5.3.8.1 Graphische Veranschaulichung der Testresultate und Interpretation der Ergebnisse vor Start des User-Space

Der erste Teil der Analyse zeigt die Verteilung der Bit-Summen der einzelnen Bitpositionen innerhalb des gesamten Entropie-Pools.

Verteilung der Anzahl der gesetzten Bits in blocking_pool

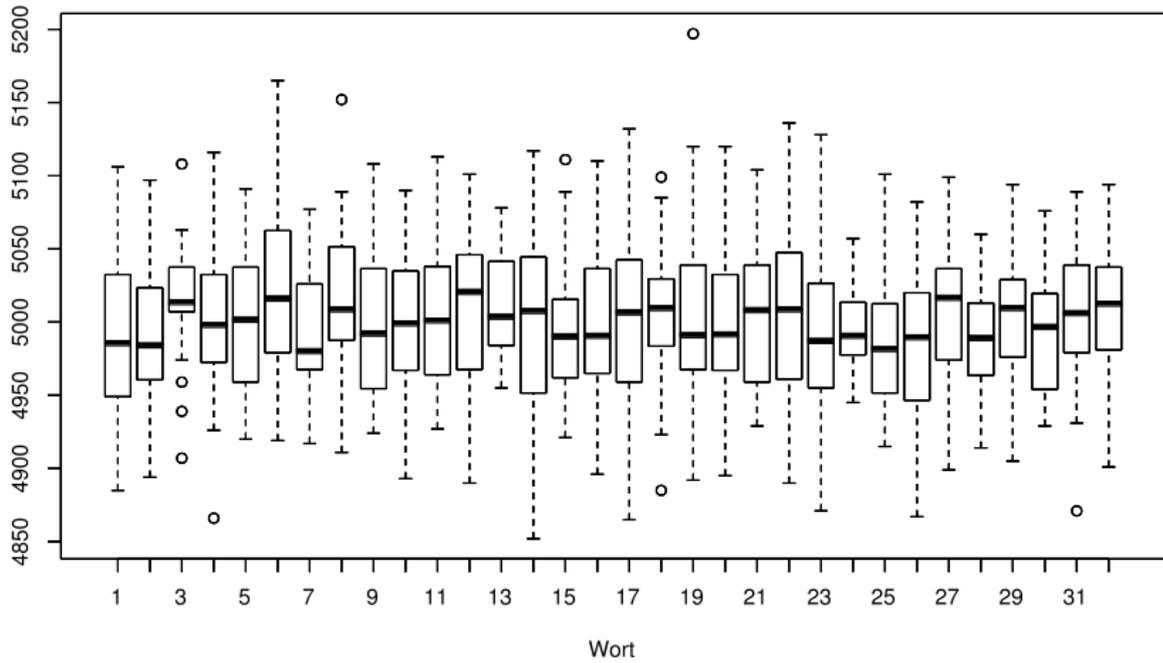


Verteilung der Anzahl der gesetzten Bits in blocking_pool

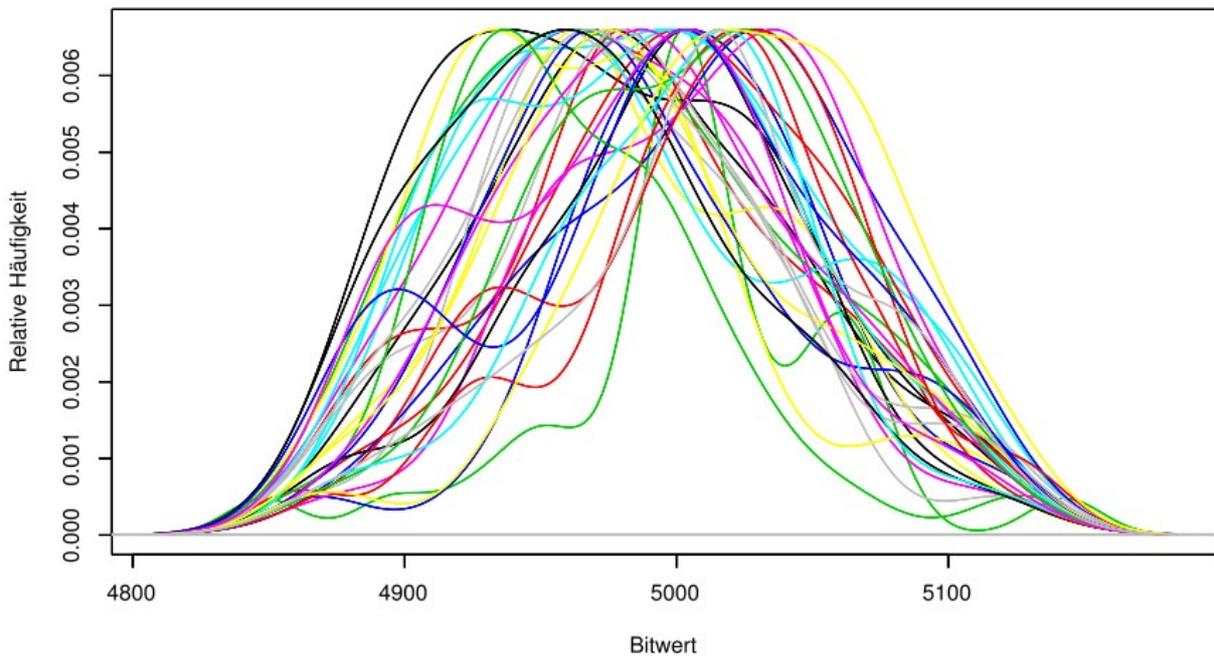


Wie zuvor ist im zweiten Analyseteil für jedes der 32 Wörter des `blocking_pool` eine eigene Messreihe von Bit-Summen für jede Bitposition zu betrachten.

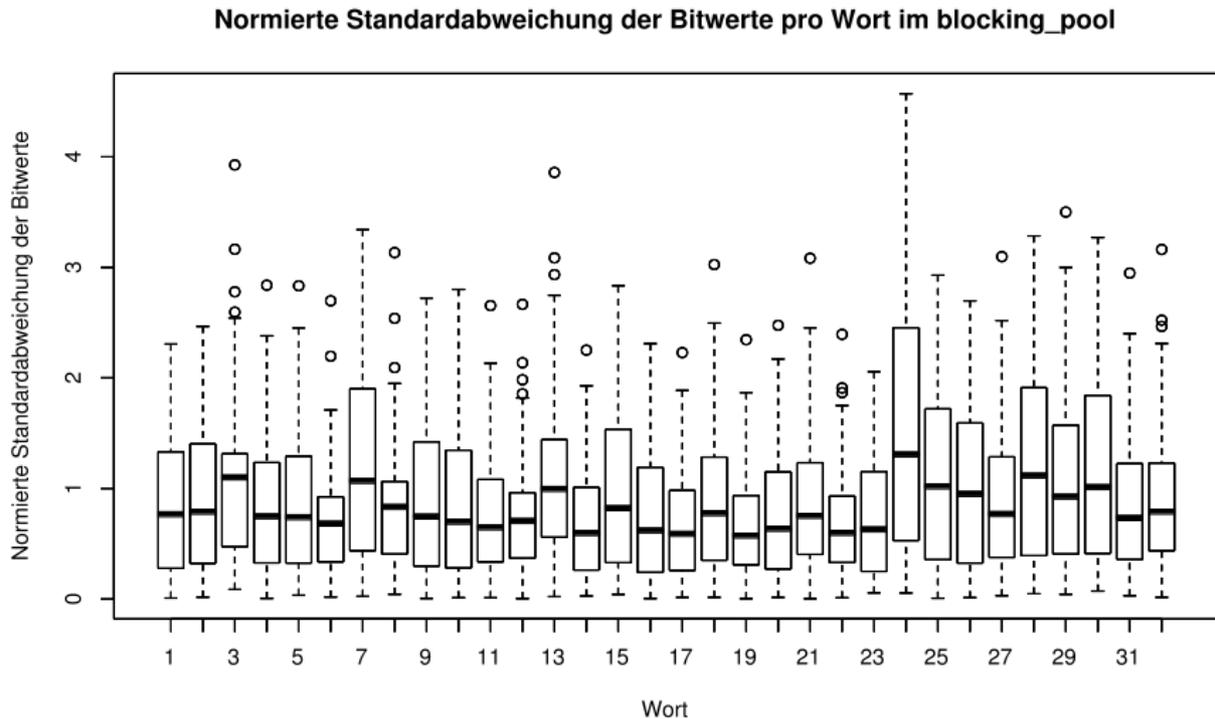
Verteilung der Anzahl der gesetzten Bits pro Wort in blocking_pool



Verteilung der Bitsummen pro Wort in blocking_pool



Auch hier folgt ein Box-Whisker-Plot mit den normierten Standardabweichungen der Bit-Summen einzelner Bitpositionen für jedes Wort.



Die Diagramme ähneln denen aus Abschnitt 5.3.7.1 sehr, sodass wir auf die Interpretation dort verweisen.

5.3.8.2 Graphische Veranschaulichung der Testresultate und Interpretation der Ergebnisse vor Hinzufügen des Seeds

Wie in 5.2.8.2 erklärt, werden Zufallszahlen aus dem `blocking_pool` nur vom User-Space via `/dev/random` ausgelesen. Da der Startvorgang bis zum Verarbeiten des Seeds keine Anfragen an `/dev/random` durchführt, wurde der Pool keinen Änderungen unterworfen.

5.4 Test 3 - Anzahl an Änderungen jeder einzelnen Bitposition

Wie in 5.1 beschrieben, wird für die Messungen im laufenden Betrieb für jede Bitposition die Anzahl der Änderungen zwischen aufeinander folgenden Zuständen bestimmt. Das lässt sich leicht durch Verwendung von XOR implementieren: Fasst man den Inhalt der Pools als n -Bit-Wort auf, so werden die Werte je zweier aufeinander folgender Messungen mit bitweisem XOR verknüpft. Ist im Ergebnis das Bit an der Stelle i gesetzt, so wurde dieses Bit geändert. Die Anzahl der gesetzten Bits für jede Bitposition der XOR-Ergebnisse entspricht also der Anzahl an Änderungen dieser Bitposition.

5.4.1 Test 3 im laufenden Betrieb - erwartete Wahrscheinlichkeiten

Im Überblick über die durchgeführten Tests in Abschnitt 5.1 wurde bereits erwähnt, dass nicht jedes Bit der Entropie-Pools beim Einmischen neuer Daten einer potentiellen Änderung unterliegt. Wir leiten jetzt für jeden Entropie-Pool separat die Anzahl an erwarteten Änderungen zwischen zwei aufeinander folgenden Zuständen her. Dabei nehmen wir an, dass ein Bit an einer bestimmten Bitposition, das potentiell geändert wird (also beim Einmischen „angefasst wird“) mit einer Wahrscheinlichkeit von 50% tatsächlich geändert wird.

5.4.1.1 `input_pool`

Entsprechend Abschnitt 2.5.1.5 wird die Datenstruktur `sample` zu dem `input_pool` hinzugefügt. Diese Datenstruktur enthält auf einem 64-Bit-System

- jiffies (long, 64 Bit),
- cycles (unsigned, 32 Bit), und
- num (unsigned, 32 Bit),

insgesamt also 16 Byte. Aufgrund der Arbeitsweise von GCC belegt die Datenstruktur ebenfalls 16 Byte Speicher unter Zugrundelegung des Padding-Mechanismus – dies kann der geneigte Leser mittels eines kleinen Programms selbst verifizieren. Die Messung erfolgt, wenn die Funktion `_mix_pool_bytes` aufgerufen wird, also nach jedem Hardware-Ereignis mit 16 Bytes. Aus Abschnitt 2.5.2 wissen wir, dass pro Byte Eingabewert, ein Wort im Entropie-Pool verändert wird. Somit werden beim Einmischen eines Hardware-Ereignisses werden 16 Wörter im `input_pool` potentiell verändert. Mit Hilfe der Annahme, dass eine potentielle Änderung in 50% der Fälle eine tatsächliche Änderung bewirkt, berechnet man schnell, dass zwischen zwei aufeinander folgende Zuständen

$$16 \cdot 32 \cdot 0.5 = 256$$

Bits verändert werden. Geht man von einer Gleichverteilung der Änderungen im Pool aus, so bedeutet das eine Änderungswahrscheinlichkeit von

$$\frac{384}{4096} = 0.0625$$

für jedes einzelne Bit pro Zustandsänderung.

Bemerkung: Aufgrund des Schieberegister-Verhalten des LRNG benötigen wir

$$E_{wrap} = \frac{W_{Entropy-Pool}}{B_{Event}}$$

Ereignisse, wobei E_{wrap} die Anzahl der Hardware-Ereignisse ist, um einmal den gesamten Entropie-Pool zu verändern, $W_{Entropy-Pool}$ die Anzahl der Wörter im Entropie-Pool ist und B_{Event} die Anzahl der Bytes, welche pro Hardware-Ereignis zu dem Entropie-Pool hinzugefügt wird, ist. Für den `input_pool` erhalten wir also

$$E_{wrap} = \frac{128}{16} = 8 \quad .$$

5.4.1.2 nonblocking_pool

Die Generierung der Beobachtungswerte wurde mit dem Befehl

```
cat /dev/urandom
```

unterstützt. Wenn wir vernachlässigen, dass Daten aus dem `input_pool` in den `nonblocking_pool` transferiert werden²², wird der `nonblocking_pool` nur noch durch den berechneten SHA-1-Wert über den `nonblocking_pool` verändert. Entsprechend Abschnitt 2.6.1 implementiert die Funktion `extract_buf` das Berechnen der Zufallszahl. Diese Funktion ruft die Mischfunktion `_mix_pool_bytes` mit einem Eingabewert 20 Bytes auf, da die Variable `hash` aus 5 Wörtern à 32 Bit besteht. Analog zu 5.4.1.1 wird beim Einmischen eines Bytes in den Entropie-Pool ein 32-Bit-Wort potentiell geändert. Somit ergibt sich eine erwartete Veränderung von

$$20 \cdot 32 \cdot 0.5 = 320$$

Bits zwischen zwei aufeinander folgenden Zuständen. Geht man wieder von einer Gleichverteilung der Änderungen im Pool aus, so bedeutet das eine Änderungswahrscheinlichkeit von

$$\frac{320}{1024} = 0.3125$$

für jedes einzelne Bit pro Zustandsänderung.

²² Dies ist eine valide Annahme, da der Befehl die Entropie im `input_pool` schnell leert.

Bemerkung: Da der `nonblocking_pool` 32 Wörter umfasst, erhält man analog zu Abschnitt 5.4.1.1

$$E_{wrap} = \frac{32}{20} = 1,6 \quad .$$

5.4.1.3 blocking_pool

Für den `blocking_pool` wurde die Generierung der Beobachtungswerte wurde mit dem Befehl

```
cat /dev/zero > /dev/random
```

unterstützt. Da die Testumgebung kein Lesen von `/dev/random` in der Zeit der Messungen durchführt, wird der `blocking_pool` nur durch das Schreiben verändert. Entsprechend Abschnitt 2.5.1.8 werden die nach `/dev/random` geschriebenen Daten in Blöcken von 64 Bytes zu dem `blocking_pool` hinzugefügt, denn über die Funktion `write_pool` wird die Funktion `_mix_pool_bytes` zum Einmischen der Variable `buf`, die aus 16 Wörtern à 32 Bit besteht, aufgerufen. Da der `blocking_pool` jedoch nur aus 32 Wörtern besteht, werden hier alle 32 Wörter potentiell geändert. Hier erwartet man also eine Änderung von

$$32 \cdot 32 \cdot 0,5 = 512$$

Bits zwischen zwei Zuständen. Zusammen mit der Annahme der Gleichverteilung der Änderungen im Pool ergibt sich eine Änderungswahrscheinlichkeit von

$$\frac{512}{1024} = 0,5$$

für jedes einzelne Bit pro Zustandsänderung.

Bemerkung: Die Formel am Ende von 5.4.1.1 liefert, unter Beachtung, dass der `blocking_pool` 32 Wörter umfasst, folgenden Wert:

$$E_{wrap} = \frac{32}{64} = 0,5 \quad .$$

Hier ist jedoch zu beachten, dass ein Wert E_{wrap} von unter eins bedeutungslos ist, da dieser Wert aussagt, dass mit einem Aufruf, alle Wörter mehr als einmal Ziel einer Veränderung sind. Es reicht aber eine Veränderung aus. Damit gilt grundsätzlich und hier insbesondere

$$\lfloor E_{wrap} \rfloor = 1 \quad .$$

5.4.2 Test 3 im laufenden Betrieb - Testansatz

Die folgenden Dateien aus dem Verzeichnis `entropy_pool_distribution` wurden für diesen Test verwendet:

- Das SystemTap-Skript `entropy_pool_distribution.stp` bestimmt die Anzahl der Veränderungen für jede Bitposition. Hier kann die Stichprobenanzahl mit der Variable `num_samplings` angepasst werden.
- Über das Bash-Skript `gendata.sh` wird das o.g. SystemTap-Skript aufgerufen.
- Das R-Project-Analyseprogramm `entropy_pool_distribution.r` erstellt die nachfolgenden Graphiken aus der mittels dem SystemTap-Skript erzeugten Datenreihe.

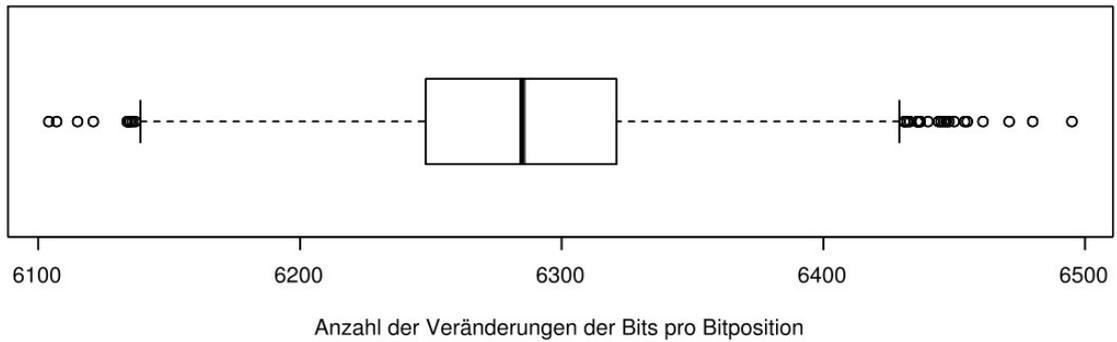
Die gewählte Stichprobengröße beträgt $S = 100.000$.

5.4.3 Test 3 - Veränderungen pro Bitposition im Entropie-Pool input_pool

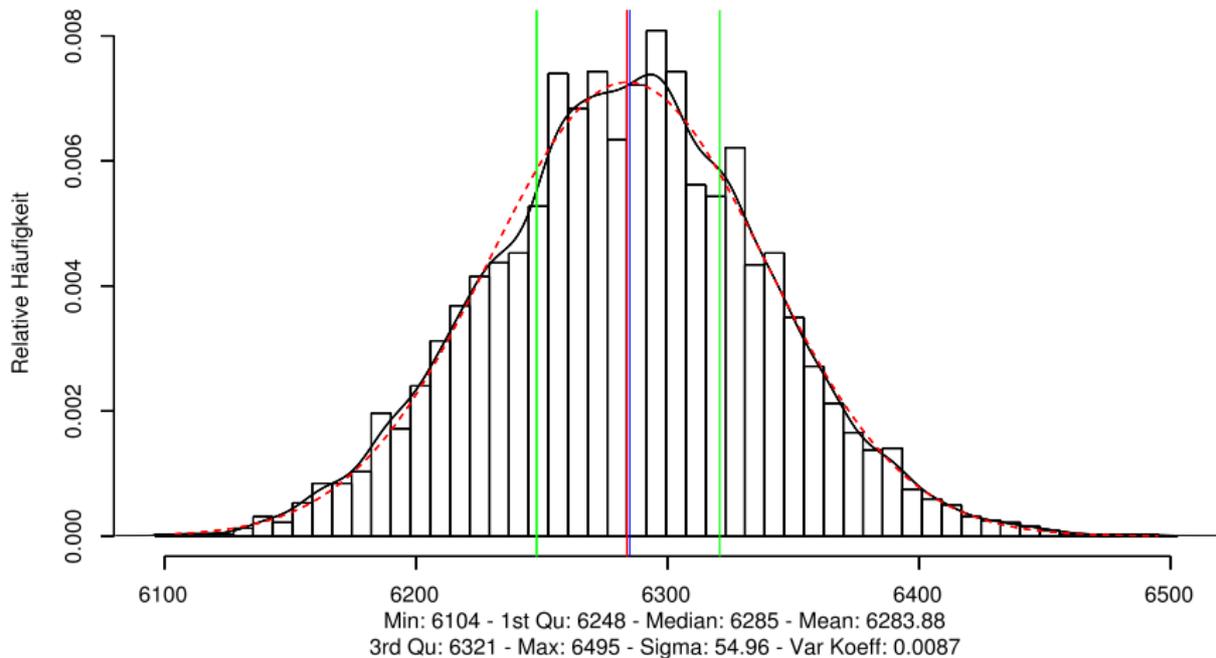
Wie in Abschnitt 5.2.2 angesprochen, erscheint eine zusätzliche Betrachtung, wie oft jedes einzelne Bit bei Veränderungen des Entropie-Pools geändert wird, als sinnvoll.

5.4.3.1 Graphische Veranschaulichung der Testresultate und Interpretation der Ergebnisse

Verteilung der Anzahl der Veränderungen der Bits pro Bitposition in input_pool

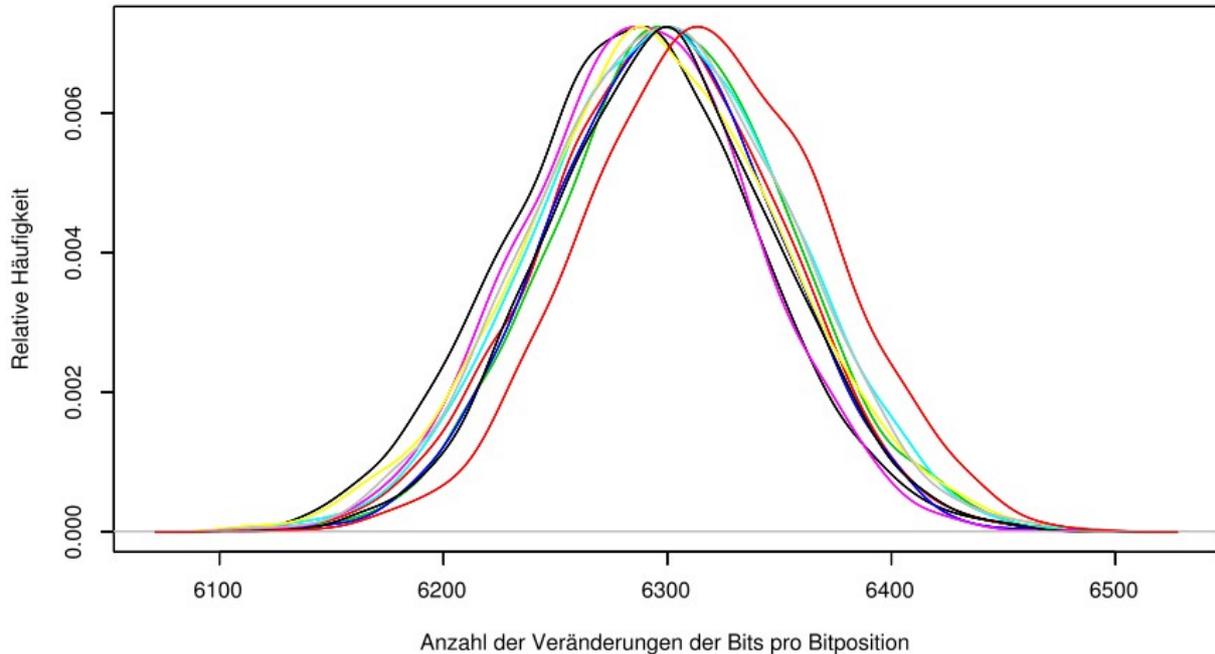


Verteilung der Anzahl der Veränderungen der Bits pro Bitposition in input_pool



Das Verhalten der Verteilung über die Zeit wurde mit 10 Testreihen mit der oben angegebenen Stichprobengröße durchgeführt. Die schwarze Kurve entspricht der ersten Testreihe, die obigem Box-Whisker-Plot und dem zugehörigen Histogramm zugrunde liegt.

Verteilung der Anzahl der Veränderungen der Bits pro Bitposition in input_pool



Zieht man die Testbeschreibung aus Abschnitt 5.1 und die erwartete Änderungswahrscheinlichkeit für jedes Bit aus Abschnitt 5.4.1.1 in Erwägung, so erwartet man bei der oben gewählten Stichprobengröße den Mittelwert/Median der Änderungszahl bei

$$100.000 \cdot 0.0625 = 6250 \quad .$$

Somit lässt sich zusammenfassen:

- Mittelwert und Median sind relativ nahe beieinander und liegen sehr nahe der theoretischen Vorhersage.
- Das Histogramm zeigt eine fast perfekte Normalverteilung, obwohl leichte Einbrüche bei den Säulen im Histogramm zu sehen sind. Der Grund dieser Einbrüche ist derzeit Unbekannt.
- Die Streuung der Werte um den Mittelwert ist relativ gering, wie im Histogramm und im Box-Whisker-Plot ersichtlich.
- Die Veränderung der Verteilung beim Wiederholen des Tests ist ebenfalls sehr gering.
- Der Variationskoeffizient zeigt, dass die Verteilung sehr „schmal“ im Verhältnis der Gesamtdaten ist.

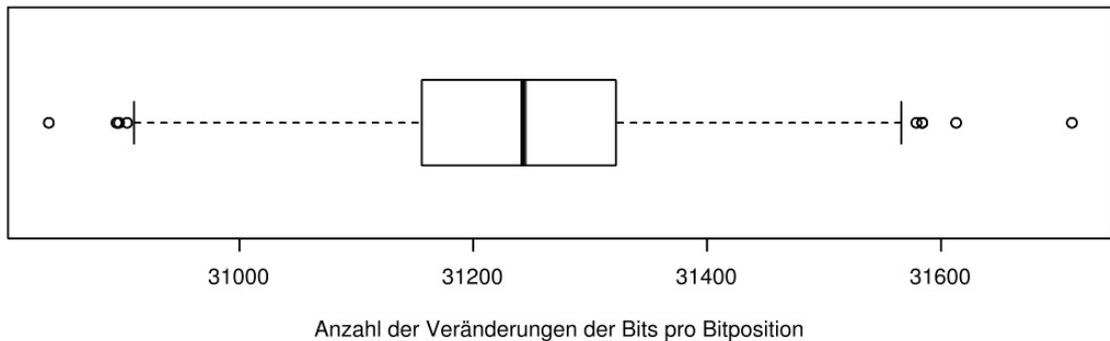
Die Ergebnisse sprechen für eine Gleichverteilung der Veränderung der Daten im input_pool und damit für die Qualität des LRNG.

5.4.4 Test 3 - Veränderungen pro Bitposition im Entropie-Pool nonblocking_pool

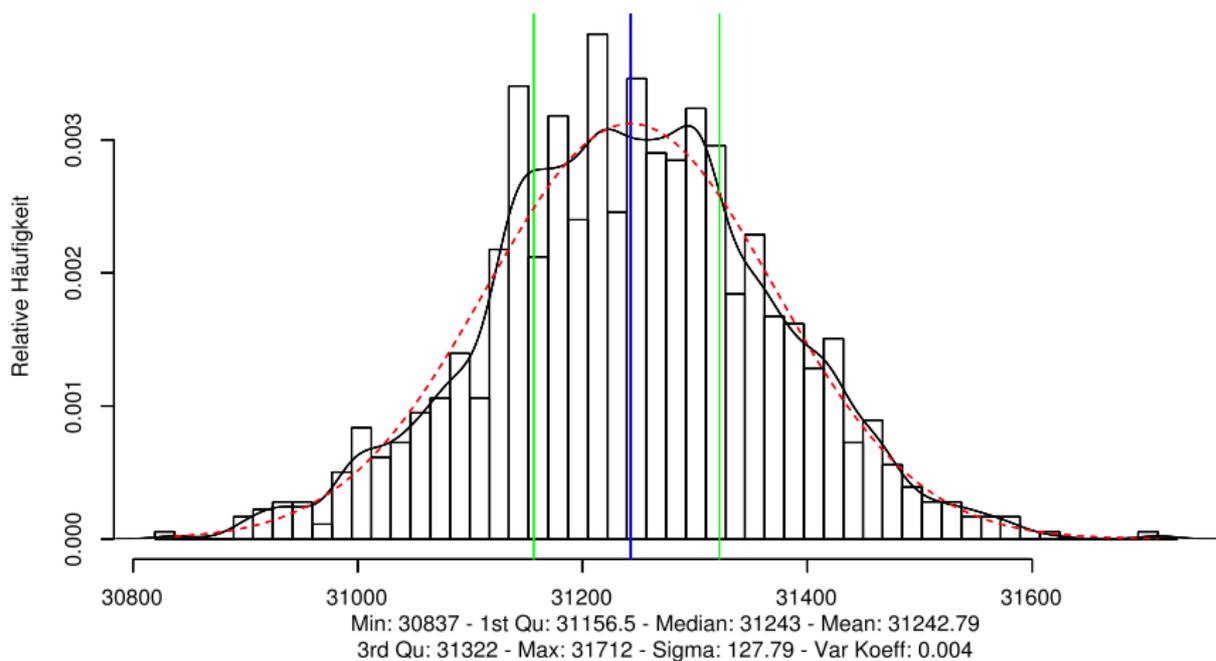
Ebenso wie in Abschnitt 5.2.3 verzichten wir hier auf eine gesonderte Betrachtung von ausschließlich Lese- oder Schreibvorgängen auf den Gerätedateien.

5.4.4.1 Graphische Veranschaulichung der Testresultate und Interpretation der Ergebnisse

Verteilung der Anzahl der Veränderungen der Bits pro Bitposition in nonblocking_pool



Verteilung der Anzahl der Veränderungen der Bits pro Bitposition in nonblocking_pool



Auch hier wird aufgrund der Testbeschreibung aus Abschnitt 5.1 und der Berechnung in 5.4.1.2 der erwartete Mittelwert/Median bei der oben gewählten Stichprobengröße bestimmt:

$$100.000 \cdot 0.3125 = 31250 \quad .$$

Somit gilt, dass

- Mittelwert und Median-Wert fast identisch und sehr nahe der theoretischen Vorhersage liegen,
- das Histogramm eine deformierte Normalverteilung zeigt,
- die Streuung der Werte um den Mittelwert ist sehr gering ist und
- der Variationskoeffizient eine sehr „schmale“ Verteilung im Verhältnis der Gesamtdaten anzeigt.

Die Ursache dieser Deformierung der Verteilung ist unklar. Dennoch sollte man sich vor einer Interpretation die Breite der Verteilung im Verhältnis zur theoretischen Breite der Beobachtungen vor Augen führen: der Variationskoeffizient zeigt, dass die Verteilung äußerst „schmal“ ist und damit kann diese leichte Deformation als nicht signifikant betrachtet werden.

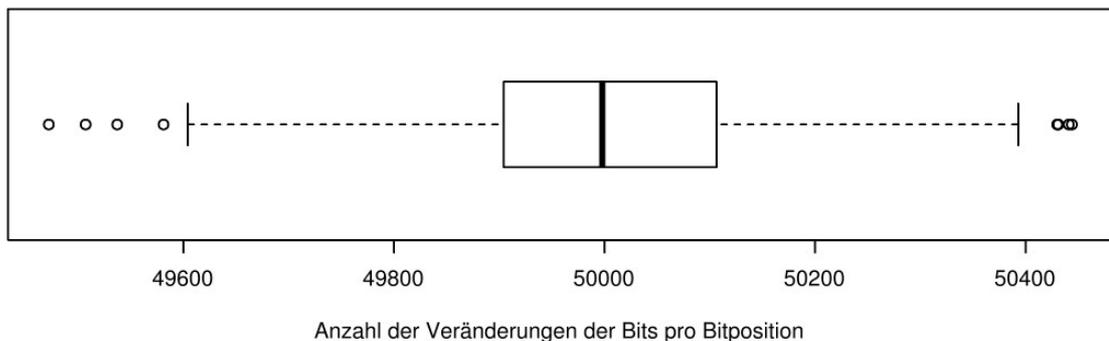
Die Ergebnisse mit den angesprochenen Deformationen sind Hinweise auf eine nicht mehr vorhandene Gleichverteilung der Veränderung der Daten im nonblocking_pool. Weitere Analysen die den Hintergrund der Deformationen beleuchten sind notwendig.

5.4.5 Test 3 - Veränderungen pro Bitposition im Entropie-Pool blocking_pool

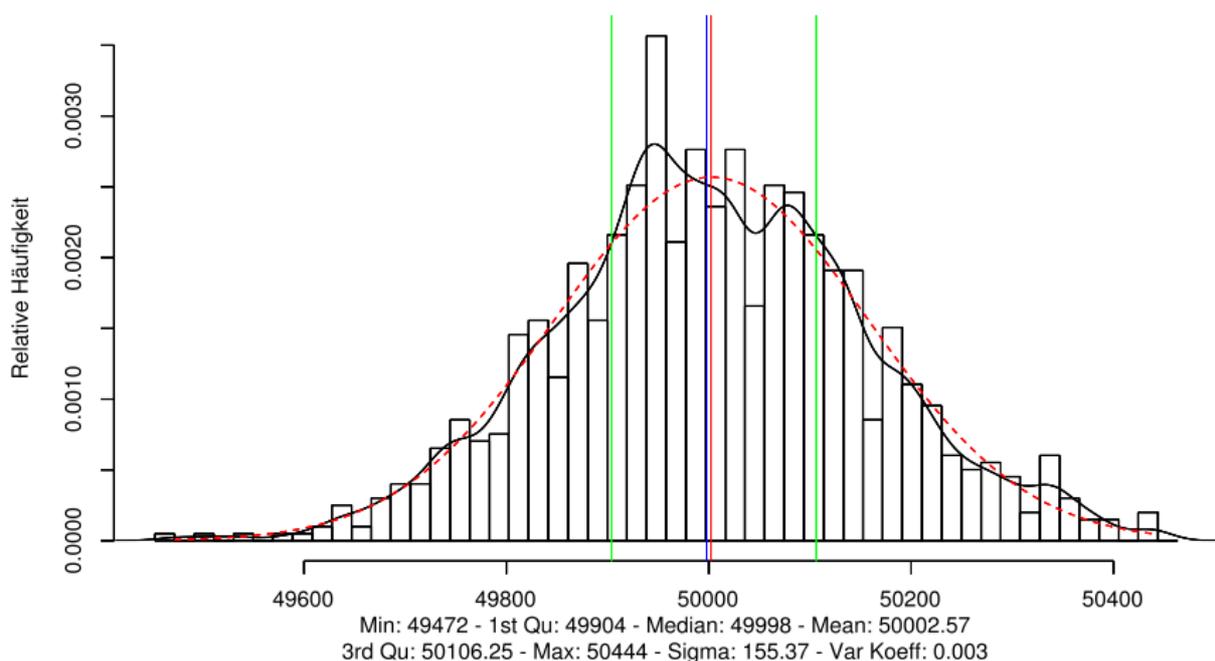
Wie in Abschnitt 5.2.4 erklärt, wird eine gesonderte Betrachtung von ausschließlich Lese- oder Schreibvorgängen auf den Gerätedateien nicht als notwendig erachtet.

5.4.5.1 Graphische Veranschaulichung der Testresultate und Interpretation der Ergebnisse

Verteilung der Anzahl der Veränderungen der Bits pro Bitposition in blocking_pool



Verteilung der Anzahl der Veränderungen der Bits pro Bitposition in blocking_pool



Wieder berechnen wir mit Hilfe der zuvor bestimmten Änderungswahrscheinlichkeit für jede Bitposition (siehe 5.4.1.3) und der oben angegebenen Stichprobengröße den erwarteten Mittelwert/Median an Änderungen:

$$100.000 \cdot 0,5 = 50.000 \quad .$$

Analog zum vorherigen Abschnitt gilt, dass

- Mittelwert und Median-Wert fast identisch und sehr nahe der theoretischen Vorhersage liegen,
- das Histogramm eine deformierte Normalverteilung zeigt, in dem Einbrüche bei den Säulen im Histogramm zu sehen sind – Grund derzeit unbekannt –,
- die Streuung der Werte um den Mittelwert ist sehr gering ist und
- der Variationskoeffizient eine sehr „schmale“ Verteilung im Verhältnis der Gesamtdaten anzeigt.

Die Ergebnisse mit den angesprochenen Deformationen sind Hinweise auf eine nicht mehr vorhandene Gleichverteilung der Veränderung der Daten im `nonblocking_pool`. Weitere Analysen die den Hintergrund der Deformationen beleuchten sind notwendig.

5.4.6 Test 3 zum Systemstart - Vorüberlegungen und Testansatz

5.4.6.1 Vorüberlegungen und Testansatz

Die Analysen zu Test 3 zum Systemstart nutzen die gleichen Messdaten wie Test 1. Somit gelten die gleichen Vorüberlegungen und der gleiche Testansatz wie in Abschnitt 5.2.5.

Jede Analyse ist zweigeteilt und entspricht immer dem in Abschnitt 5.2.5.2 dargestellten Ansatz mit einer Ausnahme. So werden in Abschnitt 5.2.5.2 die Bit-Summen für jedes Wort einzeln ermittelt und entsprechend analysiert. Demgegenüber entspricht bei der Analyse in diesem Test die Anzahl der Beobachtungen der Anzahl der Pool-Wörter.

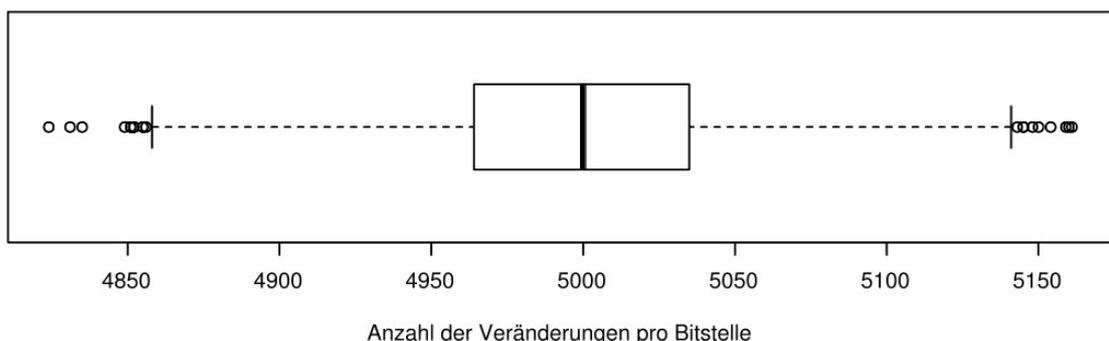
Die gewählte Stichprobengröße ergibt sich aus Abschnitt 5.2.5.2. Wir erinnern daran, dass wie in 5.2.5 jeder nachfolgende Unterabschnitt sechs Analysen enthält.

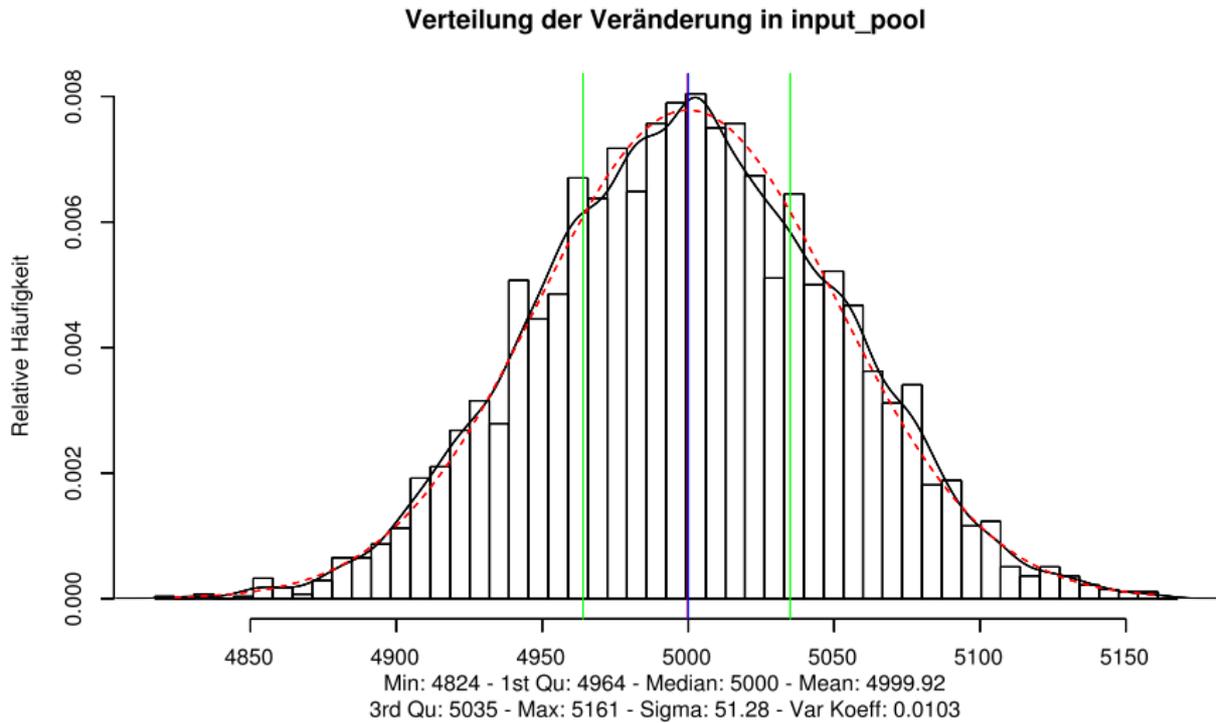
5.4.7 Test 3 - Veränderungen pro Bitposition im Entropie-Pool `input_pool` zum Systemstart

5.4.7.1 Graphische Veranschaulichung der Testresultate und Interpretation der Ergebnisse vor Start des User-Space

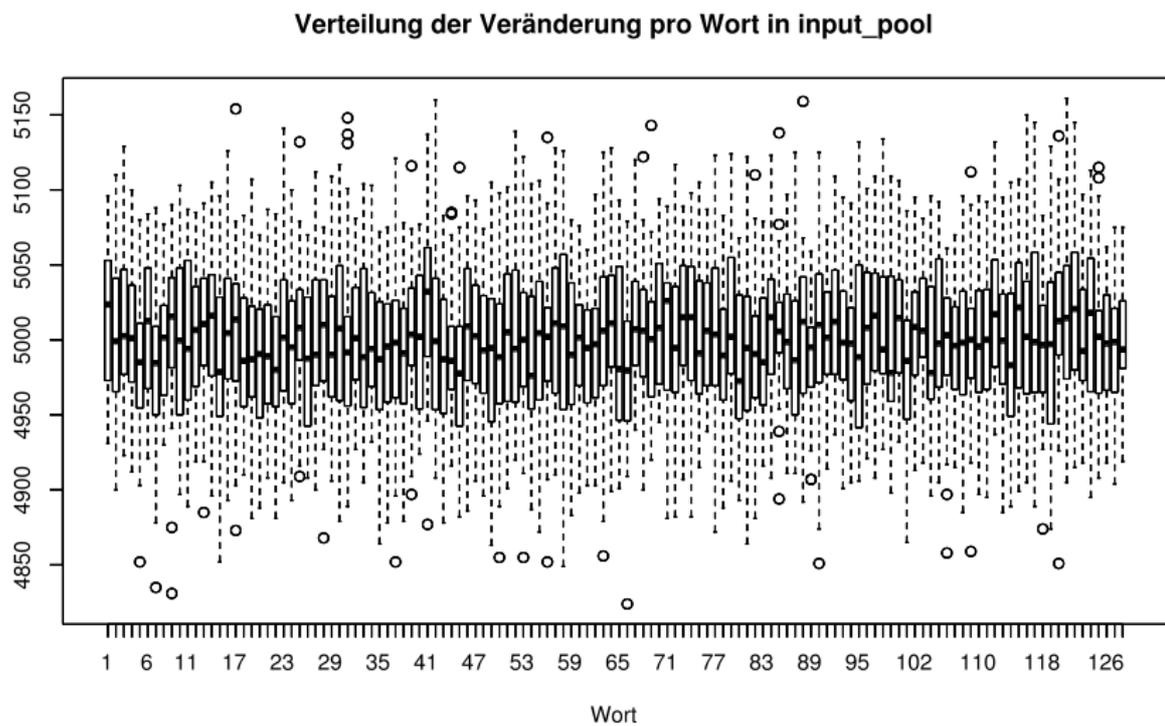
Der erste Teil der Analyse zeigt die Verteilung der Anzahl an Veränderungen für jeden Bitposition innerhalb des gesamten Entropie-Pools.

Verteilung der Veränderung in `input_pool`

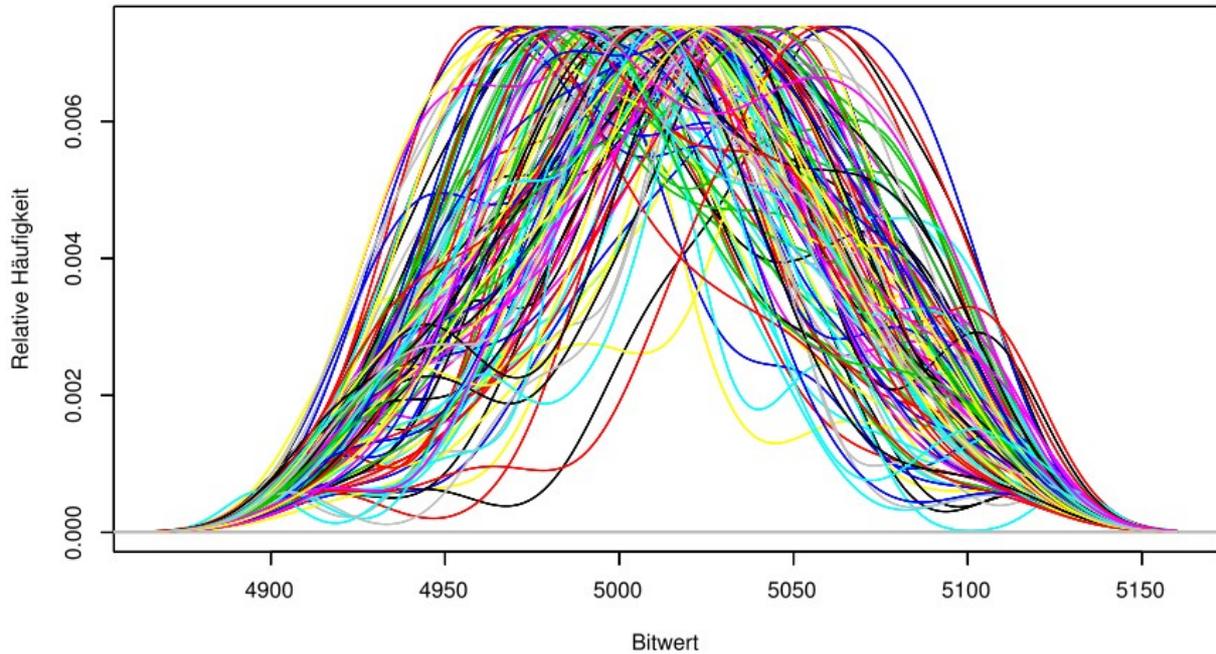




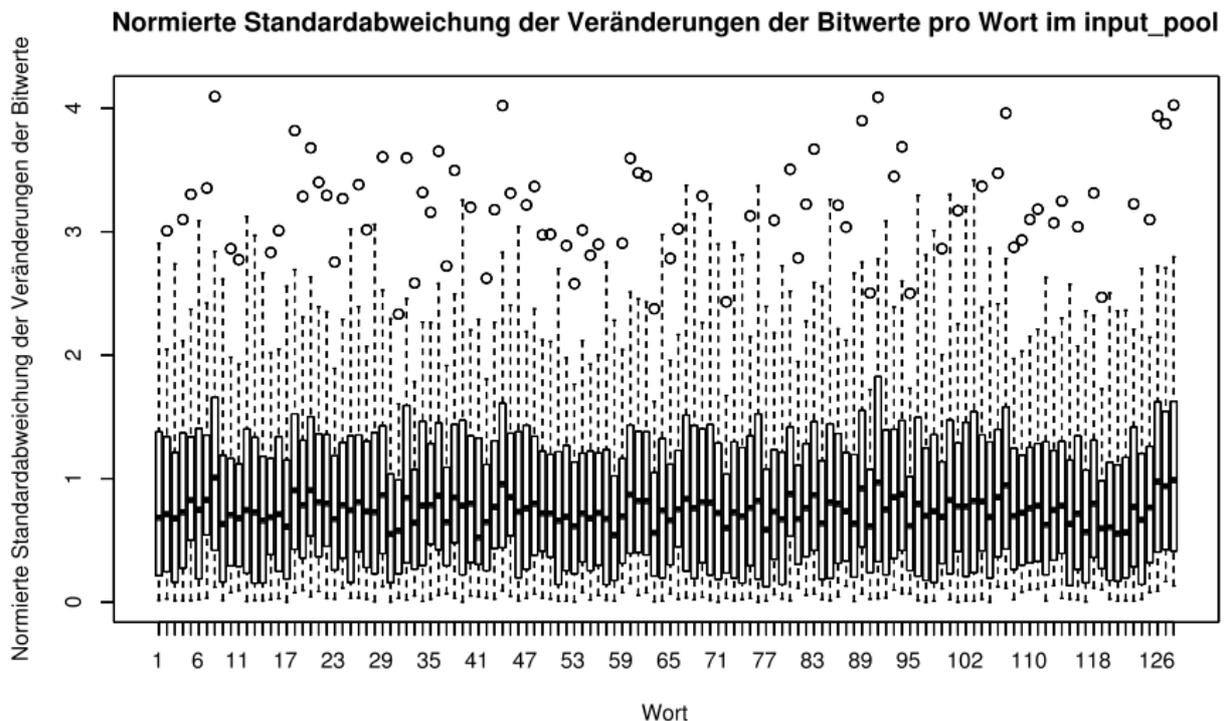
Der zweite Teil der Analyse diskutiert die Verteilungen der Anzahl an Änderungen für jede Bitposition für die einzelnen Wörter. Wie bereits beschrieben, enthält `input_pool` 128 Wörter à 32 Bit, weshalb 128 separate Messreihen betrachtet werden.



Verteilung der Veränderung der Bitsummen pro Wort im input_pool



Abschließend wird ein weiterer Box-Whisker Plot angezeigt, welcher die normierte Standardabweichung der Änderungen pro Bitposition für jedes Wort darstellt.



Die Verteilung der Änderungen einzelner Bitpositionen des Entropie-Pools entsprechen vollkommen den Erwartungen, insbesondere fallen Mittelwert und Median genau mit der Vorhersage $S/2$ zusammen.

Bei der wortweisen Betrachtung fällt die relativ starke Schwankungsbreite der Verteilungen aller Wörter auf. Es ist auch zu sehen, dass die Verteilung einzelner Wörter stark von einer Normalverteilung abweicht, indem sie beispielsweise zwei lokale Maxima besitzt.

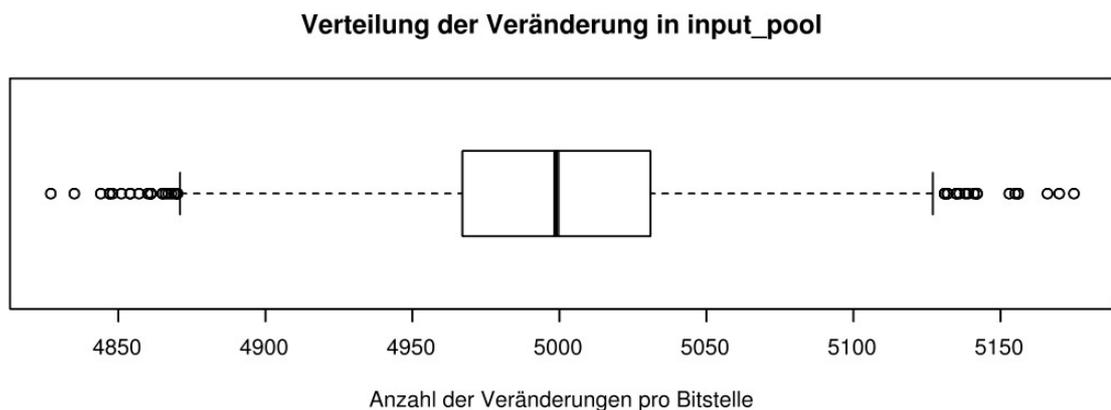
Diesen Unterschied in den Dichteverteilungen kann man auch in dem Graphen der normierten Standardabweichungen sehen: die Box-Plots unterscheiden sich, der Medianwert der verschiedenen Wörter schwankt. Es zeigen sich aber keine signifikanten Ausreißer. Alle Box-Whisker-Plots der Standardabweichungen haben einen Bias in Richtung Null. Das bedeutet, dass die Whisker Richtung Null kleiner sind, als die Whisker in die andere Richtung. Weiterhin ist der Medianwert in der Box in Richtung Null verschoben. Dieses Ergebnis ist bei einer Normalverteilung zu erwarten.

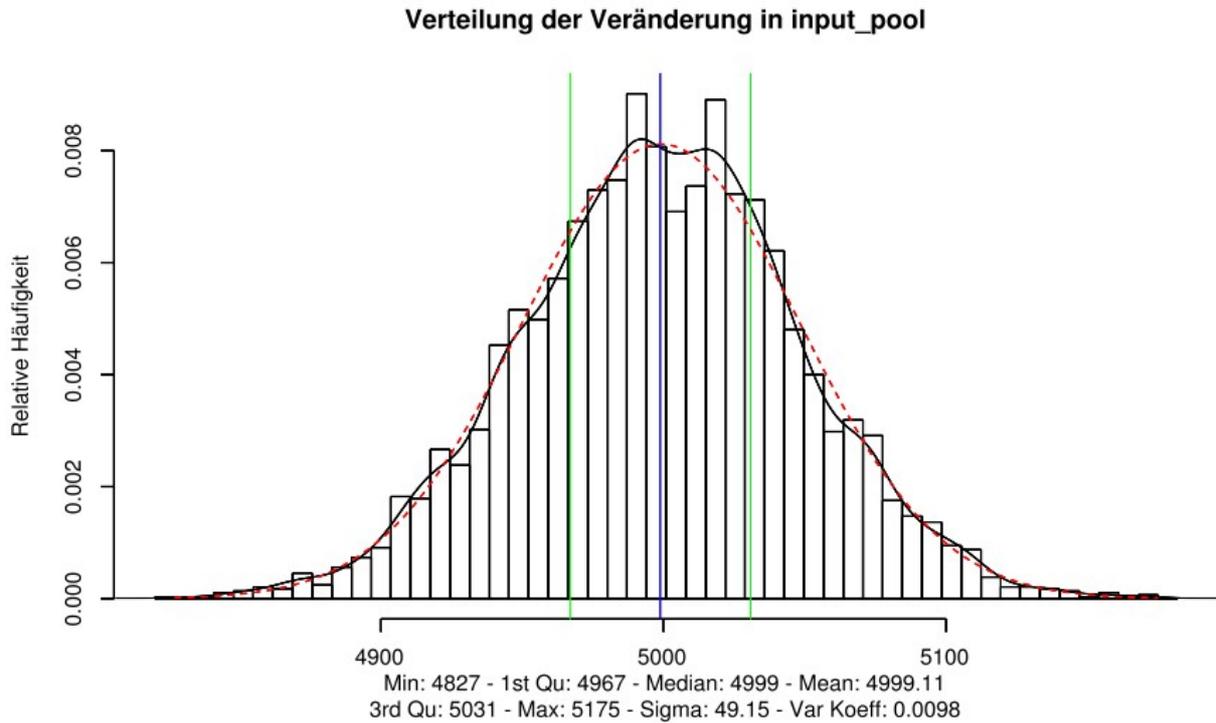
Somit scheinen die wortweisen Verteilungen der Änderungen pro Bitposition nicht immer einer Normalverteilung zu entsprechen. Die Abweichungen werden allerdings noch als unkritisch angesehen, zudem gleichen sich die Unterschiede bei Betrachtung aller Wörter aus.

Hinweise gegen die Annahme, dass jedes Bit in etwa 50% der Fälle geändert wird, wurden nicht gefunden.

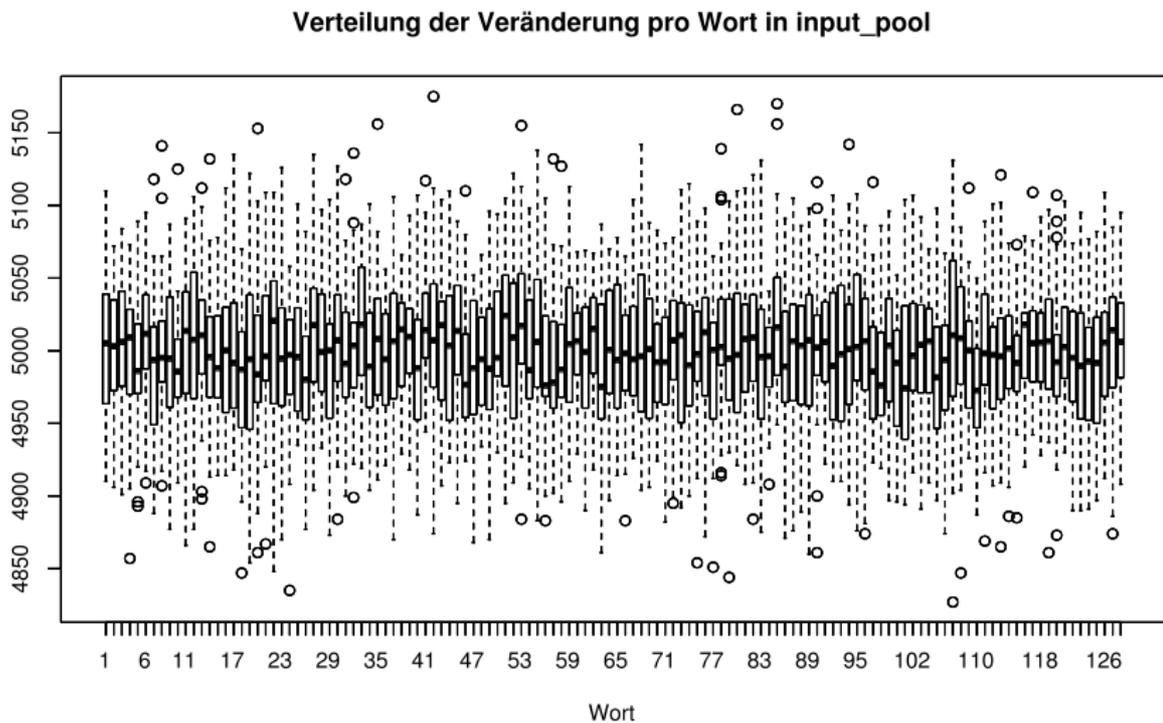
5.4.7.2 Graphische Veranschaulichung der Testresultate und Interpretation der Ergebnisse vor Hinzufügen des Seeds

Der erste Teil der Analyse zeigt die Verteilung der Anzahl an Veränderungen für jede Bitposition innerhalb des gesamten Entropie-Pools.

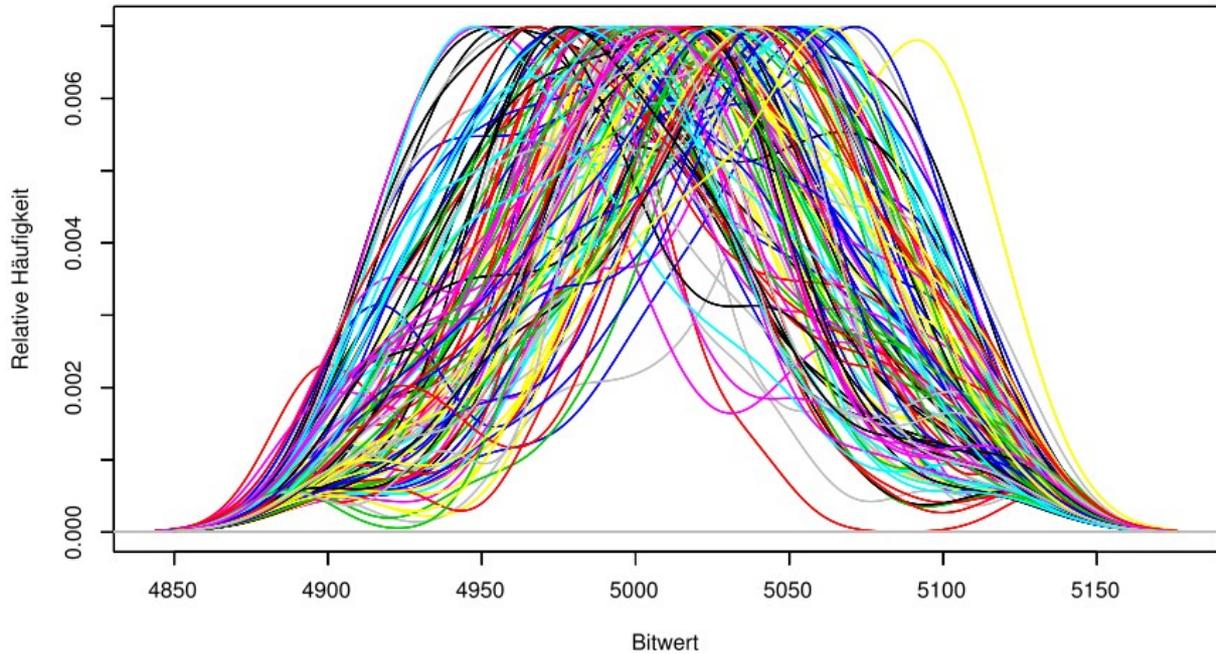




Der zweite Teil der Analyse diskutiert die Verteilungen der Anzahl an Änderungen an jeder einzelnen Bitposition für jedes der 128 Wörter.

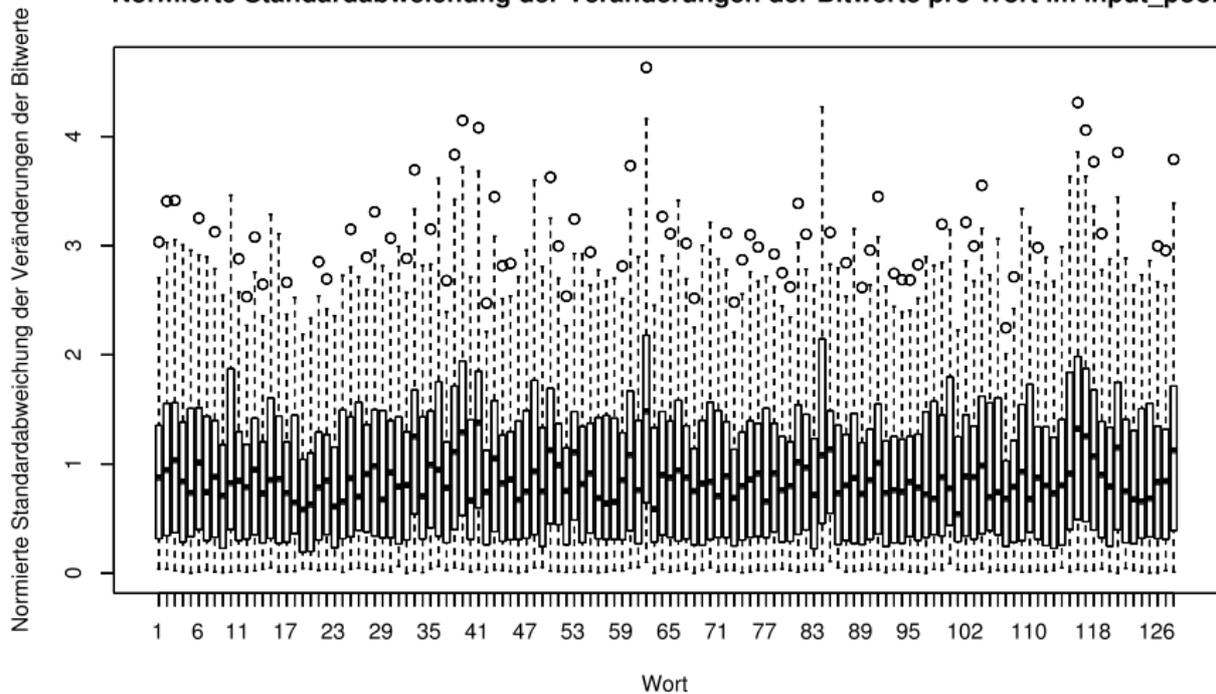


Verteilung der Veränderung der Bitsummen pro Wort im input_pool



Auch hier folgt ein Box-Whisker-Plot mit den normierten Standardabweichungen der Änderungen einzelner Bitpositionen für jedes Wort.

Normierte Standardabweichung der Veränderungen der Bitwerte pro Wort im input_pool



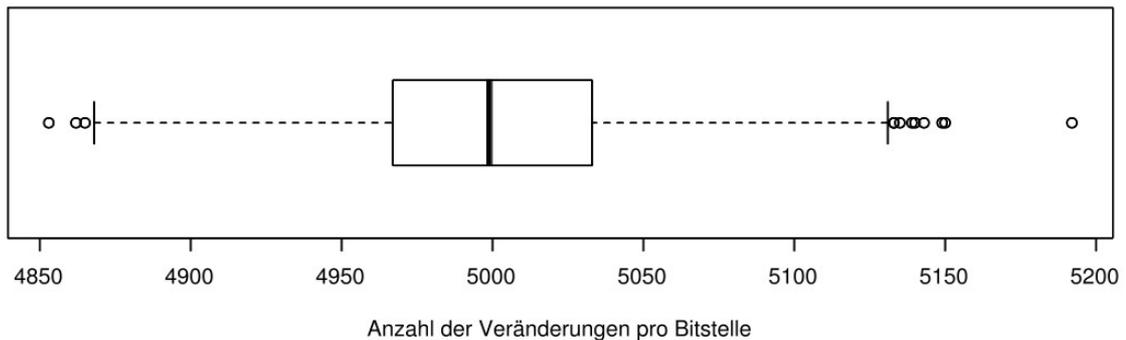
Die Ergebnisse ähneln denen des vorangegangenen Abschnitts 5.4.7.1 sehr, so dass wir auf die Interpretation dort verweisen.

5.4.8 Test 3 - Veränderungen pro Bitposition im Entropie-Pool nonblocking_pool zum Systemstart

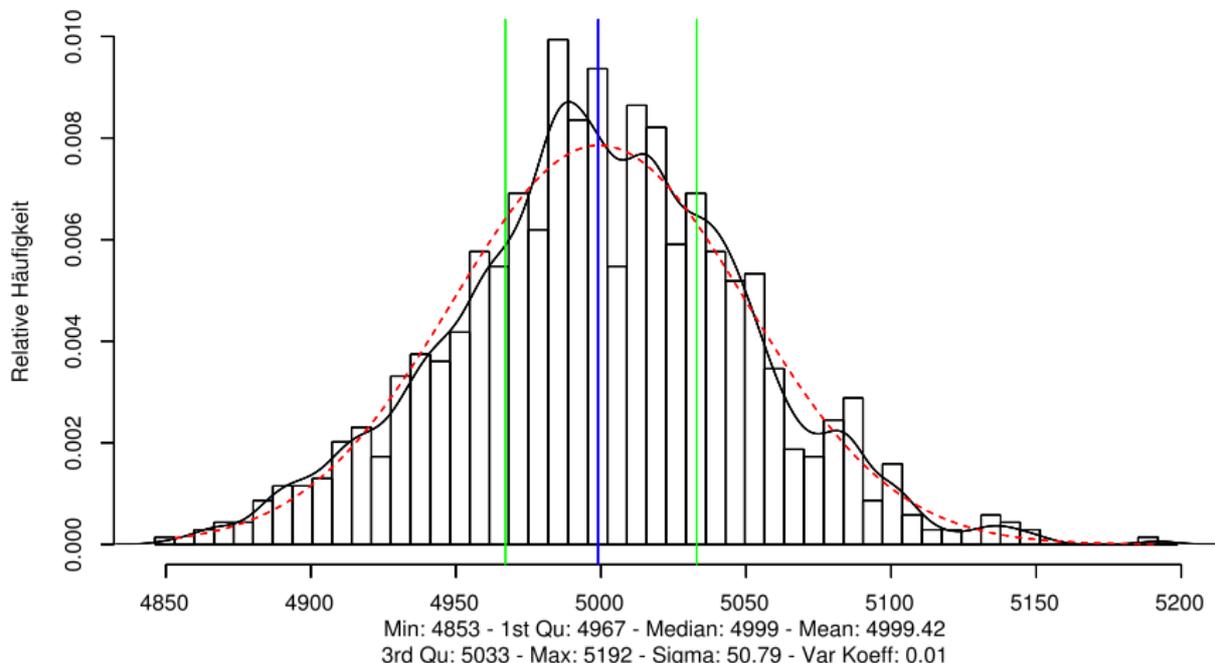
5.4.8.1 Graphische Veranschaulichung der Testresultate und Interpretation der Ergebnisse vor Start des User-Space

Der erste Teil der Analyse zeigt die Verteilung der Anzahl an Veränderungen für jede Bitposition innerhalb des gesamten Entropie-Pools.

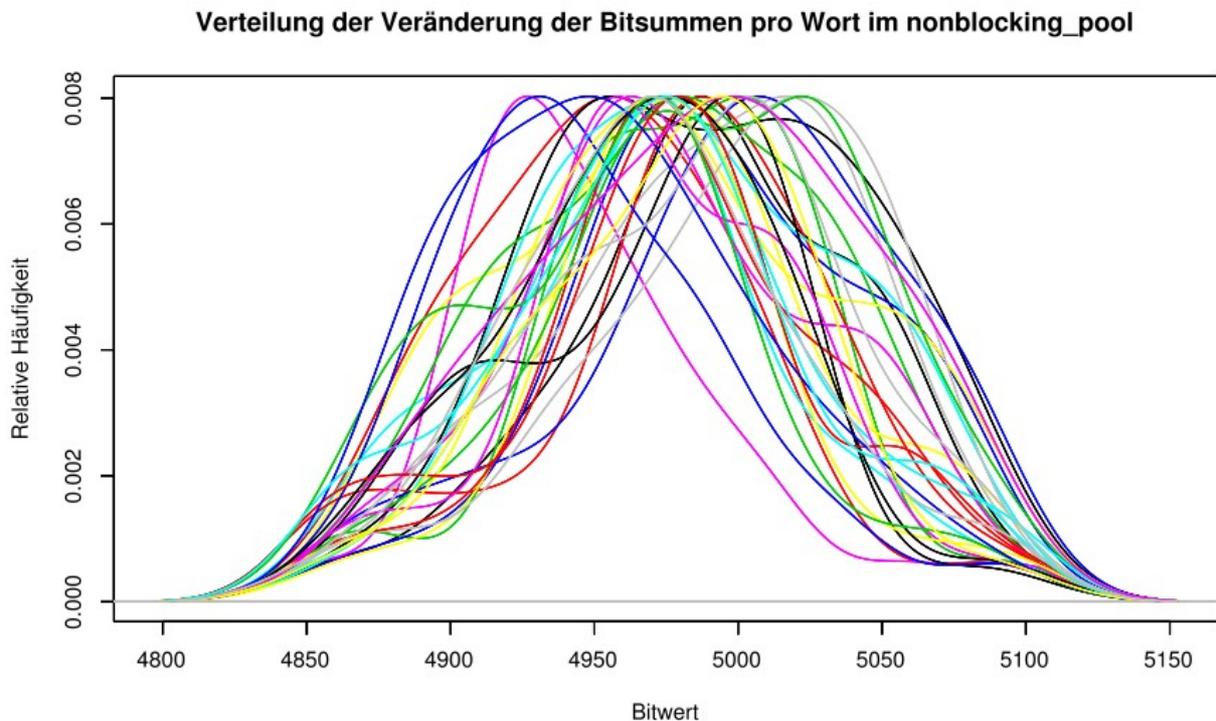
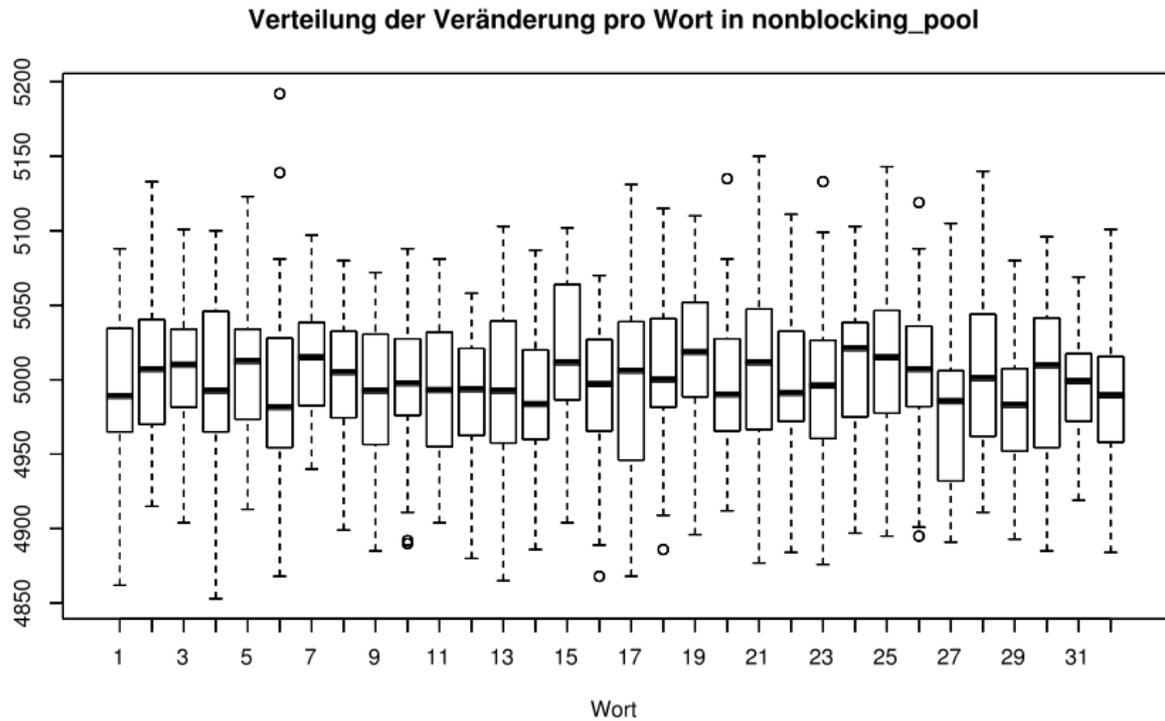
Verteilung der Veränderung in nonblocking_pool



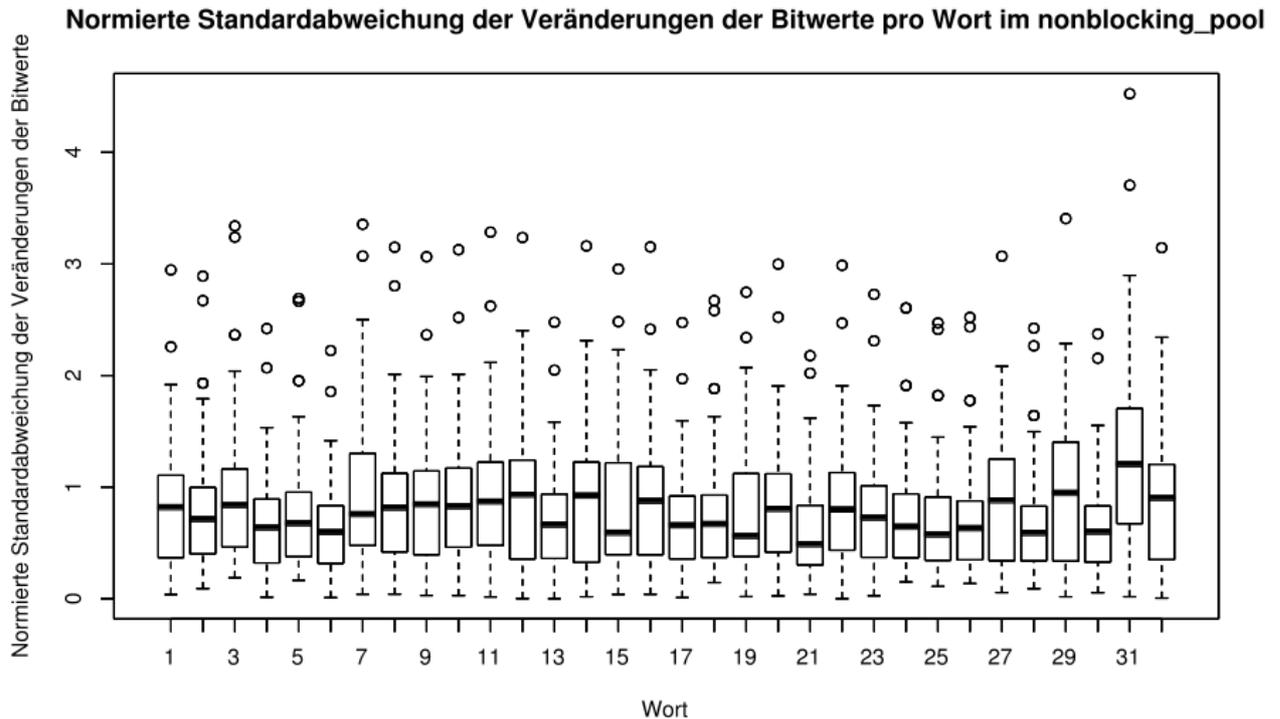
Verteilung der Veränderung in nonblocking_pool



Wie bereits mehrfach erklärt, ist im zweiten Analyseteil für jedes der 32 Wörter des nonblocking_pool eine eigene Messreihe mit der Anzahl an Änderungen der einzelnen Bitpositionen pro Wort zu betrachten.



Auch hier folgt abschließend ein Box-Whisker-Plot mit den normierten Standardabweichungen der Anzahl an Änderungen der Bitpositionen für jedes Wort.

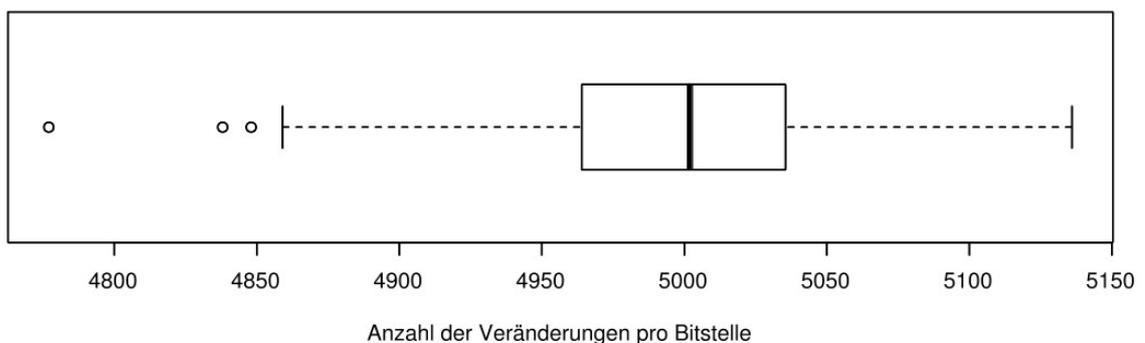


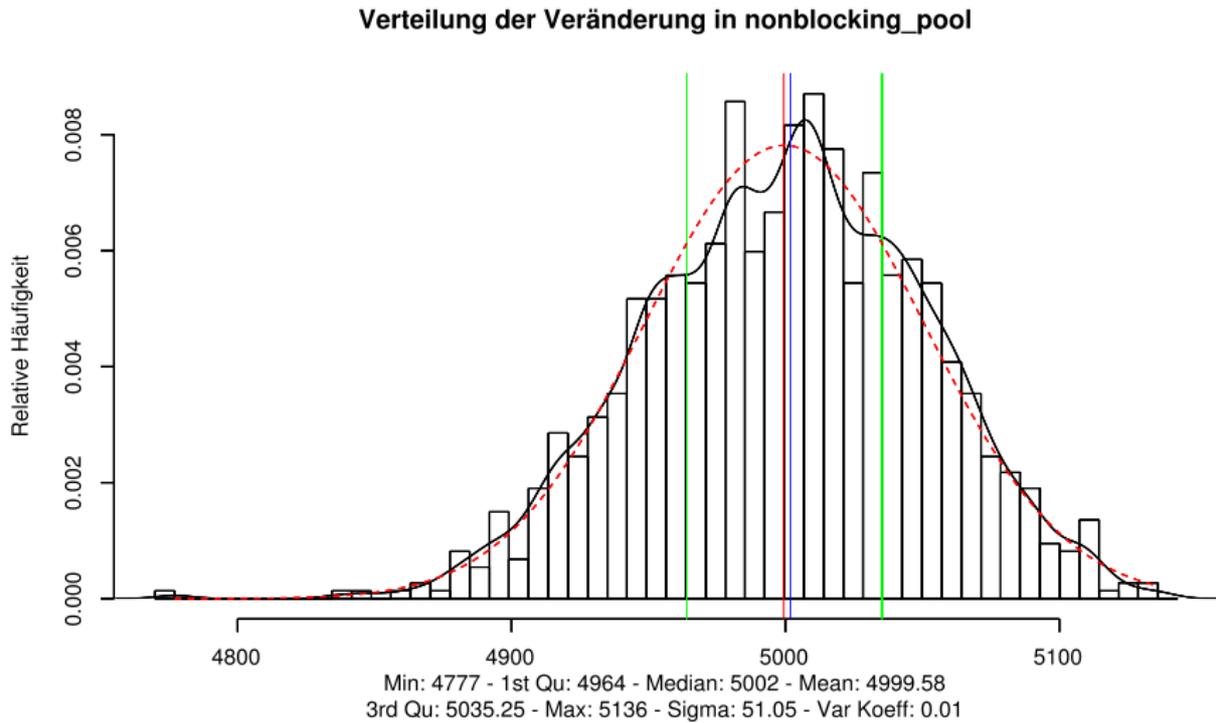
Die Verteilung der Änderungen einzelner Bitpositionen des Entropie-Pools nonblocking_pool zeigt eine deformierte Normalverteilung, die vermutlich aus der geringen Stichprobengröße resultiert. Mittelwert und Median entsprechen fast genau den Vorhersagen. Die wortweise Betrachtung zeigt, wie beim input_pool in Abschnitt 5.4.7.1 teils starke Abweichungen von einer Normalverteilung. Die in 5.4.7.1 zu findende Interpretation trifft auch hier zu. Hinweise, die gegen eine Änderungswahrscheinlichkeit für jede Bitposition von im Bereich von 50% sprechen, wurden nicht gefunden.

5.4.8.2 Graphische Veranschaulichung der Testresultate und Interpretation der Ergebnisse vor Hinzufügen des Seeds

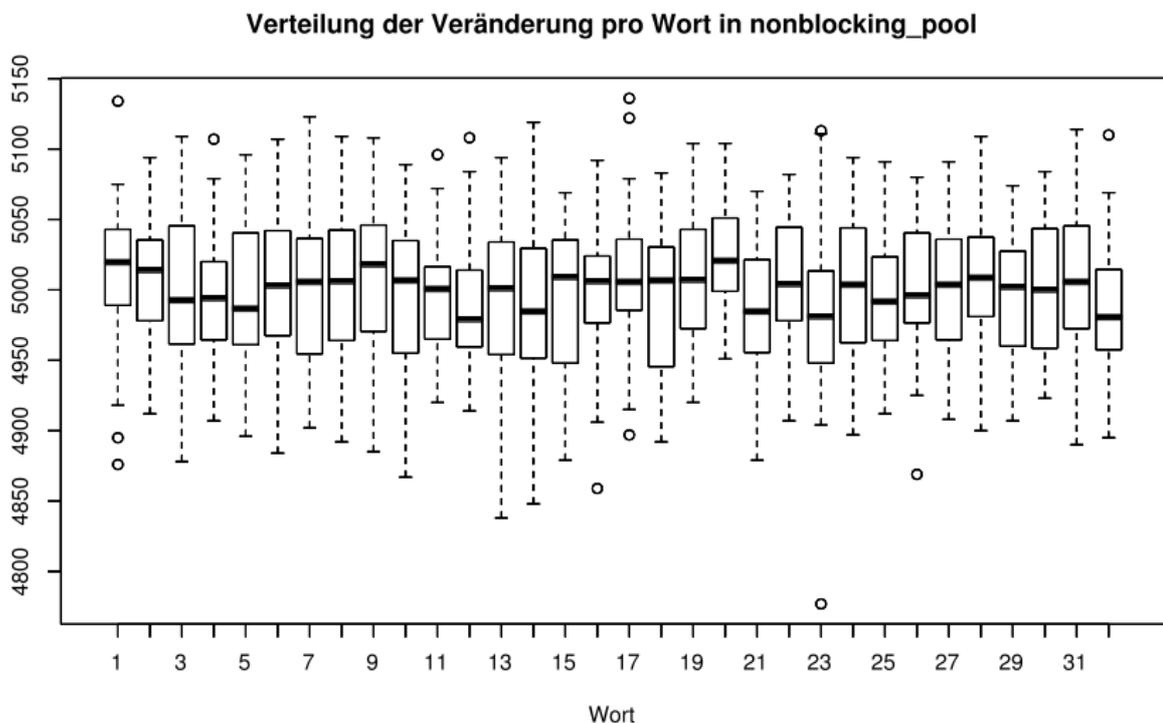
Der erste Teil der Analyse zeigt die Verteilung der Anzahl an Veränderungen für jede Bitposition innerhalb des gesamten Entropie-Pools.

Verteilung der Veränderung in nonblocking_pool

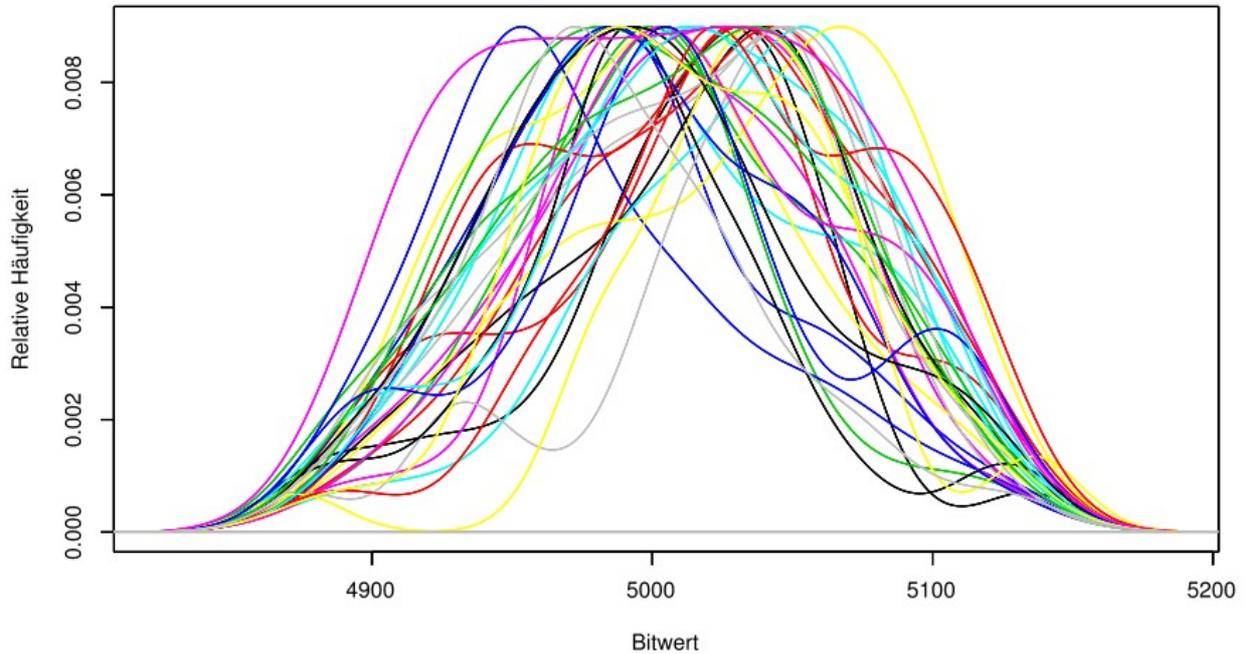




Der zweite Teil der Analyse diskutiert die Verteilungen der Anzahl an Änderungen einzelner Bitpositionen für jedes Wort.

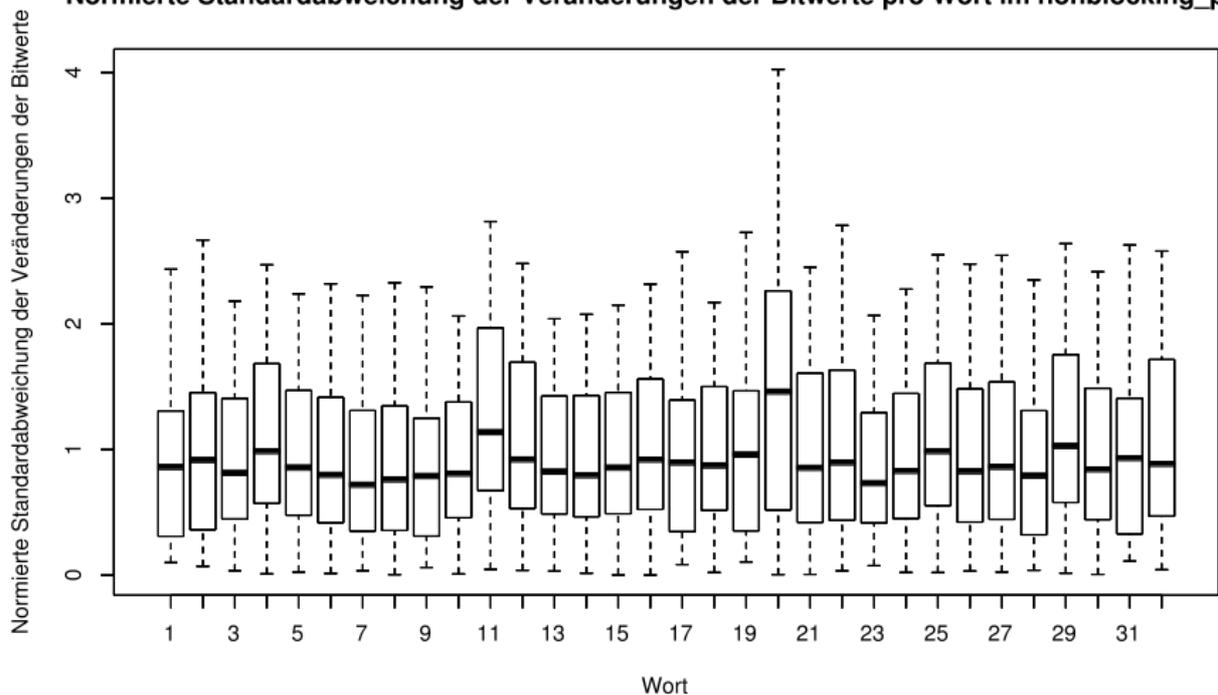


Verteilung der Veränderung der Bitsummen pro Wort im nonblocking_pool



Wie zuvor folgt nun noch ein Box-Whisker-Plot angezeigt, der die normierte Standardabweichung der Anzahl an Änderungen einzelner Bitpositionen für jedes Wort darstellt.

Normierte Standardabweichung der Veränderungen der Bitwerte pro Wort im nonblocking_pool



Die Verteilungen der Anzahl an Änderungen der einzelnen Bitpositionen des Entropie-Pools sind sehr ähnlich zu denen des vorherigen Abschnitts 5.4.8.1, wobei die Deformierung noch

etwas stärker ausgeprägt ist. Als Ursache kommen wieder die geringe Stichprobengröße, aber auch die relative Ähnlichkeit des Startvorgangs in Betracht.

Auch die wortweisen Ergebnisse sind denen aus Abschnitt 5.4.8.1 vergleichbar, wobei der Box-Whisker-Plot der normierten Standardabweichungen eine Vergrößerung der Standardabweichung auffällt. Auch hier wird vermutet, dass die relative Ähnlichkeit der Ereignisse während des Startvorgangs des User-Space bis zum Seeding zu diesem Resultat führt. Dies liefert wieder einen Hinweis für die Notwendigkeit des Seedings.

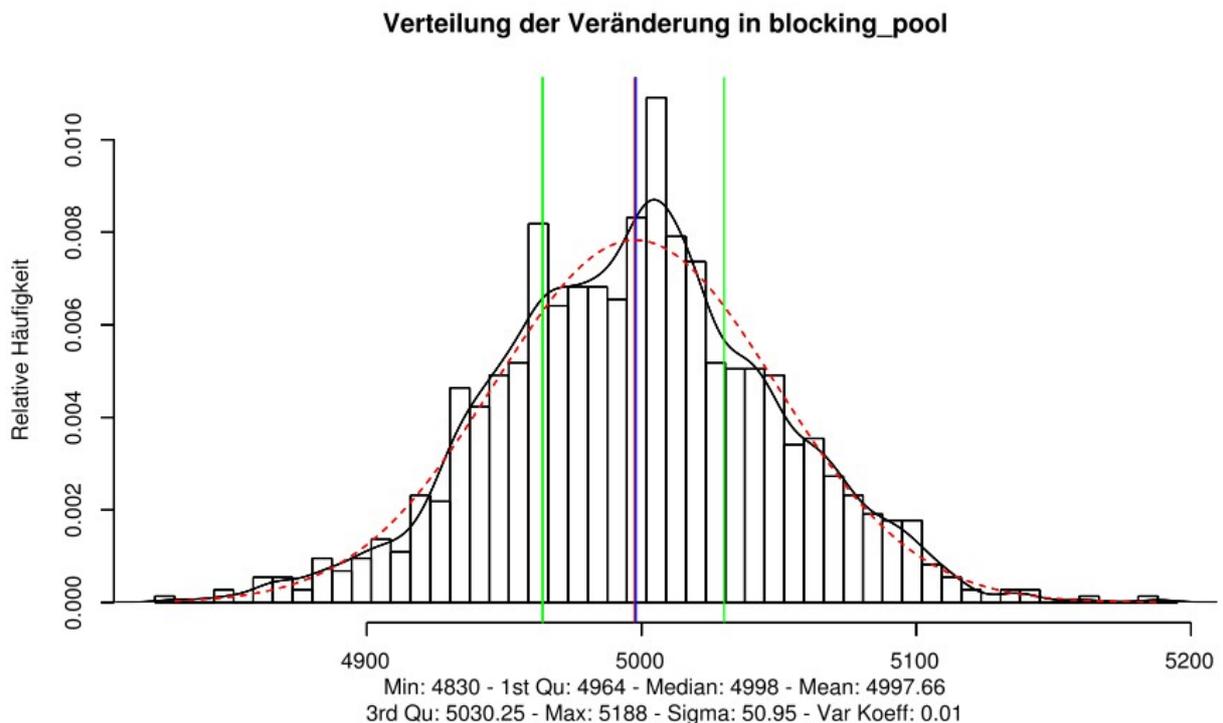
Somit scheinen die wortweisen Verteilungen der Änderungen pro Bitposition nicht immer einer Normalverteilung zu entsprechen. Die Abweichungen werden allerdings noch als unkritisch angesehen, zudem gleichen sich die Unterschiede bei Betrachtung aller Wörter aus.

Es wurden wieder keine Hinweise gegen eine Änderungswahrscheinlichkeit für jede Bitposition von im Bereich von 50% gefunden.

5.4.9 Test 3 - Veränderungen pro Bitposition im Entropie-Pool `blocking_pool` zum Systemstart

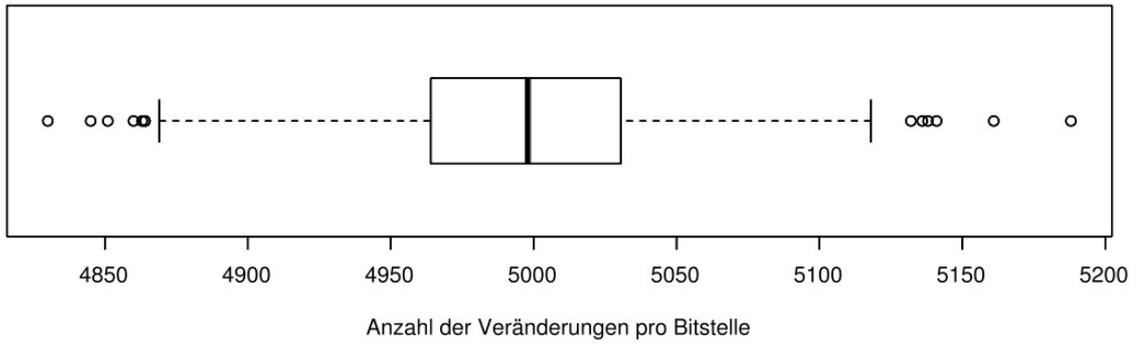
5.4.9.1 Graphische Veranschaulichung der Testresultate und Interpretation der Ergebnisse vor Start des User-Space

Der erste Teil der Analyse zeigt die Verteilung der Anzahl an Veränderungen für jede Bitposition innerhalb des gesamten Entropie-Pools.

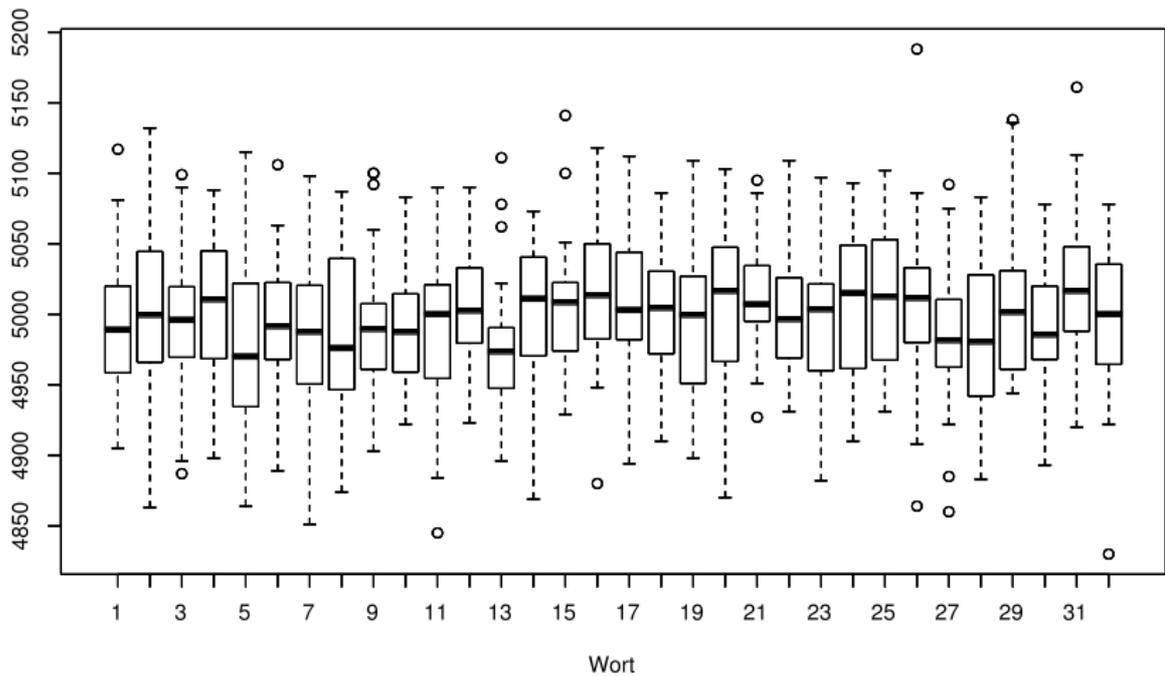


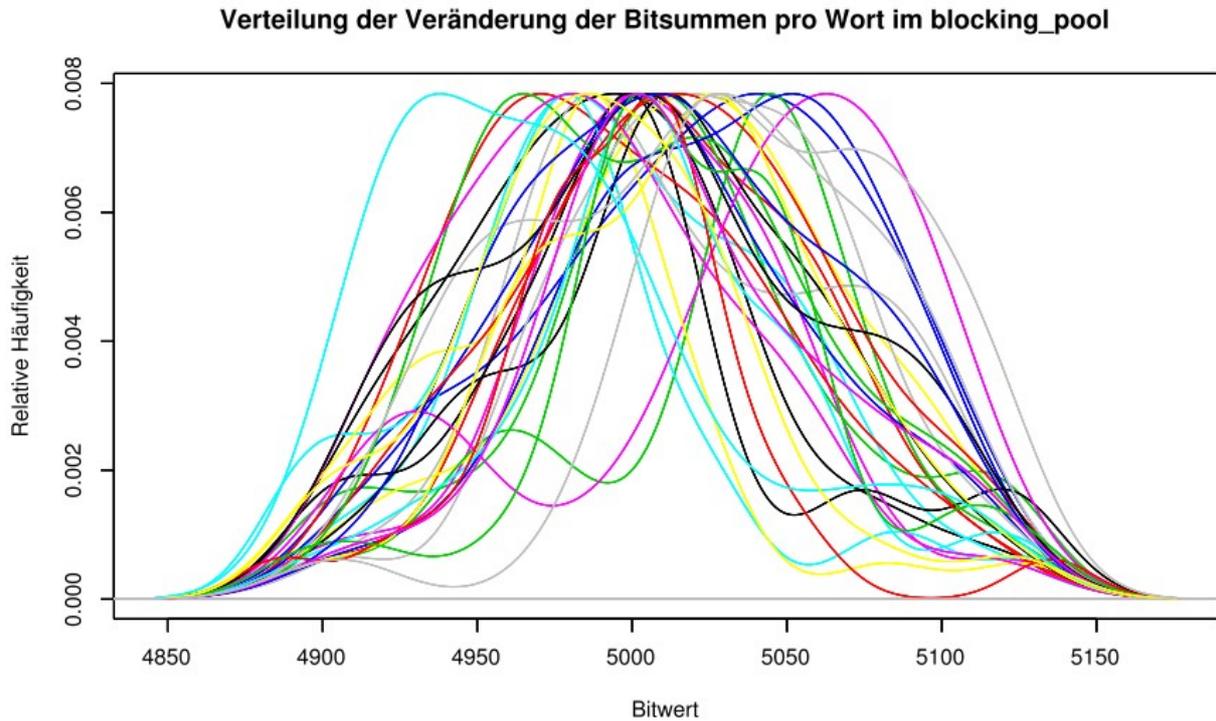
Wir erinnern hier nochmals daran, dass im zweiten Teil der Analyse aufgrund der Größe von `blocking_pool` 32 Verteilungen der Veränderungen einzelner Bitpositionen für jedes Wort betrachtet werden.

Verteilung der Veränderung in blocking_pool

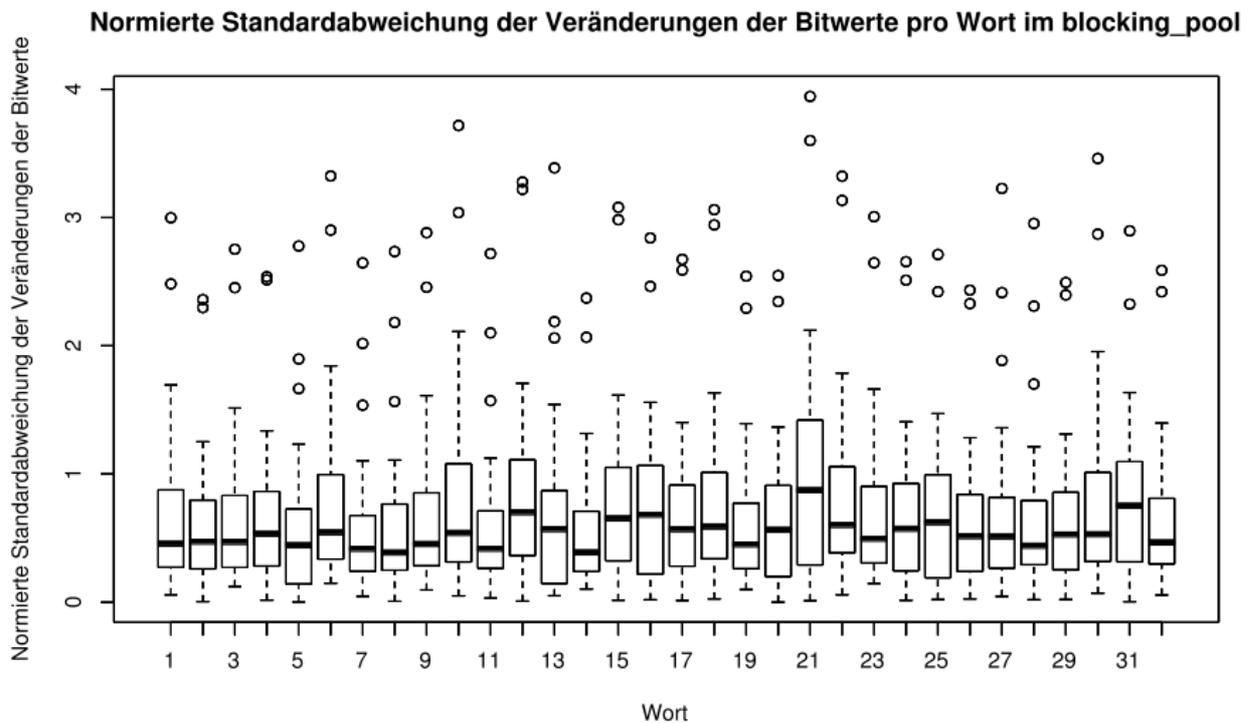


Verteilung der Veränderung pro Wort in blocking_pool





Ein weiterer Box-Whisker-Plot zeigt noch die normierten Standardabweichungen der Anzahl an Änderungen der Bitpositionen pro Wort.



Die Verteilung der Änderungen der einzelnen Bitpositionen des blocking_pool ist nahezu identisch mit den Beobachtungen in Abschnitt 5.4.8.1. Dementsprechend gilt die dort zu findende Interpretation.

5.4.9.2 Graphische Veranschaulichung der Testresultate und Interpretation der Ergebnisse vor Hinzufügen des Seeds

Wie in 5.2.8.2 erklärt, werden Zufallszahlen aus dem `blocking_pool` nur vom User-Space via `/dev/random` ausgelesen. Da der Startvorgang bis zum Verarbeiten des Seeds keine Anfragen an `/dev/random` durchführt, wurde der Pool keinen Änderungen unterworfen.

6 Testreihen II - Untersuchung der Entropie

In diesem Kapitel werden verschiedene Tests zur Untersuchung der Entropie durchgeführt. Im ersten Abschnitt wird die Entropie in `input_pool` betrachtet. Der zweite, deutlich größere Teil widmet sich der Entropie bzw. Entropieschätzung für die Hardware-Ereignisse.

6.1 Entropieschätzung für `input_pool`

Wie in Kapitel 2 beschrieben wird neue Entropie durch Hardware-Ereignisse in den 4096 Bit großen `input_pool` eingemischt. Es stellt sich die natürliche Frage, ob dieser Platz ausreicht und ob die darin gesammelte Entropie auch ausreichend ist. Die folgenden Tests gehen dieser Fragestellung nach.

6.1.1 Zeitverhalten von Trickle-Threshold

Der LRNG verwaltet eine globale Variable `trickle_thresh`, die das Verwerfen von Hardware-Ereignissen steuert. Entsprechend Abschnitt 2.5.3 führt ein Überschreiten dieser Schranke (die standardmäßig auf 3584 gesetzt ist) von der Entropieschätzung des `input_pools` dazu, dass nur noch jedes 4096-te Ereignis zu dem `input_pool` hinzugefügt wird.

In diesem Test wird für eine bestimmte Anzahl an Hardware-Ereignissen geprüft, ob diese dem `input_pool` hinzugefügt werden. Es wird also geprüft, ob die Schranke `trickle_thresh` noch nicht überschritten wurde.

6.1.1.1 Testdurchführung

Ausgehend von den Anfangsüberlegungen wurde folgender Test erstellt:

- Das SystemTap-Skript `trickle_threshold.stp`, in dem die Stichprobenanzahl mit der Variable `num_samplings` angepasst werden kann, erzeugt eine Tabelle mit der Verwendung der Hardware-Ereignisse.
- Das SystemTap-Skript wird mit dem Bash-Skript `gendata.sh` aufgerufen.
- Das Shell-Skript `update-graphs.sh` erstellt die folgenden Graphiken aus der erzeugten Datenreihe.

Als Stichprobengröße wurde $S=100.000$ gewählt, d.h. es wurden entsprechend Hardware-Ereignisse auf ihren Eingang in den `input_pool` geprüft.

6.1.1.2 Testresultate und Interpretation

Die folgende Tabelle fasst die Ergebnisse dieses Tests zusammen und ist dazu wie folgt aufgebaut:

- Die linke Spalte enthält die Hardware-Klassen.
- Die zweite Spalte enthält die Häufigkeit der Ereignisse (die Summe dieser Werte ist gleich der Stichprobenanzahl).
- Die dritte Spalte enthält die Anzahl der Ereignisse, welche zu dem `input_pool` hinzugefügt wurden.
- Die vierte Spalte gibt an, wie viele Ereignisse verworfen wurden und demzufolge nicht zu dem `input_pool` hinzugefügt wurden. Die verworfenen Ereignisse für HID und Disk sind aufgrund der Überschreitung der `trickle_thresh` Schranke verworfen wurden. Die verworfenen Ereignisse für IRQ sind aufgrund der Nutzung von `fast_pool` nicht in den `input_pool` eingemischt worden.

Event source	Events counted	Events used	Events discarded
HID	17932	11268	6657
IRQ	64464	62	64402
Disk	17605	14975	2630

Der Test zeigt, dass die Schranke `trickle_threshold` während des Testzeitraums hin und wieder erreicht wurde. Da im Testzeitraum kaum neue Programme gestartet wurden, ist dies konsistent mit den Resultaten des Tests in Abschnitt 6.1.2.

Da der `input_pool` somit im gesamten Testverlauf nicht annähernd die maximale Entropie beinhaltet, zeigt der Test, dass die Größe des `input_pools` für die gesammelte Entropie mindestens ausreichend ist.²³

6.1.2 Zeitlicher Entropieverlauf

Während des laufenden Betriebs werden immer wieder Hardware-Ereignisse aufgefangen und zur Aktualisierung des `input_pool` verwendet.

Die geschätzte Entropie über einen Zeitverlauf in einem ruhigen System kann mit einem einfachen Skript überwacht werden, da die Entropieschätzung des `input_pools` über folgende Datei auslesbar ist:

```
/proc/sys/kernel/random/entropy_avail.
```

Dieser Diese Datei wird in diesem Test periodisch ausgelesen.

Um das normale Verhalten des LRNG so gut wie möglich zu beobachten, sollte der Test so wenig Hardware-Ereignisse wie möglich selbst generieren, als auch ohne Starten von neuen Programmen auskommen. Das wird hier durch folgende Maßnahmen sichergestellt:

- Ein Perl-Skript nutzt nur Perl-interne Funktionen.
- Das Skript greift erst nach Abschluss des Tests auf die Festplatte zu und erzeugt somit keine Blockgerät-Ereignisse während des Tests.
- Das Skript benötigt keine Benutzerinteraktion und erzeugt deshalb keine HID-Ereignisse.

Demzufolge verändert das Skript die Entropie des LRNG nur durch folgende Operationen, die Blockgeräte-Ereignisse erzeugen:

- Laden der Skript-Datei von der Festplatte - dies erfolgt außerhalb des Testzeitraums.
- Laden von Perl von der Festplatte. (Das Skript ist eines der ersten Programme im System, damit liegt das Perl-Programm noch nicht im Blockgeräte-Zwischenspeicher).

Die Stichprobe basiert auf der Anzahl der Messungen der Entropieschätzung, wobei ein Abgreifen der Entropieschätzung nach einer bestimmten Zeit erfolgt. Es bestünde auch die Möglichkeit, die Messungen ausschließlich über einen gegebenen Zeitraum durchzuführen. Dieser Ansatz wurde aber verworfen, um eine bessere Vergleichbarkeit zu den vorangegangenen Tests zu ermöglichen.

6.1.2.1 Testdurchführung

Ausgehend von diesen Anfangsüberlegungen wurde folgender Test erstellt:

- Ein Perl-Skript `entropy_estimator_time.pl` wurde erstellt, in welchem die Stichprobenanzahl mit der Variable `num_samplings` angepasst werden kann. Dieses Skript erzeugt eine Liste mit den beobachteten Werten der Entropieschätzung.

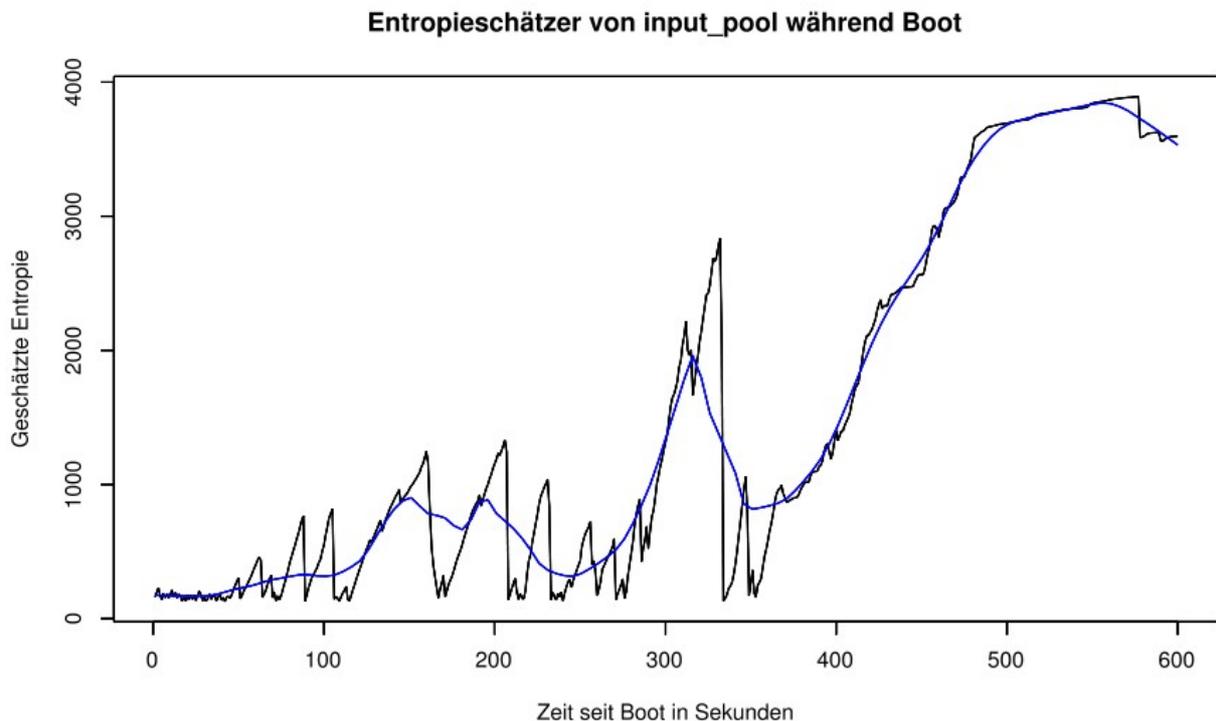
²³ Das bedeutet jedoch nicht, dass im System auch genügend Entropie vorhanden ist.

- Das Bash-Skript wird mit dem Bash-Skript `gendata.sh` initialisiert. Dieses Skript installiert `entropy_estimator_time.sh` nach `/sbin` und `entropy_estimator_time.conf` als `upstart-job`, damit der Test beim Starten durch `/sbin/init` ausgeführt wird.
- Das R-Project-Programm `entropy_estimator_time.r` erstellt die folgenden Graphiken aus der erzeugten Datenreihe.

Als Stichprobenanzahl wurde $S=600$ gewählt, als Zeit zwischen den Auslesevorgängen 1 Sekunde.

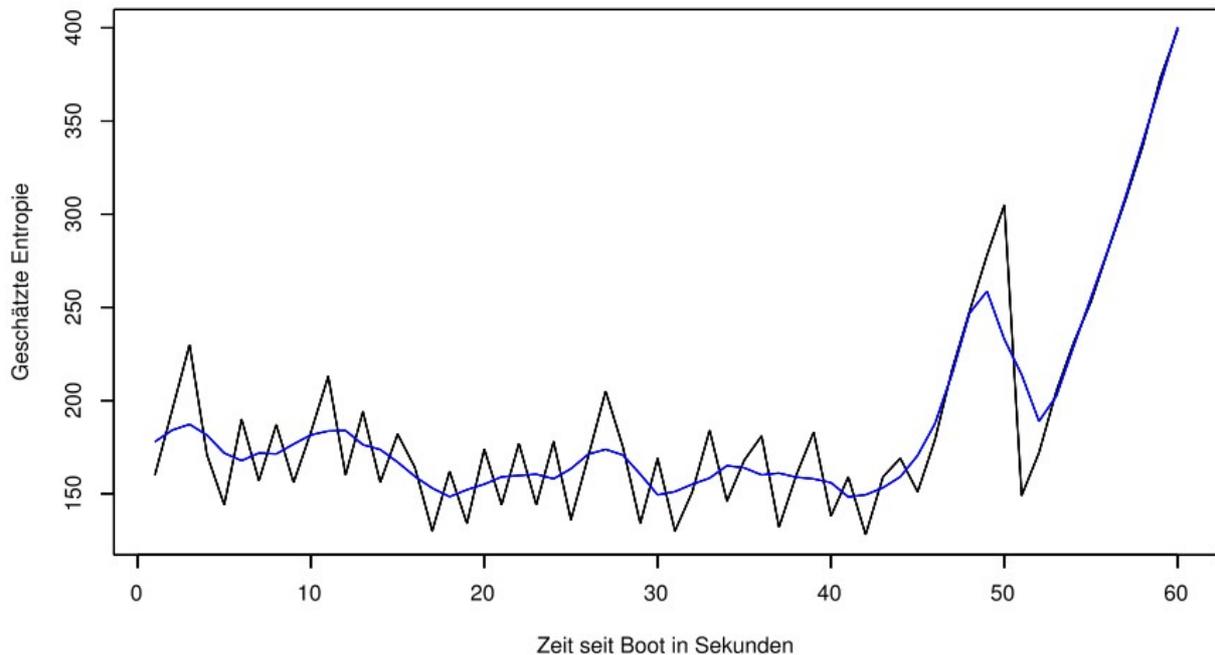
6.1.2.2 Testresultate und Interpretation

Das folgende Diagramm veranschaulicht den auf unserem Testsystem gemessenen Verlauf der Entropieschätzung. Dabei beschreibt die schwarze Linie im den Verlauf der einzelnen Messwerte, die blaue Linie eine Glättung derselben Werte.



Aufgrund der starken Schwankung der Entropieschätzung während der Initialisierung des User-Space, sind die ersten 60 Sekunden vergrößert in folgendem Diagramm dargestellt.

Entropieschätzer von input_pool während Boot - ersten 60 Sekunden



Die Testresultate sind unerwartet. Den Erwartungen entsprechen würde ein Kurvenverlauf, der

- während des Bootvorgangs von 40 Sekunden stetig ansteigt, da viel Entropie durch Festplattenzugriffe generiert wird. Gegebenenfalls wären ein oder zwei sehr kleine Einbrüche festzustellen, wenn der OpenSSH-Daemon startet, der Entropie für seinen DRNG bezieht;
- nach dem Beenden des Bootvorgangs und beim normalen Betrieb weiter ansteigt. Nach Erreichen von Trickle-Threshold würde sich die Zunahme der Entropie stark abschwächen. Je nach Nutzung wären wieder kleine Einbrüche festzustellen, z.B. wenn der Nutzer Schlüssel oder Seeds, etwa beim Starten des OpenSSH Clients, beziehen will.

Die aufgezeichnete Kurve entspricht diesen Erwartungen in keiner Weise - auf Anstiege folgen starke Einbrüche, die sogar eine weitere Extraktion von Entropie aus dem input_pool unmöglich machen.

Um eine Erklärung für das unerwartete Verhalten des Entropie-Verlaufs zu finden, sind folgende Beobachtungen hilfreich:

- Der erste extreme Einbruch des Entropie-Verlaufs findet um Sekunde 50 der Betrachtung statt. Zu diesem Zeitpunkt werden sehr viele Prozesse aufgrund des Startens des graphischen Desktops in der Testumgebung initialisiert.
- Die Ausgabe von folgendem Shell-Code

```
while [ 1 ]
do
    cat /proc/sys/kernel/random/entropy_avail
    sleep 1
done
```

Quellcode 34: Auslesen der Entropieschätzung des input_pool

zeigt, dass die Entropieschätzung des `input_pools` fast ständig auf ein Niveau abnimmt, das keine weitere Extraktion von Entropie aus dem `input_pool` erlaubt.

Beide Beobachtungen legen die Vermutung nahe, dass beim Starten von Prozessen Entropie verbraucht wird. Die Resultate dieser Analyse hatten zur Folge, dass die Autoren den Test, welcher im Abschnitt 7.1 beschrieben ist, programmierten und zusammen mit dem Quellcode 34 ausführten.

Das Resultat ist, dass bei jedem `execve`-Systemaufruf die Funktion `create_elf_tables` aufgerufen wird, welche 16 Bytes aus dem `nonblocking_pool` mittels der Funktion `get_random_bytes` extrahiert. Diese 16 Bytes werden in einen wohldefinierten Speicherbereich des Prozesses kopiert. Nach Rücksprache mit Kernentwicklern werden folgende Gründe für diese Entropienutzung angeführt, wobei sich die Kernentwickler nicht vollkommen sicher waren:

- Address-Space-Layout-Randomization basiert auf dem zufälligen Aufbau des Prozess-Speicherbereichs, d.h. die geladenen Bibliotheken werden an zufällige Speichersegmente geladen. Hierfür könnten diese 16 Bytes verwendet werden.
- Des Weiteren scheint eine API der C-Bibliothek zu definieren, dass ein Prozess Seed-Material für einen DRNG in einem bestimmten Speichersegment finden kann.

Da jedes Ausführen eines Prozesses 16 Bytes Entropie verbraucht, sind die Testresultate inklusive der oben dargestellten Kurve erklärbar. In der Kurve werden maximal ca. 1500 Bits Entropie erreicht, bevor ein Einbruch stattfindet. Ein Einbruch von 1500 unter die Schwelle, unter der aus dem `input_pool` keine Daten mehr extrahiert werden, ist bereits mit dem Ausführen von

$$\frac{1500 \text{ Bits}}{128 \frac{\text{Bits}}{\text{Prozess}}} \approx 11 \text{ Prozesse}$$

erreicht. Beim Starten der graphischen Oberfläche werden auf jeden Fall mehr Prozesse innerhalb sehr kurzer Zeit (wenige Sekunden) erzeugt.

Diese Testresultate wurden an die Kernentwickler weitergeleitet. Dies löste eine interessante Diskussion zwischen den Kernentwicklern und den C-Bibliotheksentwicklern aus. Ein Ergebnis ist derzeit noch nicht erkennbar.

6.2 Analyse des Entropiemaßes für Ereignisse

6.2.1 Entropieschätzung des LRNG pro Hardware-Ereignis

Vor den eigentlichen Tests zur Analyse des Entropiemaßes für Ereignisse führen wir noch eine grundlegende Untersuchung durch.

Für jedes Hardware-Ereignis, das durch die Entropiesammelfunktionen aufgefangen wurde, berechnet der LRNG eine Schätzung der Entropie. In Abschnitt 2.5.3 haben wir gesehen, dass diese Entropie auf den Zeitvarianzen basiert. **In diesem Test wird für jede relevante Hardware-Klasse eine bestimmte Anzahl an Hardware-Ereignissen samt zugehöriger Entropieschätzung gemessen.** Im Hinblick auf die Qualität des LRNG ist eine konservative Entropieschätzung wünschenswert. Dafür müssten die folgenden Aussagen zutreffen:

- Es ist zu erwarten, dass niedrige Entropiewerte in erheblich größerem Umfang berechnet werden als hohe Werte.
- Es gibt eine sinnvolle obere Schranke für die Entropie pro Ereignis. Dies erfüllt der LRNG nach Abschnitt 2.5.3, wobei die maximale Entropie pro Ereignis 11 Bit beträgt.

Die hier durchgeführte separate Betrachtung pro Hardware-Klasse erlaubt Vergleiche, ob die Heuristik der Entropieschätzung des LRNG eine Hardware-Klasse bevorzugt.

6.2.1.1 Testdurchführung

Ausgehend von den Anfangsüberlegungen wurde folgender Test erstellt:

- Das SystemTap-Skript `entropy_per_event.stp`, in dem die Stichprobenanzahl mit der Variable `num_samplings` angepasst werden kann, speichert die Entropie pro Ereignis und Klasse. Dabei wird sichergestellt, dass die Stichprobenzahl für jede Hardware-Klasse erreicht wird.
- Das SystemTap-Skript wird mit dem Bash-Skript `gendata.sh` aufgerufen.
- Ein R-Project Analyseprogramm `entropy_per_event.r` erstellt die nachfolgenden Graphiken aus der mittels dem SystemTap-Skript erzeugten Datenreihe.

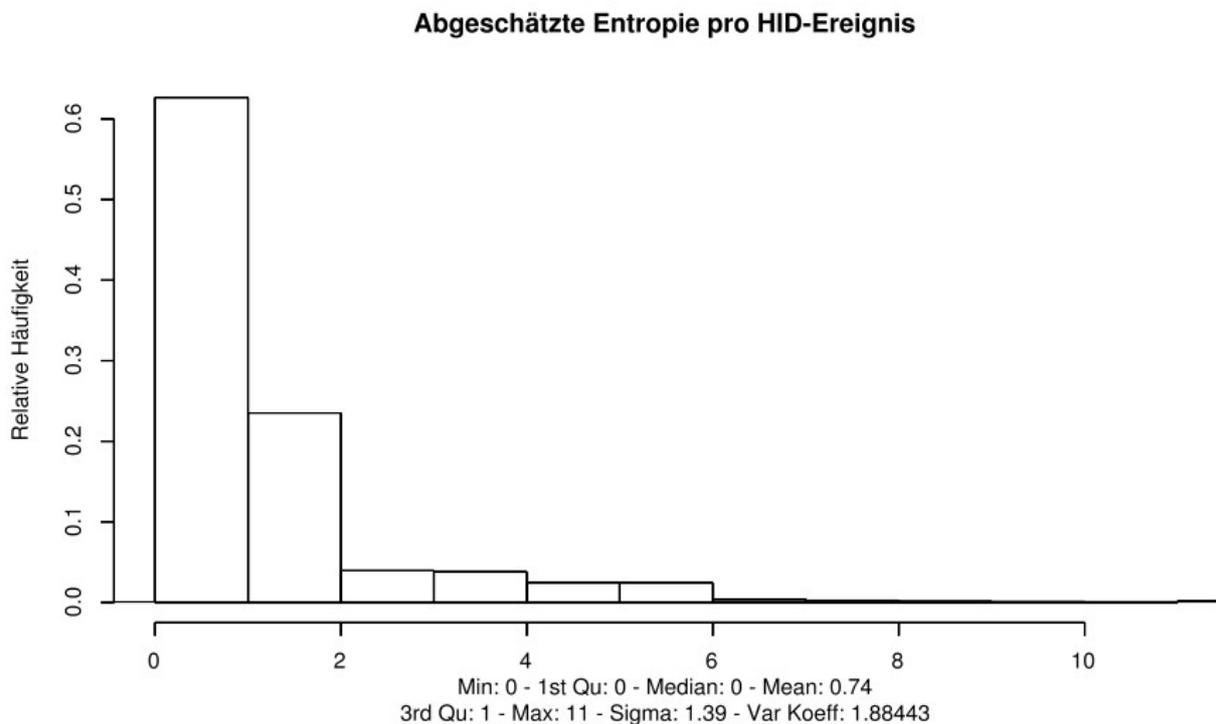
Der gewählte Stichprobenumfang beträgt für jede Hardware-Klasse $S=100.000$.

6.2.1.2 Graphische Veranschaulichung der Testresultate und Interpretation

Wir bemerken zuerst, dass aufgrund der Anbindung des LRNG an die Interrupts, welche das Sammeln von Ereignissen auf wenige Gerätetreiber reduziert, keine Interrupt-Ereignisse vom LRNG aufgezeichnet werden konnten.

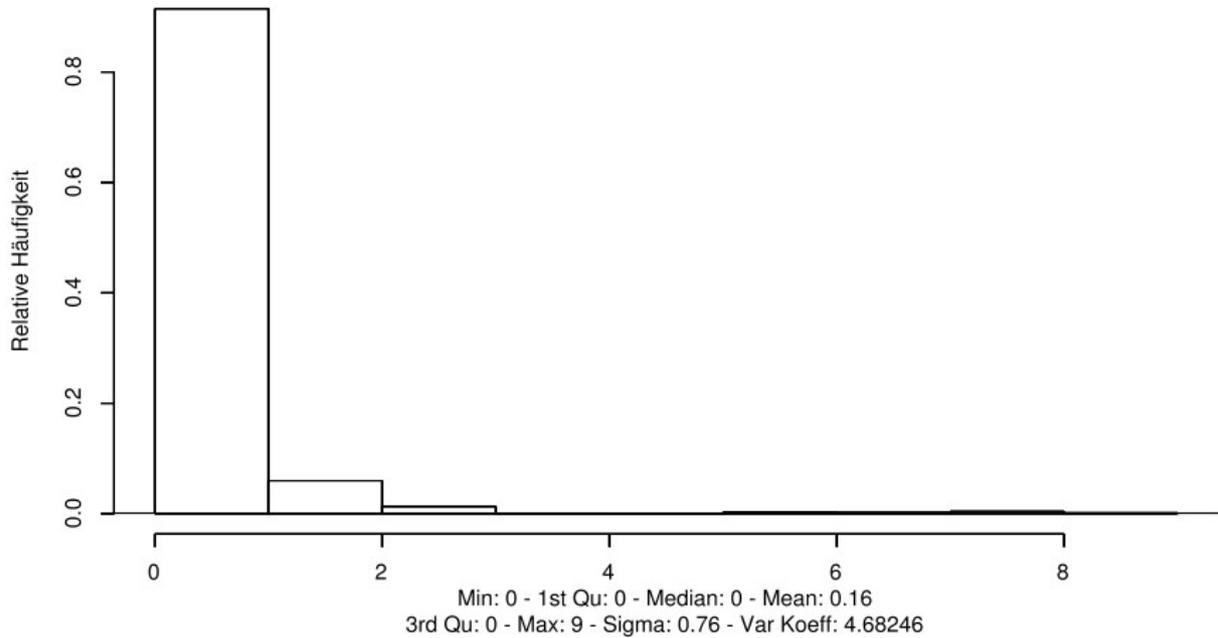
Für die Eingabegeräte (HID) erhält man folgendes Histogramm, das die Verteilung der geschätzten Entropie pro Ereignis aus der Klasse der Eingabegeräte, wobei

- die X-Achse die geschätzten Entropiewerte angibt
- und die Y-Achse die relative Häufigkeit des jeweiligen Entropiewerts zeigt.



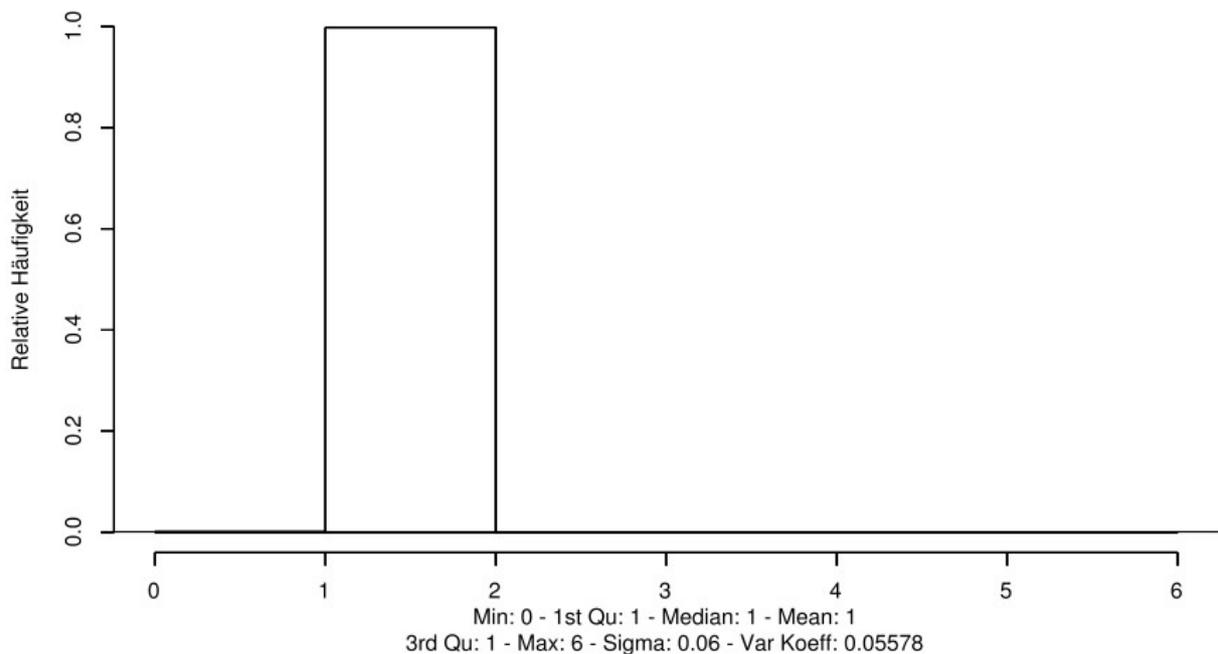
Für die Blockgeräte erhält man das folgende, analog aufgebaute Histogramm:

Abgeschätzte Entropie pro Blockgerät-Ereignis



Für die Interrupts erhält man das folgende, analog aufgebaute Histogramm:

Abgeschätzte Entropie pro IRQ-Ereignis



Basierend auf der Implementation der Verarbeitung der Interrupt-Ereignisse darf die abgeschätzte Entropie nur den Wert 1 haben. Der Graph hingegen zeigt auch andere Entropiewerte als 1. Dies lässt sich auf den in Anhang A.1.1 beschriebenen Messfehler zurückführen. Im Folgenden wird dieser Messfehler vernachlässigt.

Die Histogramme zeigen deutlich, dass für über 60% der HID-Ereignisse und über 80% der Blockgeräteeignisse die Entropieschätzung 0 Bit beträgt. Danach ist mit ca. 20% (HID) beziehungsweise ca. 10% (Blockgeräte) die Schätzung von 1 Bit am zweit-häufigsten. Somit scheint der LRNG die gewünschten Bedingungen zu erfüllen und dementsprechend konservativ bei der Abschätzung der Entropie pro Ereignis zu sein.

Eine genauere Betrachtung der Verarbeitung der Interrupts ist aufgrund folgender Eigenschaft wichtig: Die Ereignisse von den Eingabegeräten und Blockgeräten stammen alle von Operationen verschiedener Hardware-Geräte. Alle diese Hardware-Geräte erzeugen aber ein Interrupt für jede Operation. Der Interrupt signalisiert entweder, dass Daten vom Gerät abzuholen sind (falls eine Art „Lese-“Anfrage gestellt wurde) oder dass die Operation durchgeführt wurde (im Falle einer „Schreib-“Anfrage).

Demzufolge haben wir eine hohe Korrelation zwischen Interrupts und den Ereignissen bei Block- und Eingabegeräten.

Bei der Verarbeitung der Interrupts wird jeder Interrupt in den `fast_pool` der CPU eingemischt, die den Interrupt verarbeitet. Es wird **nicht** der `input_pool` verändert! Erst wenn in dem betrachteten `fast_pool` 1024 (oder ein vielfaches davon) Interrupts eingemischt wurden, werden die vier u32 Wörter des `fast_pool` in den `input_pool` eingemischt.

Desweiteren ist zu beachten, dass ein Interrupt mit Schieberegister-Verhalten in den `fast_pool` eingemischt wird – es werden keine bereits vorhandenen Daten überschrieben.

Mit diesen beiden Implementierungsdetails, wird damit nach Ansicht der Autoren die genannte Korrelation soweit gebrochen, dass die Zuweisung von 1 Bit Entropie pro Einmischen eines `fast_pools` in den `input_pool` als konservativ angesehen werden kann.

6.2.2 Übersicht: Analyse des Entropiemaßes

In den folgenden Abschnitten betrachten wir die folgenden zwei Fälle:

- Entropieerzeugung ausschließlich aufgrund von Aktivität der Blockgeräte.
- Entropieerzeugung ausschließlich aufgrund von Eingabegeräteaktivität.

Die Hardware-Klasse der Interrupts wird hier nicht betrachtet, da sie in allen anderen Messungen keine Werte geliefert hat.

Die Testdaten wurden erzeugt, indem im Quellcode ausschließlich die zu betrachtende Klasse von Ereignissen selektiert wurde. Die verwendeten SystemTap-Skripte zeichnen nur die Ereignisse der jeweiligen Klasse auf.

Für jede Hardware-Klasse werden die drei Komponenten des Ereignisses, die zusammen dem `input_pool` hinzugefügt werden, einzeln und in separaten Messungen aufgezeichnet. Diese drei Komponenten, die ausführlich in Abschnitt 2.5.1 beschrieben wurden, sind:

- Ereigniswert, der das Hardware-Ereignis definiert – Analyse siehe Abschnitt 6.2.3
- Prozessorzyklen – Analyse siehe Abschnitt 6.2.4
- Jiffies – Analyse siehe Abschnitt 6.2.5

Dieser Abschnitt beschäftigt sich mit der Entropiebetrachtung für die Ereignistypen, wobei die empirische Häufigkeitsverteilung für die Ereignistypen analysiert wird.

Die Wahrscheinlichkeit des Auftretens eines Ereignistyps ist gegeben durch

$$p = \frac{E}{N} ,$$

wobei E die Anzahl der Ausprägungen eines Ereigniswerts ist und N die Anzahl aller aufgetretenen Ereignisse bezeichnet.

Aus der empirischen Häufigkeitsverteilung wird durch

$$H_{min} = -\lg(p_{max})$$

die minimale Entropie (oder untere Entropieschranke) berechnet, wobei p_{max} die maximal aufgetretene Wahrscheinlichkeit für einen Ereigniswert ist und \lg der Logarithmus zur Basis 2.

Ferner geben wir einen Wert für die Shannon-Entropie auf Basis dieser empirischen Daten an, der sich berechnet als

$$H = -\sum_{i=1}^N p_i * \lg(p_i) ,$$

wobei p_i die Wahrscheinlichkeit für das Auftreten eines Einzelereigniswerts ist und N die Anzahl aller empirisch aufgetretenen Ereignisse bezeichnet.

6.2.3 Häufigkeitsverteilung und Entropie der Ereigniswerte

Die Häufigkeitsverteilung und die Entropie der Ereigniswerte pro Hardware-Klasse wird mittels diesem Test analysiert.

6.2.3.1 Testdurchführung

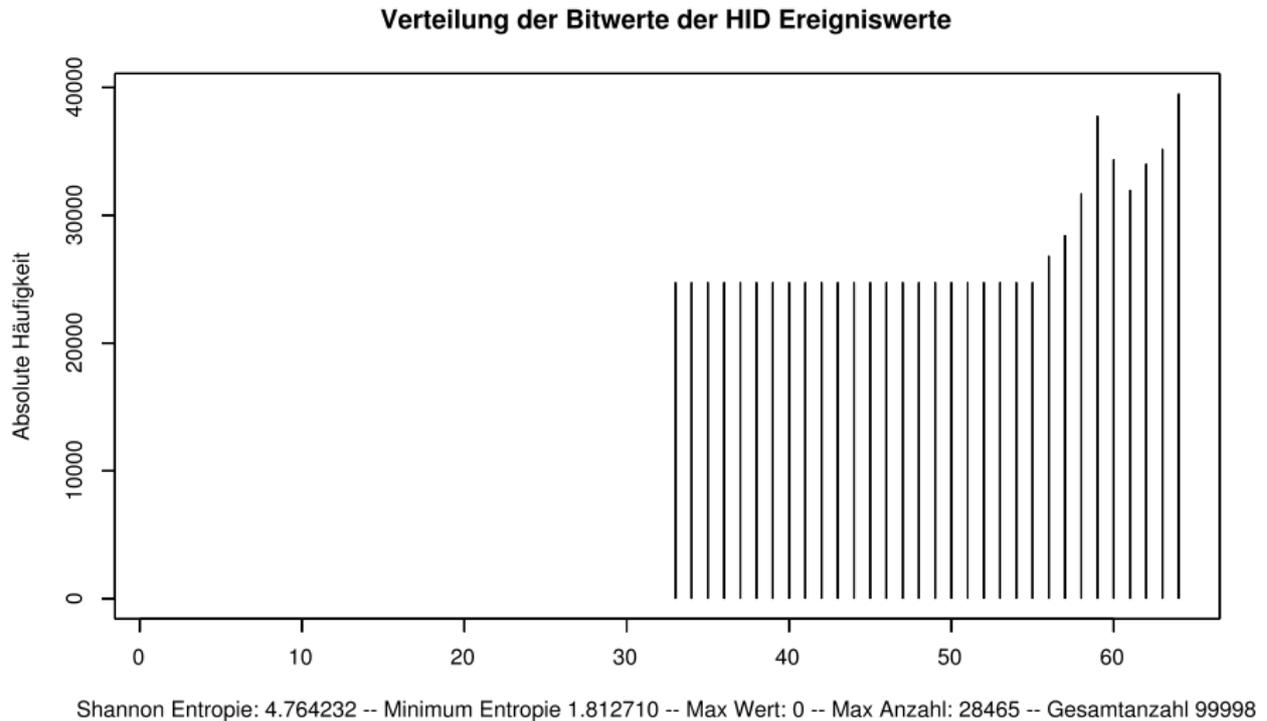
Ausgehend von diesen Anfangsüberlegungen wurden folgende Tests für die Hardware-Klassen für Eingabegeräte und Blockgeräteaktivität definiert:

- Die SystemTap-Skripte `raw_entropy_hid.stp`, `raw_entropy_disk.stp` und `raw_entropy_irq.stp` zeichnen die Ereigniswerte für die jeweilige Klasse auf. Dabei werden sowohl die Ereigniswerte aufgezeichnet. Aus diesen Daten wird mittels dem R-Project Programm die Anzahl der gesetzten Bits für jeden Ereigniswert bestimmt. Diese Ergebnisse erlauben die graphische Darstellung der Häufigkeitsverteilung der auftretenden Bits und eine Entropieabschätzung über die gewonnen Ereigniswerte durchzuführen. Die Stichprobengröße wird mit der Variable `num_samplings` in den SystemTap-Skripten festgelegt.
- Die SystemTap-Skripte werden mit dem Bash-Skript `gendata.sh` initialisiert.
- Die R-Project-Programme `raw_entropy_hid.r`, `raw_entropy_disk.r` und `raw_entropy_irq.r` erstellen die folgenden Graphiken aus den erzeugten Datenreihen.

6.2.3.2 Testresultate für Eingabegeräte

Die gesetzte Stichprobenanzahl ist: $S = 100.000$

Für die Entropieerzeugung ausschließlich aufgrund von Eingabegeräteaktivität erhalten wir die verschiedenen Ereignisse, die in der Datei `raw_entropy_hid-event-histtable.txt` zu finden sind. Dabei zeigt die Tabelle den Ereigniswert und die Wahrscheinlichkeit des Eintretens auf. Die Häufigkeitsverteilung der Summen pro Bitpositionen der Ereigniswerte der Eingabegeräte lässt sich mittels folgender Graphik darstellen.



Des Weiteren sind in dem Histogramm folgende Werte bezogen auf die Ereigniswerte dargestellt:

- Shannon-Entropie der Ereigniswerte
- Untere Entropieschranke
- Ereigniswert, welcher am häufigsten auftritt und damit die untere Entropieschranke bestimmt
- Anzahl des am häufigsten auftretenden Ereigniswerts
- Gesamtzahl der Ereigniswerte

6.2.3.3 Testresultate für Blockgeräte

Die gesetzte Stichprobenanzahl ist: $S = 100.000$

Für die Entropieerzeugung ausschließlich aufgrund von Blockgeräteaktivität erhalten wir auf unserer Testmaschine nur einen einzigen Ereigniswert, nämlich 0x00800100 (8388864). Die Berechnung einer Verteilung erübrigt sich damit.

6.2.3.4 Testresultate für Interrupts

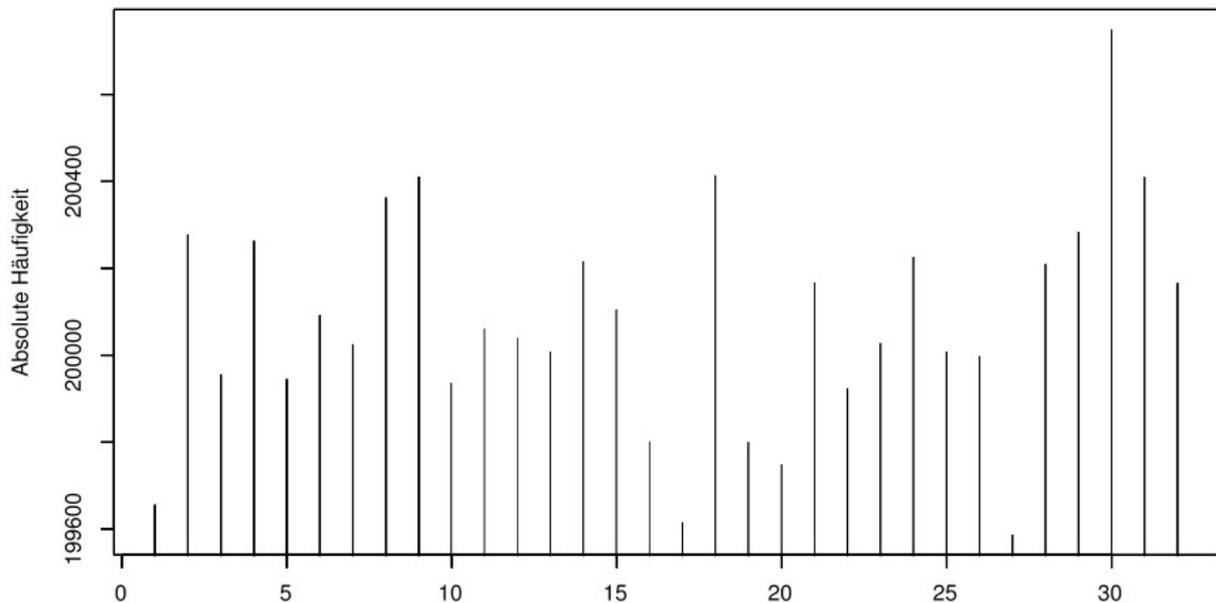
Die gesetzte Stichprobenanzahl ist: $S = 100.000$

Die Tests analysieren den Einfluss von Interrupts auf den `input_pool`. Entsprechend dem Design ist klar, dass für jedes Einmischen von Interrupt-Daten in den `input_pool` die vier u32 Wörter des `fast_pools` verwendet werden. Da die vier Wörter des `fast_pools` mittels dem Schieberegister-Verhalten und XOR verändert werden, ist zu erwarten, dass kein beobachteter Wert eines Wortes des `fast_pools` zweimal auftritt. Die Datei `raw_entropy_irq-event-histtable.txt` zeigt dies eindrucksvoll²⁴.

Die Häufigkeitsverteilung der Summen pro Bitpositionen der einzelnen Wörter des `fast_pools` lässt sich mittels folgender Graphik darstellen.

²⁴ Der erste Wert in `raw_entropy_irq-event-histtable.txt` enthält die 43 Messfehler (0.0001075 / 0.0000025), die auch schon in dem Histogramm in Abschnitt 6.2.1.2 angesprochen wurden. Diese Fehler werden im Folgenden vernachlässigt.

Verteilung der Bitwerte der IRQ Ereigniswerte



Shannon Entropie: 18.608379 -- Minimum Entropie 13.183376 -- Max Wert: 1 -- Max Anzahl: 43 -- Gesamtanzahl 400000

Zu beachten ist, dass der Wert „Gesamtzahl“ in der obigen Graphik sich zusammensetzt aus der Stichprobengröße multipliziert mit vier für die vier Wörter des `fast_pools`.

6.2.3.5 Interpretation der Ergebnisse

Die Interpretation der Ergebnisse muss separat für die Hardware-Quellen erfolgen:

- Eingabegeräte: Die Anzahl der aufgezeichneten Ereigniswerte ist relativ gering. Demzufolge ist die theoretische Shannon-Entropie und die untere Schranke der Entropie relativ gering.

Die Ereigniswerte der Eingabegeräte basieren auf folgenden Konzepten:

- Bei Tastaturen hat sowohl das Drücken, also auch das Loslassen jeder Taste²⁵ einen eigenen Ereigniswert. Dieser Wert ändert sich nicht, da er fest codiert ist. Demzufolge gibt es nur einige hundert verschiedene Ereigniswerte für Tastaturen.
- Bei Mäusen und ähnlichen Zeigegeräten werden die verschiedenen Bewegungsrichtungen des Gerätes als Ereigniswerte verarbeitet. Abhängig vom Zeigegerät kann es verschiedene Bewegungsrichtungen geben – d.h. es werden häufig beliebige Richtungsvektoren der zweidimensionalen Ebene aufgezeichnet. Dennoch sind auch hier wie bei Tastaturen nur eine geringe Zahl von unterschiedlichen Ereigniswerten vorhanden, welche wohl-bekannt sind.

Demzufolge kann für die praktische, qualitative Betrachtung der Ereigniswerte der Eingabegeräte geschlossen werden, dass wenig Entropie in den Ereigniswerten enthalten ist.

Zu beachten ist auch folgende, gegebenenfalls irreführende Information in dem oben stehenden Graphen: Die Bits 32 bis 55 scheinen gleichverteilt zu sein. Dies liegt aber ausschließlich daran, dass die Ereigniswerte für Mäuse sehr knapp unterhalb von 2^{32} liegen. Die anderen Ereigniswerte sind hingegen unterhalb von 1000. Demzufolge sind die Bits 32 bis 55 immer für jedes Ereignis einer Maus gesetzt. Damit enthält die gezeigte Gleichverteilung aber kaum reale Entropie.

25 Das Drücken einer Taste mit einer der verschiedenen Umschalttasten (Shift, Strg, Alt, AltGr, ...) wird für den Ereigniswert als eine separate Taste interpretiert.

- Blockgeräte: Es ist nur ein Ereigniswert für alle beobachteten Festplatten-Ereignisse aufgezeichnet worden. Dies ist darauf zurückzuführen, dass das Testsystem nur eine Festplatte enthält. Demzufolge zeigt die Verteilung der Bitwerte der Ereigniswerte also nur einen statischen Wert an. Entsprechend sind die theoretisch vorhandene Shannon-Entropie und die untere Schranke der Entropie fast Null.

Der Ereigniswert besteht aus der kerninternen Gerätenummer, die um einem konstanten Wert erhöht wird. Diese Gerätenummer ist fest in den Kern codiert, da aus dieser Nummer die Major- und Minor-Gerätenummern erzeugt werden²⁶. Wenn man bedenkt, dass der Ereigniswert der Blockgeräte relativ einfach vorhersagbar ist, ist die tatsächliche Entropie ebenfalls als Null bei einer Festplatte und bei fast Null bei mehreren Festplatten anzusehen.

- Interrupts: Die Wörter der `fast_pools` hingegen enthalten für sich betrachtet sehr viel Entropie. Alleine die berechnete Shannon-Entropie ist im Vergleich zu den anderen Ereigniswerten sehr hoch. Wenn man nun diese theoretische Entropie mit der in Abschnitt 6.2.1.2 diskutierte Korrelationsproblematik kombiniert, kann man folgendes feststellen: In Abschnitt 6.2.1.2 wurde beschrieben, dass die Korrelation bereits durch die Implementierung stark reduziert wurde. Die heuristische Entropie pro Einmischen der Interrupt-Daten in den `input_pool` ist genau 1 Bit – weniger als ein 13tel der theoretischen Shannon-Entropie. Demzufolge ist die heuristische Entropie als sehr konservativ anzusehen.

Schlussendlich kann für die Ereigniswerte folgende Aussage getroffen werden:

Die Ereigniswerte der Block- und Eingabegeräte liefern so gut wie keine theoretische Entropie für den LRNG. Die Ereigniswerte aus den `fast_pools` für die Interrupts hingegen liefern viel theoretische Entropie, welche aber unter dem Hintergrund der Korrelationsproblematik zu interpretieren ist.

Diese Aussage gilt sowohl bei der Betrachtung der theoretischen Entropie, als auch bei der praktischen Vorhersagbarkeit der Ereigniswerte.

6.2.4 Häufigkeitsverteilung und Entropie der Prozessorzyklen

Dieser Abschnitt betrachtet ausschließlich die Verteilung der Prozessorzyklen, wobei jede Hardware-Klasse in einem eigenen Unterabschnitte behandelt wird. Die angewandte Methodik ist dabei in allen Unterabschnitten identisch und wird hier vorgestellt.

Der erste Test ist eine empirische Analyse der Verteilung der einzelnen Bits in den Prozessorzyklen. Die Ergebnisse werden in Form einer Tabelle dargestellt.

Die nachfolgende Analyse beschäftigt sich ausschließlich mit der Differenz von direkt aufeinander folgenden Prozessorzyklen. Aufgrund der Auflösung des Zeitgebers (die Jiffies haben eine Auflösung von 4 ms), ist es teilweise schwierig, gute Aussagen über die Verteilung zu tätigen, wenn man die Differenzen (=Deltas) als 64 Bit-Werte betrachtet. Aus diesem Grund wurden zusätzliche Graphen erstellt, die ausschließlich die niederwertigen Bits betrachten.

Im Anschluss daran findet die Entropieabschätzung bezogen auf die betrachteten niederwertigen Bits statt. Hierzu erfolgt eine graphische Analyse der aufgetretenen Differenzen von Prozessorzyklen, aus der die zeitliche Verteilung der Differenzen von Prozessorzyklen abzulesen ist und ob in der zeitlichen Verteilung Änderungen in der Verteilung zu vermuten sind.

Eine weitere Grafik zeigt die absteigend nach Größe sortierten Differenzen von direkt aufeinander folgenden Prozessorzyklen. Aus dieser Grafik sind Wertebereiche von häufig aufgetretenen Differenzen von direkt aufeinander folgenden Prozessorzyklen abzulesen.

Danach folgt noch ein Histogramm der Differenzen von direkt aufeinander folgenden Prozessorzyklen.

²⁶ Der User-Space – vor allem der UDEV-Prozess – muss diese Major- und Minor-Nummern genau kennen, um die Gerätedateien zu erzeugen. Damit folgt die Erstellung dieser Nummern fest vorgegebenen und wohlbekanntes Schemas. Demzufolge folgt auch die Erstellung der kerninternen Gerätenummer diesem Schema.

Neben der in Abschnitt 6.2 vorgestellten Formel wird unter der Annahme, dass die einzelnen Bits stochastisch unabhängig sind, durch

$$H_{max} = - \sum_{i=1}^{32} (p_{0i} * \lg(p_{0i}) + p_{1i} * \lg(p_{1i}))$$

eine obere Schranke für die Shannon-Entropie berechnet, wobei p_{0i} die Wahrscheinlichkeit dafür ist, dass das i-te Bit nicht gesetzt ist und p_{1i} die zugehörige Gegenwahrscheinlichkeit.

6.2.4.1 Testdurchführung

Ausgehend von diesen Anfangsüberlegungen wurden folgende Tests für die Hardware-Klassen für Eingabegeräte und Blockgeräteaktivität definiert:

- Die SystemTap-Skripte `raw_entropy_hid.stp` und `raw_entropy_disk.stp` zeichnen die Ereigniswerte für die jeweilige Klasse auf. Dabei werden die 64 Bit-Werte der Prozessorzyklen aufgezeichnet. Diese Ergebnisse erlauben die graphische Darstellung einer Häufigkeitsverteilung der auftretenden Prozessorzyklen und deren Varianzen und eine Entropieabschätzung über die gewonnenen Ereigniswerte durchzuführen. Die Stichprobengröße wird mit der Variable `num_samplings` in den SystemTap-Skripten festgelegt.
- Die SystemTap-Skripte werden mit dem Bash-Skript `gendata.sh` initialisiert.
- Die R-Project-Programme `raw_entropy_hid.r` und `raw_entropy_disk.r` erstellen die nachfolgenden Graphiken aus den erzeugten Datenreihen. Dabei kann die Analyse auf die niederwertigen Bits mittels Setzen der Variable `significantbits` in den R-Project-Programmen auf einen beliebigen Wert zwischen 1 und 63 erfolgen.

Die Interrupts wurden nicht getestet, da die Prozessorzyklen bereits in den Wörtern des `fast_pools` berücksichtigt sind. Beim Einmischen der Zustände der `fast_pools` in den `input_pool` werden keine Prozessorzyklen mehr berücksichtigt. Demzufolge umfasst der Test in Abschnitt 6.2.3.4 bereits die Prozessorzyklen.

6.2.4.2 Testresultate für Eingabegeräte

Die gesetzte Stichprobenanzahl ist:

$$S = 100.000$$

6.2.4.2.1 Verteilung der Bits der Prozessorzyklen

Die empirische Analyse der Verteilung der einzelnen Bits in den Prozessorzyklen ist in folgender Tabelle dargestellt. Dabei können folgende Informationen aus der Tabelle gelesen werden:

- Die linke Spalte spezifiziert das diskutierte Bit beginnend mit dem signifikantesten (d.h. dem linken Bit) in absteigender Reihenfolge.
- Die rechte Spalte listet die Wahrscheinlichkeit p_{1i} eines gesetzten Bitwertes auf.
- Auf Basis der rechten Spalte, kann man mittels $1 - p_{1i} = p_{0i}$ die Gegenwahrscheinlichkeit berechnen.

Diese Tabelle basiert auf den Daten in der Datei `raw_entropy_hid_cycles-histtable-64.txt`.

Bitposition	Wahrscheinlichkeit für gesetztes Bit	Bitposition	Wahrscheinlichkeit für gesetztes Bit
1	0	33	0.492769855397108
2	0	34	0.50995019900398
3	0	35	0.503090061801236

Bitposition	Wahrscheinlichkeit für gesetztes Bit	Bitposition	Wahrscheinlichkeit für gesetztes Bit
4	0	36	0.49919998399968
5	0	37	0.500280005600112
6	0	38	0.502370047400948
7	0	39	0.502920058401168
8	0	40	0.503370067401348
9	0	41	0.498979979599592
10	0	42	0.500660013200264
11	0	43	0.50060001200024
12	0	44	0.500520010400208
13	0	45	0.506670133402668
14	0	46	0.497469949398988
15	0	47	0.5035000700014
16	0	48	0.500260005200104
17	0	49	0.50190003800076
18	0	50	0.500090001800036
19	0	51	0.499529990599812
20	0	52	0.501370027400548
21	0	53	0.499359987199744
22	0	54	0.501630032600652
23	0	55	0.496979939598792
24	0	56	0.501530030600612
25	0	57	0.49859997199944
26	0	58	0.499089981799636
27	0	59	0.49784995699914
28	0	60	0.498819976399528
29	0	61	0.497239944798896
30	0	62	0.500360007200144
31	0	63	0.518020360407208
32	0	64	0.457269145382908

Wie in der Tabelle zu sehen ist, werden nur die unteren 32 Bits verändert. Die Erklärung dafür ist die Verwendung einer 32 Bit unsigned Integer-Variable für die Prozessorzyklen, wie in Abschnitt 5.4.1.1 erläutert. Da alle 32 Bits in etwa gleichverteilt sind, umfasste der Testzeitraum mindestens

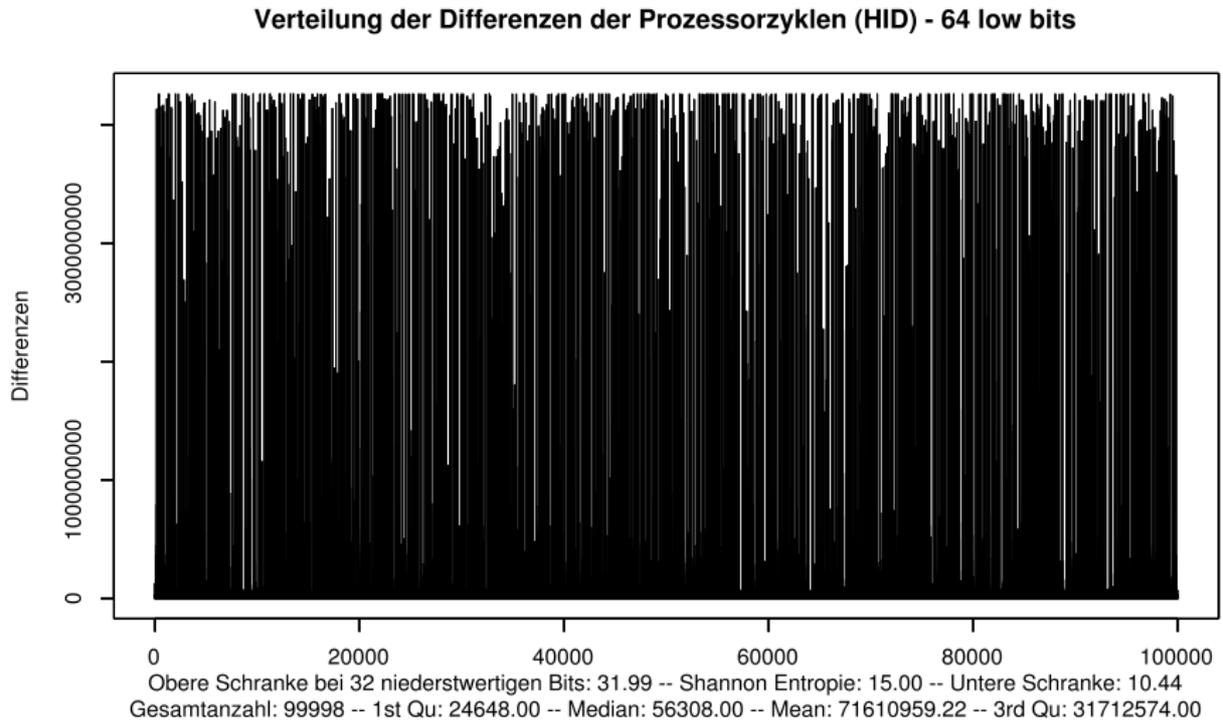
$$2^{32} ns \approx 4,3 sec$$

Der Testdurchlauf dauerte ca. 3 Stunden.

6.2.4.2.2 Differenzen der Prozessorzyklen

In einem weiteren Test werden die Differenzen der Prozessorzyklen bei direkt aufeinander folgenden Ereignissen untersucht. Der folgende Plot stellt diese Differenzen mit 64 Bit dar -

es ist dabei ausgehend von der oben stehenden Tabelle klar, dass auch die Differenzen niemals größer als 32 Bit werden können.

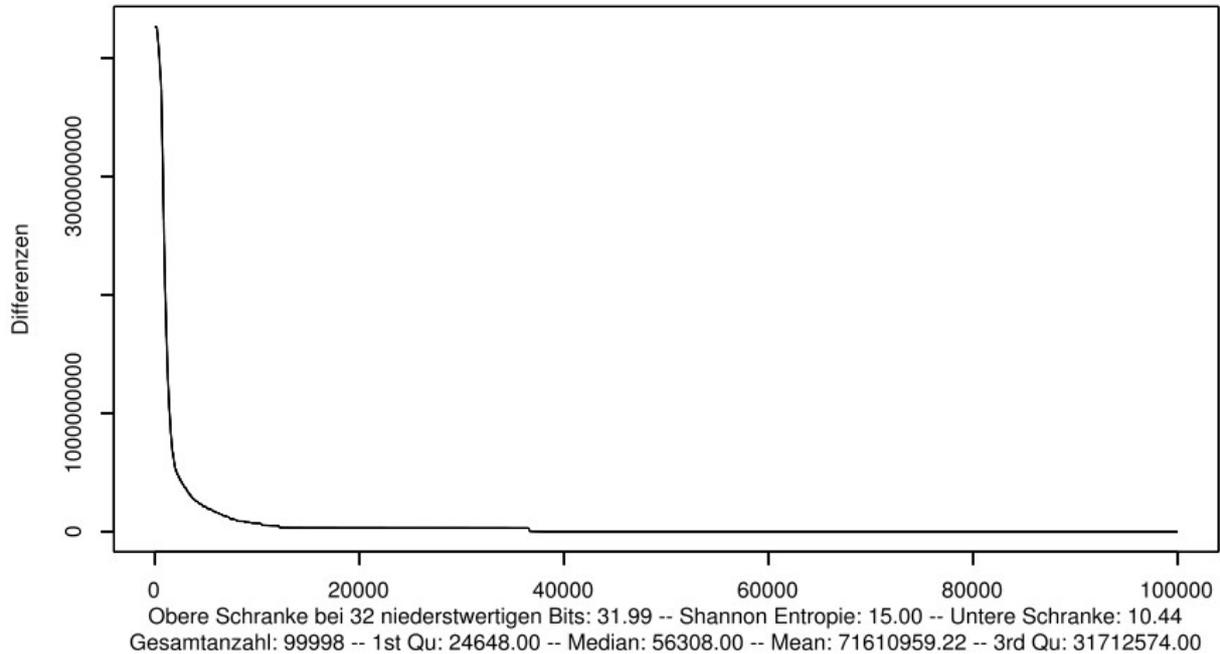


Der Plot enthält zusätzlich folgende empirische Informationen bezüglich der Stichprobe der Prozessorzyklen:

- Obere Schranke der Entropie
- Shannon Entropie
- Untere Schranke der Entropie
- Gesamtzahl der betrachteten Ereignisse (jede Abweichung von der Stichprobengröße zeigt wiederum auf die in Anhang A.1.1 erläuterte Messungenauigkeit).

Folgende Abbildung zeigt die sortierten Differenzen der Prozessorzyklen. Zusätzlich sind nochmals die gleichen Werte wie in der vorangegangenen Abbildung aufgelistet.

Geordnete Verteilung der Differenzen der Prozessorzyklen (HID) - 64 low bits

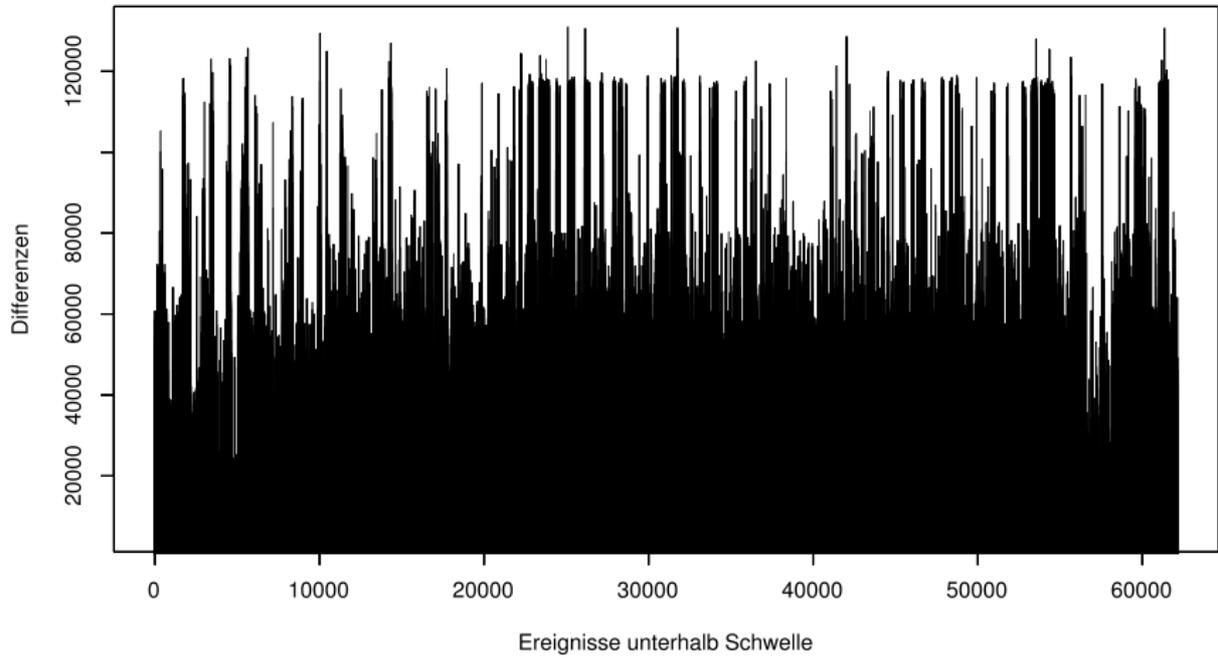


Dieser „Hockey-Stick“ lässt sich erklären, wenn man die Differenzen der Prozessorzyklen genauer betrachtet: jeder Tastendruck und jedes Loslassen einer Taste erzeugt 3 Ereignisse, die in den `input_pool` eingefügt werden. Dabei hat das erste Ereignis einen viel höheren Wert für die Differenz zum vorhergehenden Ereignis, als die 2 nachfolgenden Ereignisse, da diese noch im Code-Pfad zur Abarbeitung der Bedienung einer Taste liegen und damit nicht direkt vom Benutzer beeinflussbar sind. Folgende Tabelle stellt exemplarisch dieses Zusammenspiel dar:

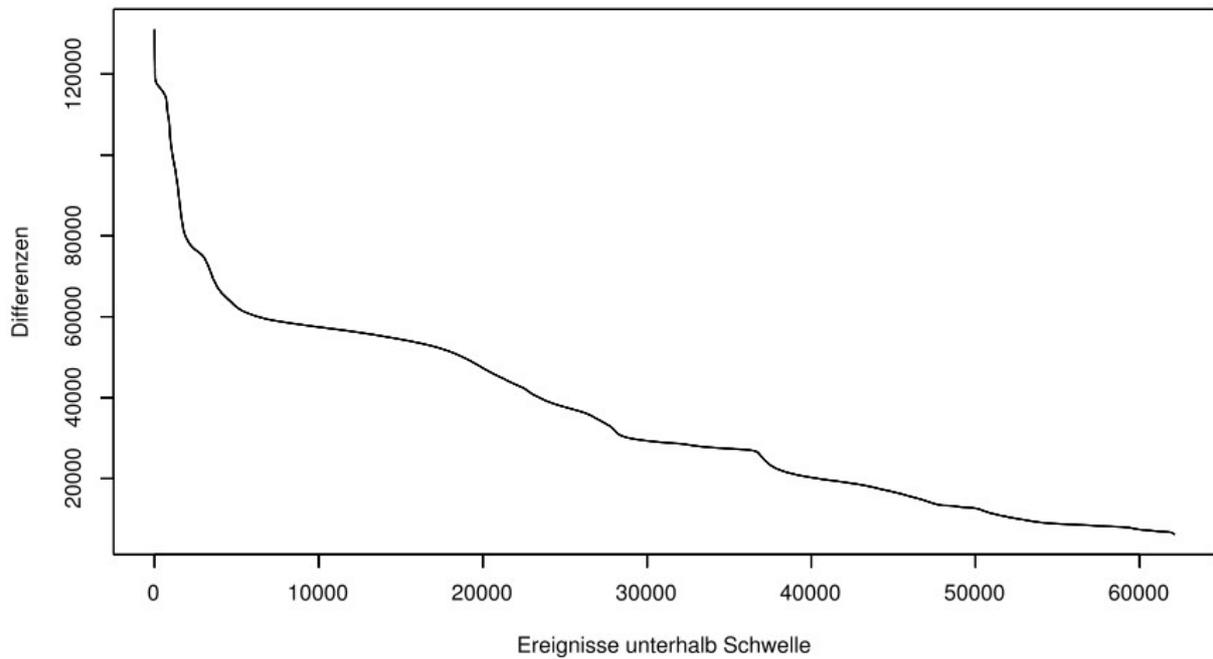
Differenzen von aufeinander-folgenden Prozessorzyklen:
31438098
38799
20898
31133889
46008
19939
31666410
43781
21060

Die kleineren Differenzen sind immer unter der Schwelle von 2^{17} . Demzufolge wird der Graph mit den Prozessorzyklen im Folgenden nochmal dargestellt, aber diesmal separat für die Daten über und unter der Schwelle.

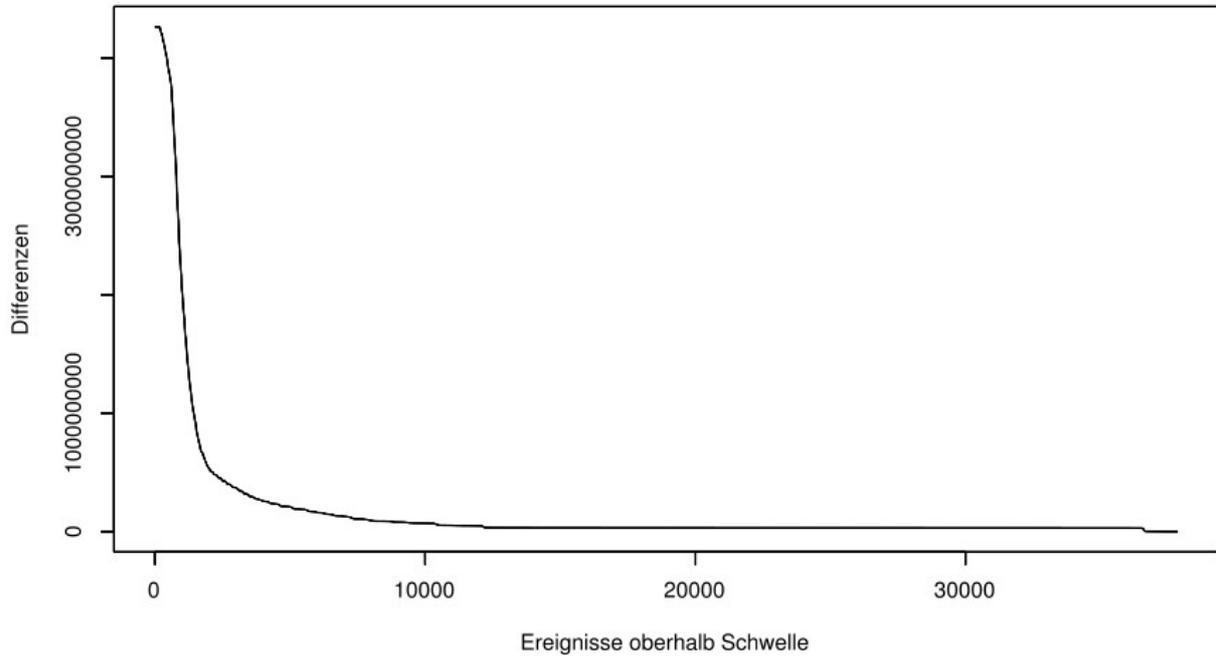
Verteilung der Differenzen der Prozessorzyklen unterhalb Schwelle (HID) - 64 low bits



Geordnete Verteilung der Differenzen der Prozessorzyklen unterhalb Schwelle (HID) - 64 low bits

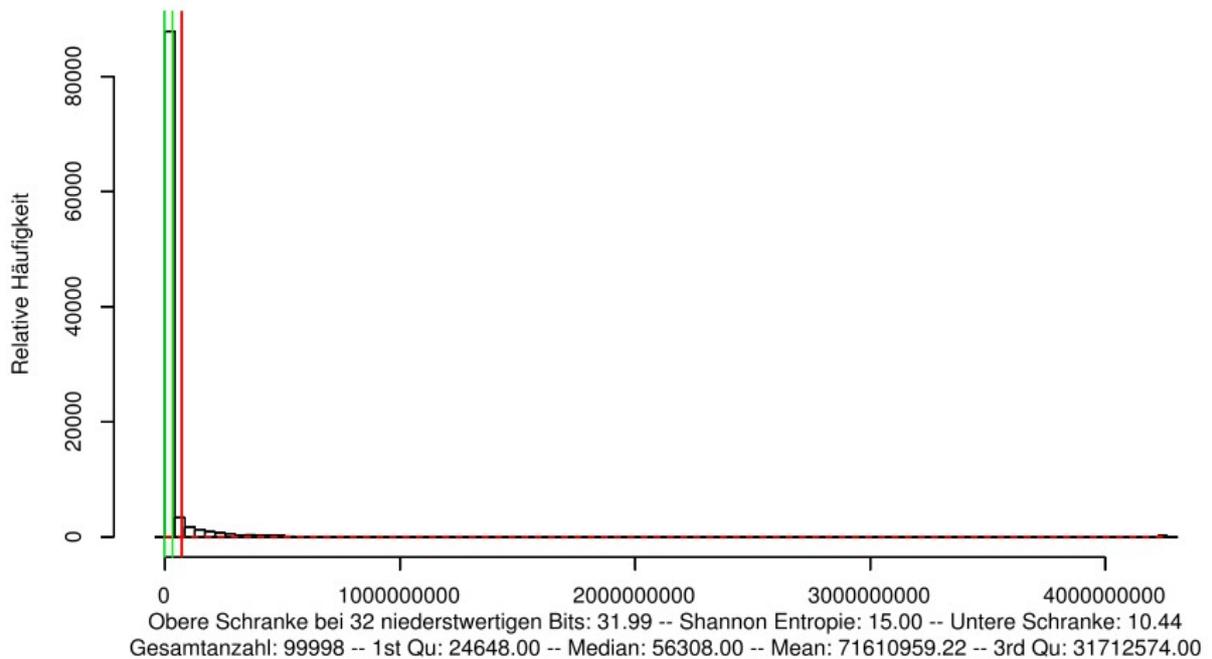


Geordnete Verteilung der Differenzen der Prozessorzyklen oberhalb Schwelle (HID) - 64 low bits



Die nächste Abbildung zeigt das Histogramm der empirischen Verteilung der Differenz von Prozessorzyklen. Wiederum sind die gleichen Werte wie in der vorangegangenen Abbildung aufgelistet.

Histogramm der Differenz der Prozessorzyklen (HID) - 64 low bits

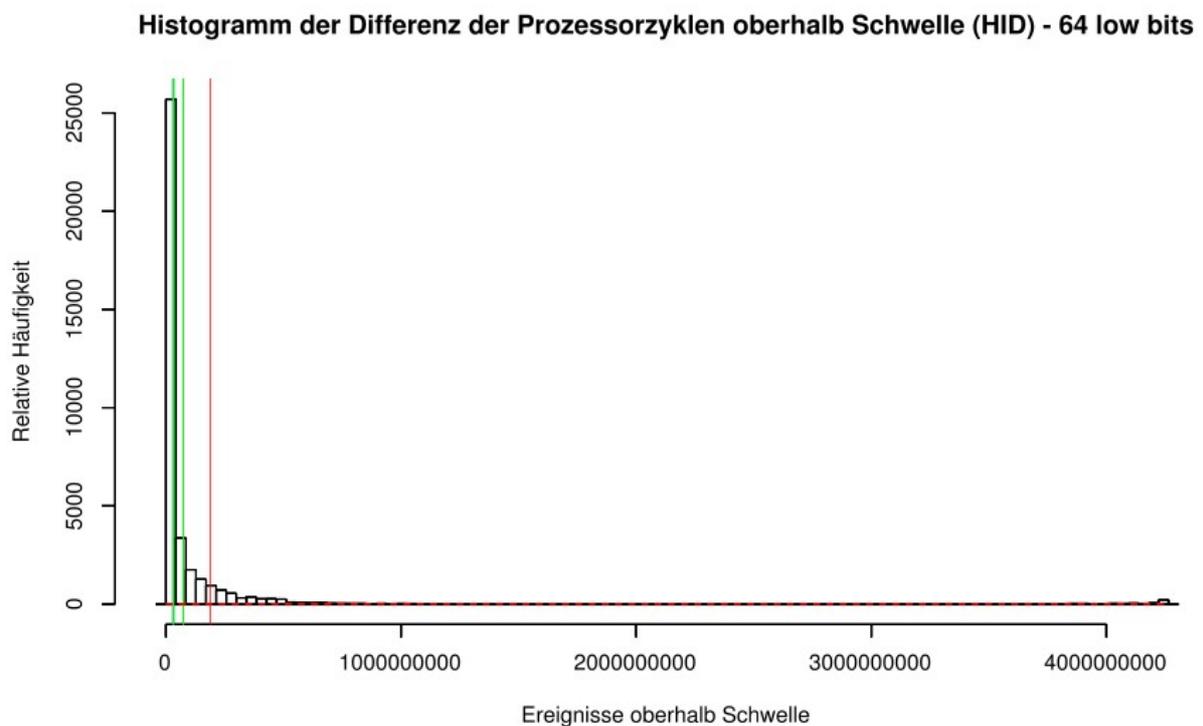
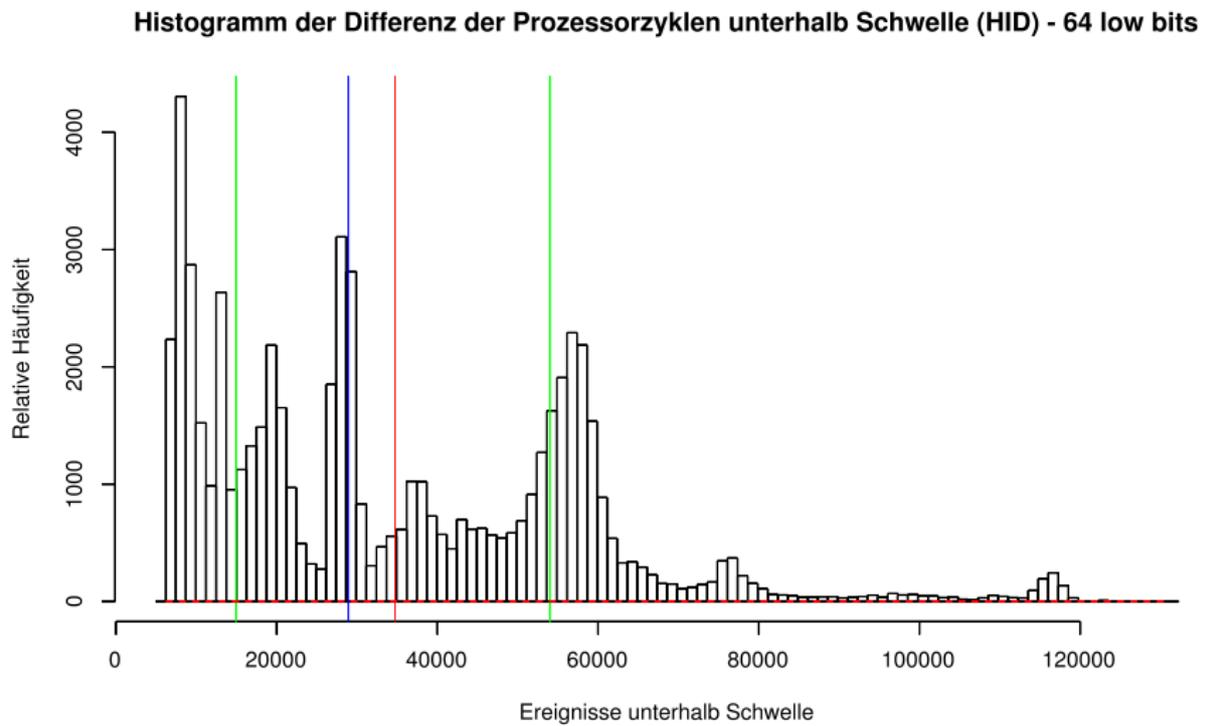


Das Histogramm enthält folgende Visualisierungen:

- Die Quartile sind mit grünen Linien markiert.
- Der Mittelwert ist mit einer roten Linie markiert.

- Der Medianwert ist mit einer blauen Linie markiert.

Ebenfalls wird das gleiche Histogramm mit den Werten über und unter der Schwelle dargestellt.



6.2.4.2.3 Interpretation der Ergebnisse

Die Prozessorzyklen werden mit einer Variable aufgezeichnet, die

$$2^{32} ns = \frac{2^{32}}{10^9} sec \approx \frac{2^{32}}{2^{30}} sec = 2^2 sec$$

umfaßt Aufgrund der Größe der Variable werden alle Bits abhängig vom Beobachtungszeitraum häufig verändert.

Da die Zeitvarianzen zwischen zwei Ereignissen häufiger über der maximalen Größe der Variable liegen²⁷, werden auch die höherwertigen Bits der Zeitvarianzen entsprechend häufiger verändert.

Folgende Aussagen können getroffen werden:

- Die Verteilung der Differenzen der Prozessorzyklen ist relativ gleichförmig und zeigt nur sehr wenige Verzerrungen. Es wird derzeit vermutet, dass die aufgezeichneten Verzerrungen aus der Art und Weise der Messungen resultieren. Der Tester hat häufig die Maus bewegt und gelegentlich die Tastatur benutzt. Bei der Bewegung der Maus werden durch die hohe Auflösung der Messungen der Mauskoordinaten durch den Maus-Treiber viel häufiger Ereigniswerte aufgezeichnet (eine Messung pro wenige Millisekunden). Im Gegensatz dazu sind die Zeitabstände der Tastendrücke im höheren zweistelligen oder dreistelligen Millisekunden-Bereich. Es wird vermutet, dass durch die genannten zeitlichen Abstände der Ereignisse für Tastatur und Maus einige Spitzen im Bereich der Maus-Messungen und Tastaturmessungen zu sehen sind.
- Auf Basis der beobachteten Daten für die Prozessorzyklen können die folgenden theoretischen Werte berechnet werden:
 - **Obere Schranke der Entropie: 31,99 Bits**
 - **Shannon-Entropie: 15,0 Bits**
 - **Untere Schranke der Entropie: 10,44 Bits**
- **Die theoretischen Entropiewerte sind sehr ähnlich zu denen der beobachteten Prozessorzyklen für die Blockgeräte.**

Für die praktische Analyse der Entropiequelle, kann folgendes gesagt werden: Die Anzahl der niederwertigen Bits, welche für die Entropie-Berechnungen verwendet wird, hängt von der Größe der Zeitvarianzen ab. Wenn man annimmt, dass diese Varianzen maximal mehrere Tage sind (Annahme eines Headless Serversystems, an dessen Konsole ein Administrator nur alle paar Tage geht) und minimal einige Millisekunden sind, sind immer noch eine gewisse Anzahl an Bits in der Variable der Prozessorzyklen vorhanden, die für eine Entropie-Berechnung herangezogen werden kann. Diese Überlegungen sind konsistent mit den theoretischen Entropiewerten.

6.2.4.3 Testresultate für Blockgeräte

Die gesetzte Stichprobenanzahl ist:

$$S = 100.000$$

6.2.4.3.1 Verteilung der Bits der Prozessorzyklen

Die empirische Analyse der Verteilung der einzelnen Bits in den Prozessorzyklen ist in folgender Tabelle dargestellt. Dabei können folgende Informationen aus der Tabelle gelesen werden:

- Die linke Spalte spezifiziert das diskutierte Bit beginnend mit dem signifikantesten (d.h. dem linken Bit) in absteigender Reihenfolge.
- Die rechte Spalte listet die Wahrscheinlichkeit p_{1i} eines gesetzten Bitwertes auf.
- Auf Basis der rechten Spalte, kann man mittels $1 - p_{1i} = p_{0i}$ die Gegenwahrscheinlichkeit berechnen.

Diese Tabelle basiert auf den Daten in der Datei raw_entropy_disk_cycles-histtable-64.txt.

²⁷ Einen Kaffee holt man nun mal nicht innerhalb von 4 Sekunden - damit kommen in dieser Zeit keine Ereignisse von Eingabegeräten. ☺

Bitposition	Wahrscheinlichkeit für gesetztes Bit	Bitposition	Wahrscheinlichkeit für gesetztes Bit
1	0	33	0.492269845396908
2	0	34	0.497039940798816
3	0	35	0.509580191603832
4	0	36	0.497089941798836
5	0	37	0.50385007700154
6	0	38	0.497159943198864
7	0	39	0.497609952199044
8	0	40	0.501120022400448
9	0	41	0.497869957399148
10	0	42	0.500330006600132
11	0	43	0.499849996999994
12	0	44	0.499439988799776
13	0	45	0.50160003200064
14	0	46	0.498779975599512
15	0	47	0.501530030600612
16	0	48	0.501240024800496
17	0	49	0.498509970199404
18	0	50	0.5
19	0	51	0.500320006400128
20	0	52	0.4997499949999
21	0	53	0.49909998199964
22	0	54	0.498309966199324
23	0	55	0.499769995399908
24	0	56	0.498989979799596
25	0	57	0.501340026800536
26	0	58	0.499979999599992
27	0	59	0.50230004600092
28	0	60	0.501010020200404
29	0	61	0.501360027200544
30	0	62	0.501280025600512
31	0	63	0.553291065821316
32	0	64	0.370817416348327

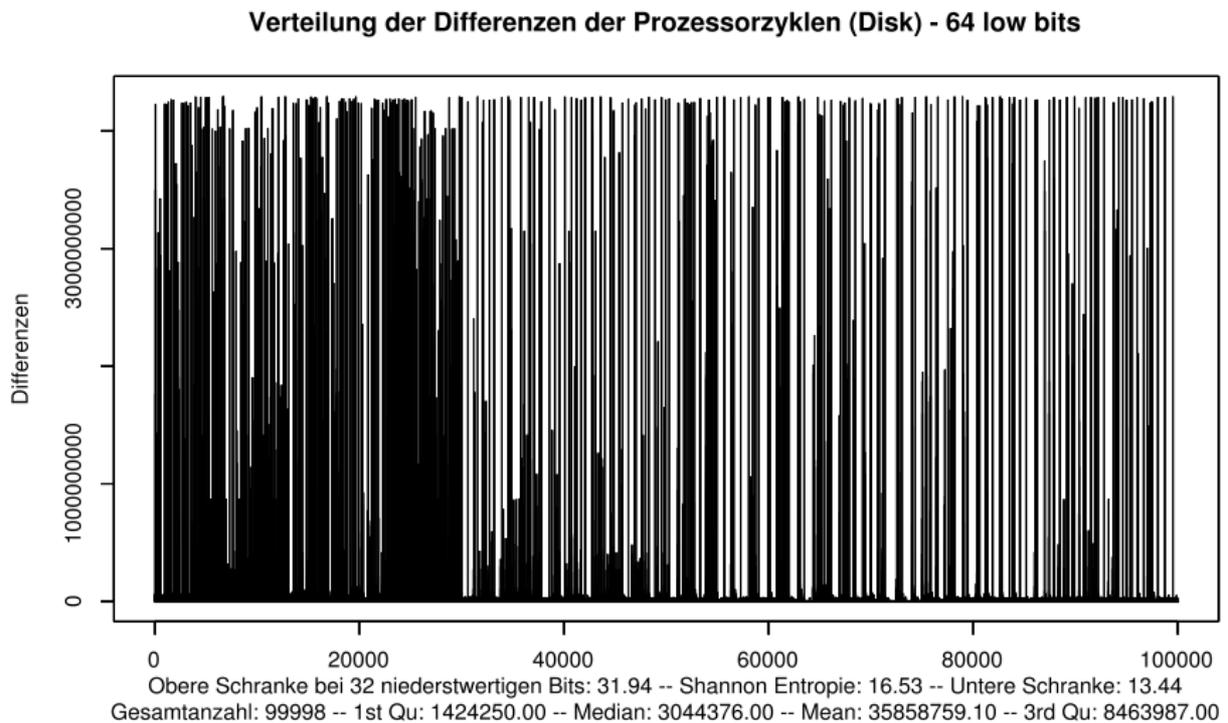
Wie in der Tabelle zu sehen ist, werden nur die unteren 32 Bits verändert. Die Erklärung dafür ist die Verwendung einer 32 Bit unsigned Integer-Variable für die Prozessorzyklen, wie in Abschnitt 5.4.1.1 erläutert. Da alle 32 Bits in etwa gleichverteilt sind, umfasste der Testzeitraum mindestens

$$2^{32} ns \approx 4,3 sec$$

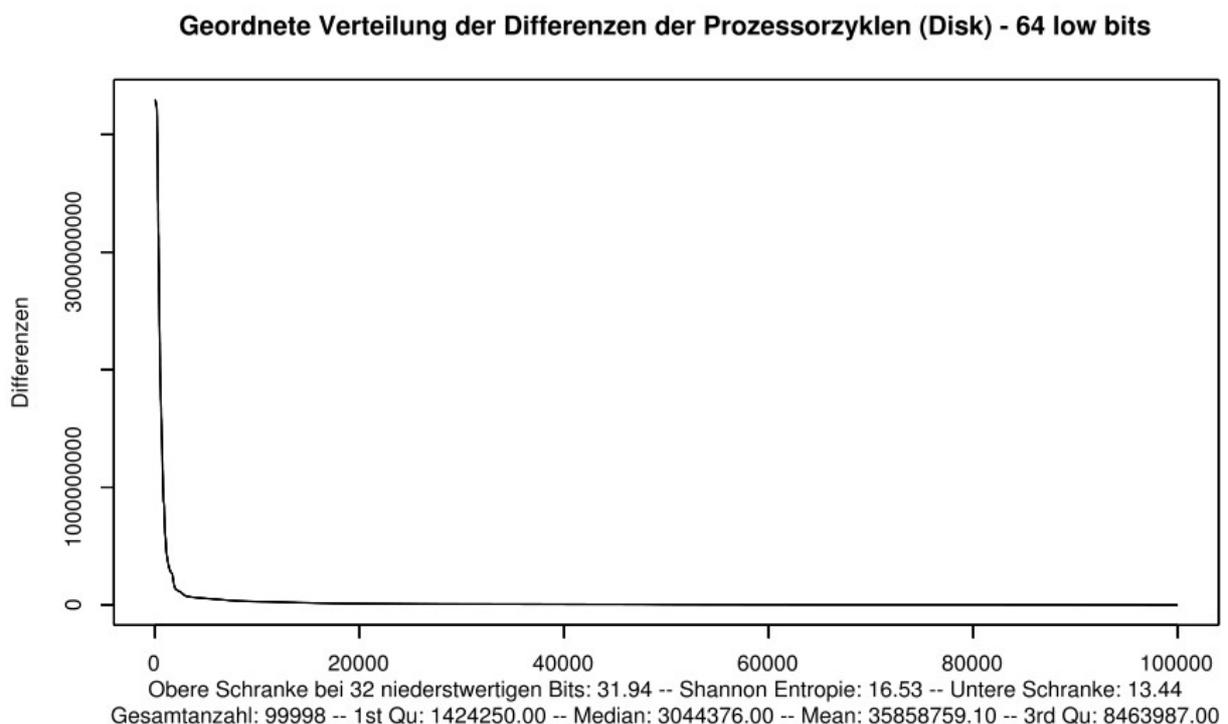
Der Testdurchlauf dauerte ca. 3 Stunden.

6.2.4.3.2 Differenzen der Prozessorzyklen

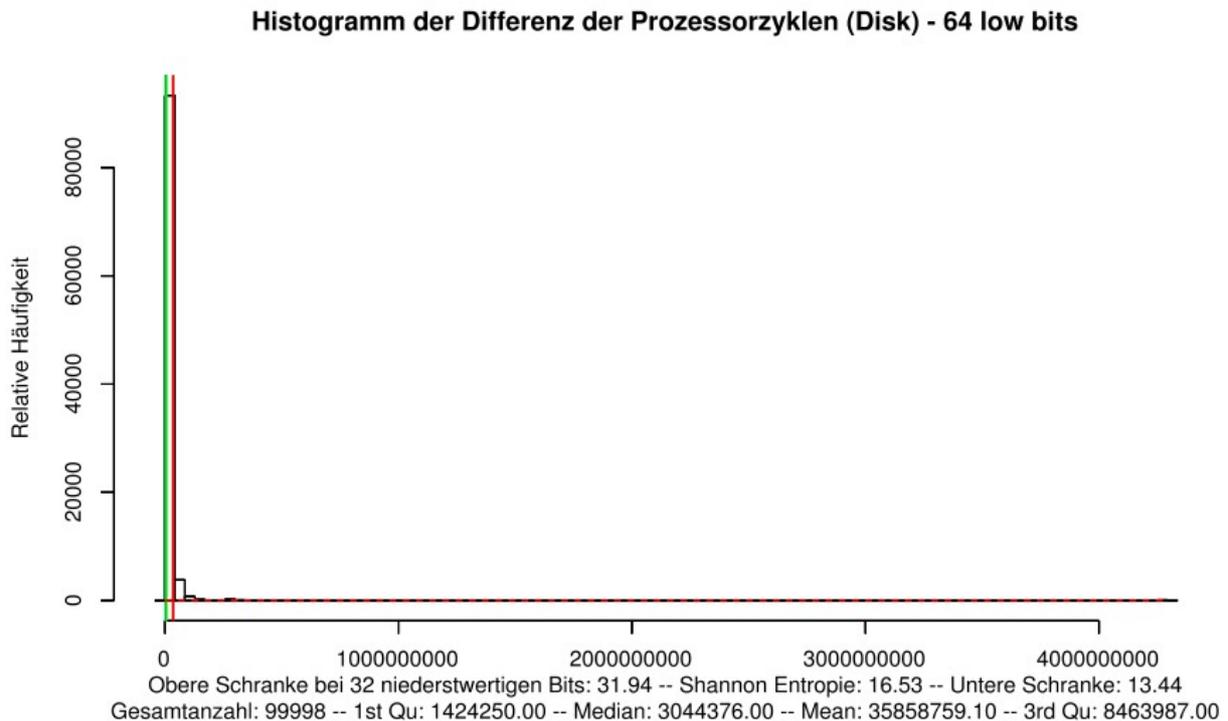
In einem weiteren Test werden die Differenzen der Prozessorzyklen bei direkt aufeinander folgenden Ereignissen untersucht. Der folgende Plot stellt diese Differenzen mit 64 Bit dar.



Folgende Abbildung zeigt die sortierten Differenzen der Prozessorzyklen. Zusätzlich sind nochmals die gleichen Werte wie in der vorangegangenen Abbildung aufgelistet.



Die nächste Abbildung zeigt das Histogramm der empirischen Verteilung der Differenzen von Prozessorzyklen.



Das Histogramm enthält folgende Visualisierungen:

- Die Quartile sind mit grünen Linien markiert.
- Der Mittelwert ist mit einer roten Linie markiert.
- Der Medianwert ist mit einer blauen Linie markiert.

6.2.4.3.3 Interpretation der Ergebnisse

Die generelle Interpretation der Prozessorzyklen wird in Abschnitt 6.2.4.2.3 gegeben.

Es können folgende Aussagen getroffen werden:

- Die Verteilung der Differenzen der Prozessorzyklen ist relativ gleichmäßig. Spitzen wechseln sich gleichmäßig mit weniger häufig gesetzten Bits ab.
- Auf Basis der beobachteten Daten für die Prozessorzyklen können die folgenden theoretischen Werte berechnet werden:
 - **Obere Schranke der Entropie: 31,94 Bits**
 - **Shannon-Entropie: 16,51 Bits**
 - **Untere Schranke der Entropie: 13,44 Bits**

Die theoretischen Entropiewerte sind sehr ähnlich zu denen der beobachteten Prozessorzyklen für die Eingabegeräte.

Für die praktische Analyse der Entropiequelle kann folgendes gesagt werden: Die Anzahl der niederwertigen Bits, welche für die Entropie-Berechnungen verwendet wird, hängt von der Größe der Zeitvarianzen ab. Wenn man annimmt, dass diese Varianzen maximal eine Minute sind (der Kern führt mindestens einmal pro Minute eine Datensynchronisation mit der Festplatte durch) und minimal einige hundert Mikrosekunden sind, sind immer noch eine gewisse Anzahl an Bits in der Variable der Prozessorzyklen vorhanden, die für eine

Entropie-Berechnung herangezogen werden kann. Diese Überlegungen sind konsistent mit den theoretischen Entropiewerten.

6.2.5 Häufigkeitsverteilung und Entropie der Jiffies

Dieser Abschnitt betrachtet ausschließlich die Verteilung der Jiffies („least significant word“).

Die Analyse ist analog der Häufigkeitsverteilung der Prozessorzyklen aufgebaut, da beide Werte einen Zeitstempel angeben. Die angewandte Methodik in diesen Unterabschnitten basiert auf der in Abschnitt 6.2.4 vorgestellten Methodik, mit der Abwandlung, dass der Jiffies-Wert anstelle der Prozessorzyklen betrachtet wird.

6.2.5.1 Testdurchführung

Ausgehend von diesen Anfangsüberlegungen wurden folgende Tests für die Hardware-Klassen für Eingabegeräte und Blockgeräteaktivität definiert:

- Die SystemTap-Skripte werden mit dem Bash-Skript `gendata.sh` initialisiert.
- Die SystemTap-Skripte „`raw_entropy_hid.stp`“ und „`raw_entropy_disk.stp`“ zeichnen die Ereigniswerte für die jeweilige Klasse auf. Dabei werden die 64 Bit-Werte der Jiffies aufgezeichnet. Diese Ergebnisse erlauben die graphische Darstellung einer Häufigkeitsverteilung der auftretenden Jiffies und deren Varianzen und eine Entropieabschätzung über die gewonnenen Ereigniswerte durchzuführen. Die Stichprobengröße wird mit der Variable „`num_samplings`“ in den SystemTap-Skripten festgelegt.
- Die R-Project-Programme `raw_entropy_hid.r` und `raw_entropy_disk.r` erstellen die folgenden Graphiken aus den erzeugten Datenreihen. Dabei kann die Analyse auf die niederwertigen Bits mittels Setzen der Variable `significantbits` in den R-Project-Programmen auf einen beliebigen Wert zwischen 1 und 63 erfolgen.

Die Interrupts wurden nicht getestet, da die Jiffies bereits in den Wörtern des `fast_pools` berücksichtigt sind. Beim Einmischen der Zustände der `fast_pools` in den `input_pool` werden keine Jiffies mehr berücksichtigt. Demzufolge umfasst der Test in Abschnitt 6.2.3.4 bereits die Jiffies.

6.2.5.2 Testresultate für Eingabegeräte

Die gesetzte Stichprobenanzahl ist:

$$S = 100.000$$

6.2.5.2.1 Verteilung der Bits der Jiffies

Die empirische Analyse der Verteilung der einzelnen Bits in den Jiffies ist in folgender Tabelle dargestellt. Dabei können folgende Informationen aus der Tabelle gelesen werden:

- Die linke Spalte spezifiziert das diskutierte Bit beginnend mit dem signifikantesten (d.h. dem linken) Bit in absteigender Reihenfolge.
- Die rechte Spalte listet die Wahrscheinlichkeit p_{1i} eines gesetzten Bitwertes auf.
- Auf Basis der rechten Spalte, kann man mittels $1 - p_{1i} = p_{0i}$ die Gegenwahrscheinlichkeit berechnen.

Diese Tabelle basiert auf den Daten in der Datei `raw_entropy_hid_jiffies-histtable-64.txt`.

Bitposition	Wahrscheinlichkeit für gesetztes Bit	Bitposition	Wahrscheinlichkeit für gesetztes Bit
1	0	33	0
2	0	34	0
3	0	35	0
4	0	36	0
5	0	37	1

Bitposition	Wahrscheinlichkeit für gesetztes Bit	Bitposition	Wahrscheinlichkeit für gesetztes Bit
6	0	38	0

Bitposition	Wahrscheinlichkeit für gesetztes Bit	Bitposition	Wahrscheinlichkeit für gesetztes Bit
7	0	39	0

Bitposition	Wahrscheinlichkeit für gesetztes Bit	Bitposition	Wahrscheinlichkeit für gesetztes Bit
8	0	40	0
9	0	41	0.457210293132245
10	0	42	0.542789706867755
11	0	43	0.381501965216174
12	0	44	0.806888757763354
13	0	45	0.526167878466631
14	0	46	0.513166448309314
15	0	47	0.68426526917961
16	0	48	0.49861484763324
17	0	49	0.561641780595866
18	0	50	0.565262178839672
19	0	51	0.502605286581524
20	0	52	0.476102371260839
21	0	53	0.493164248067287
22	0	54	0.516456810249127
23	0	55	0.489593855324086
24	0	56	0.497084679314725
25	0	57	0.496024562701897
26	0	58	0.499354929042195
27	0	59	0.500065007150787
28	0	60	0.4980247827261
29	0	61	0.501645180969907
30	0	62	0.501045114962646
31	0	63	0.500825090759984
32	1	64	0.500725079758773

Wie in der Tabelle zu sehen ist, sind die Bits Nr.45 bis Nr.64 in etwa gleichverteilt. Die Bits Nr.1 bis Nr.44 ändern sich frühestens alle

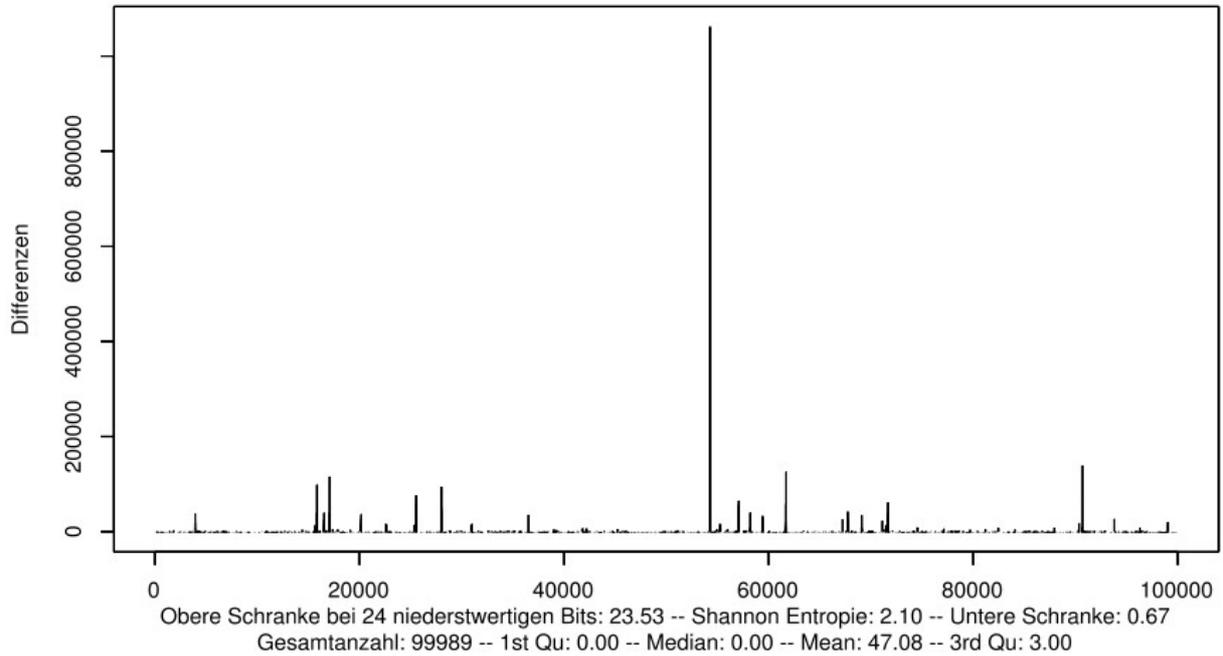
$$2^{64-44} * 4ms = 2^{15} * 4ms \approx 4200 sec \approx 70 min$$

und bleiben deshalb im Betrachtungszeitraum konstant (Bits Nr.1 bis Nr.42) oder ändern sich nur selten (Bits Nr.43 und Nr.44), sodass keine annähernde Gleichverteilung gewährleistet ist. Die Bits Nr.45 bis Nr.49 werden für eine annähernde Gleichverteilung ebenfalls noch zu selten geändert. Der Testdurchlauf dauerte ca. 3 Stunden. Die Diskrepanz zu den berechneten 70 Minuten ist in Bit Nr.43 zu finden: eine Änderung impliziert die Verdoppelung der Zeit auf 140 Min, also 2 ½ Stunden.

6.2.5.2.2 Differenzen der Jiffies

In einem weiteren Test werden die Differenzen der Jiffies bei direkt aufeinander folgenden Ereignissen untersucht. Der folgende Plot stellt diese Differenzen dar.

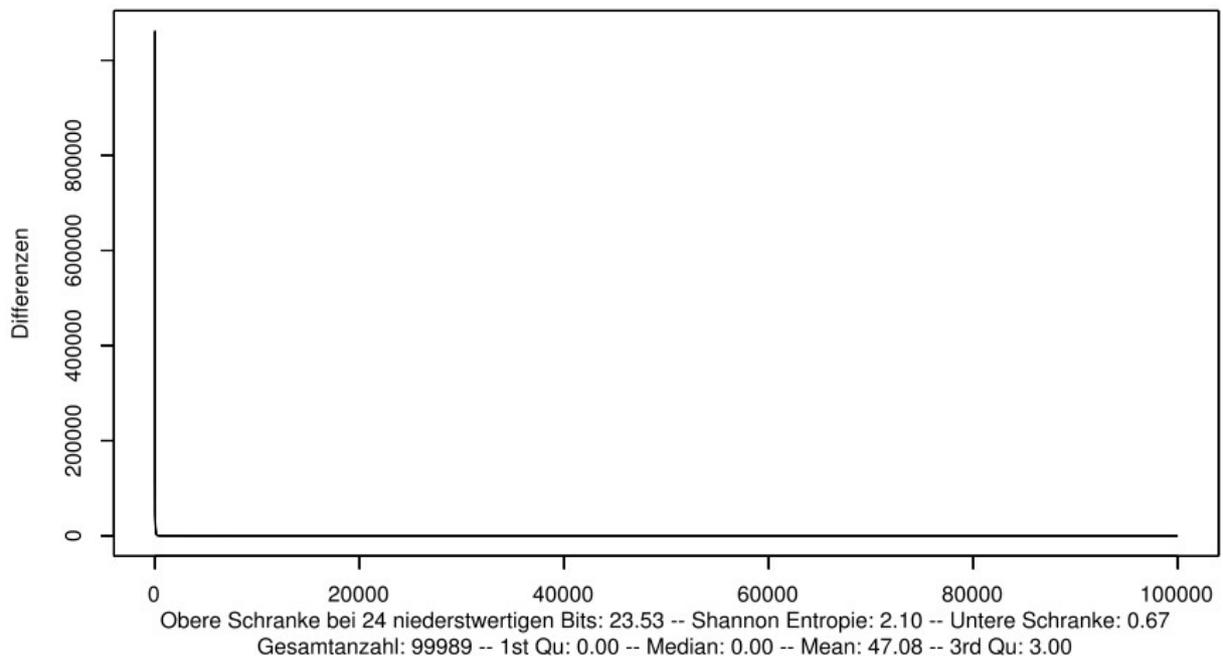
Verteilung der Differenzen der Jiffies (HID) - 64 low bits



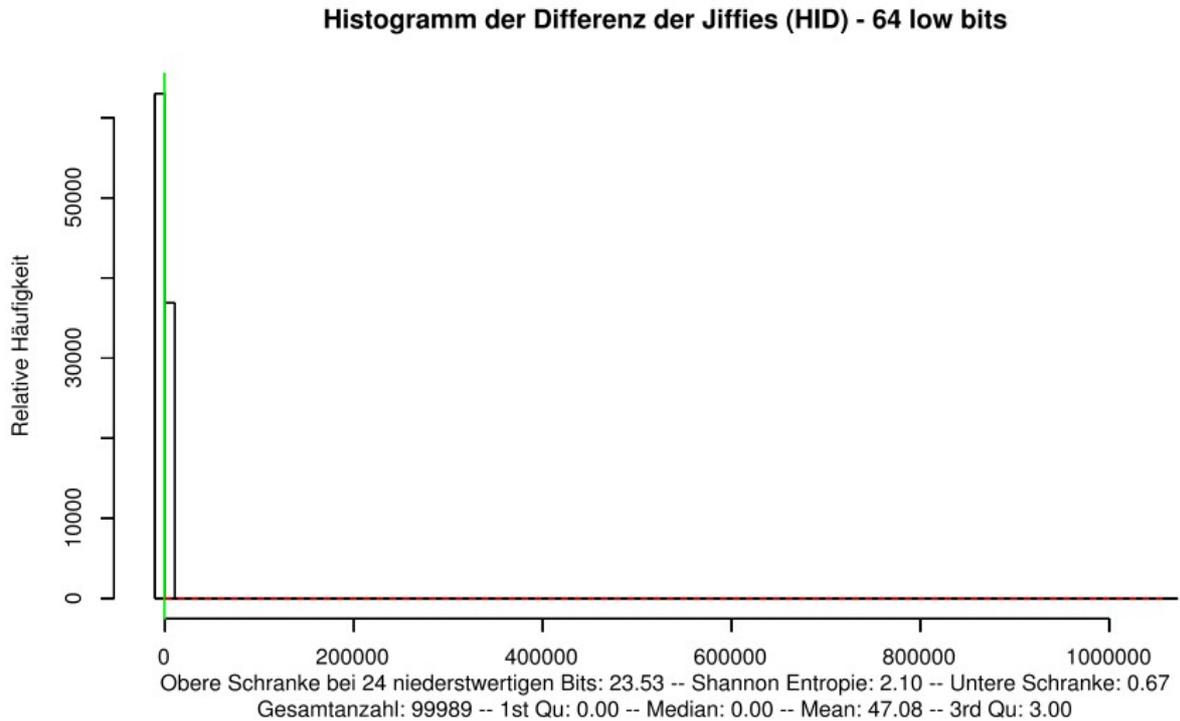
Der Plot enthält zusätzlich folgende empirische Informationen bezüglich der Stichprobe der Jiffies: Wir bemerken, dass die obere Schranke der Shannon-Entropie unter der Annahme, dass die Bits unabhängig sind nicht berechenbar ist aufgrund der unveränderten höherwertigen Bits. Deswegen gibt das Histogramm die obere Schranke basierend auf den allen sich ändernden Bits an.

Folgende Abbildung zeigt die sortierten Differenzen der Jiffies.

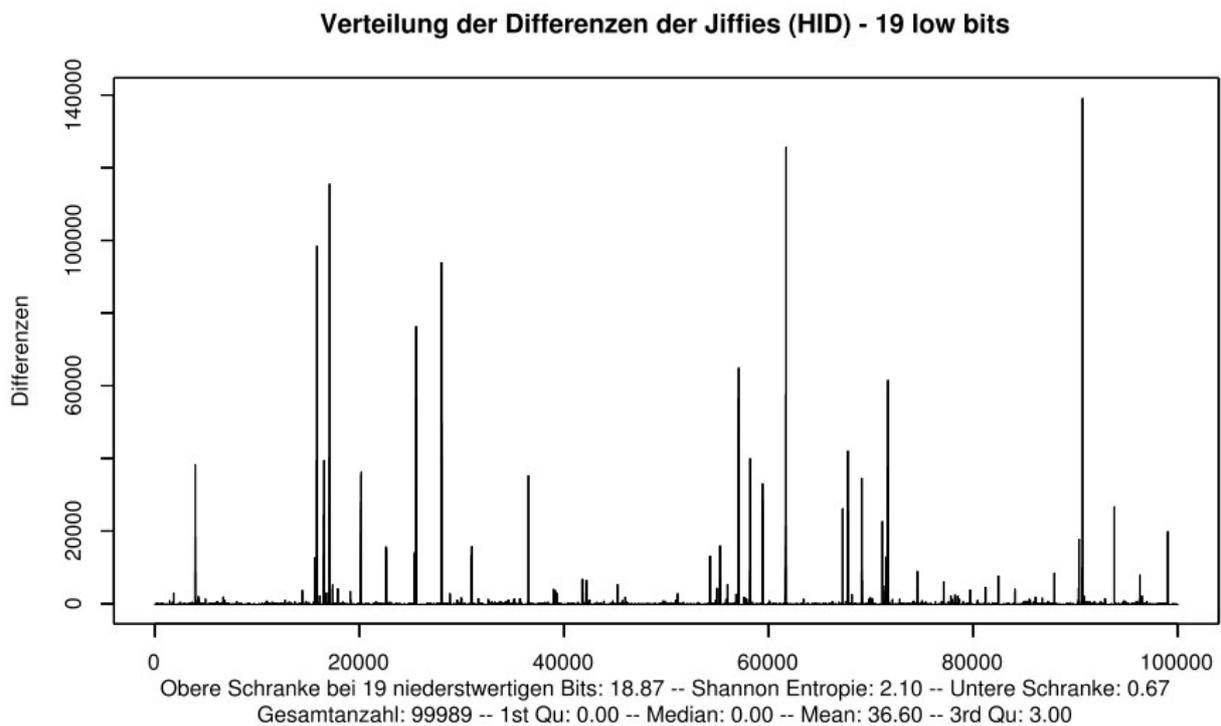
Geordnete Verteilung der Differenzen der Jiffies (HID) - 64 low bits



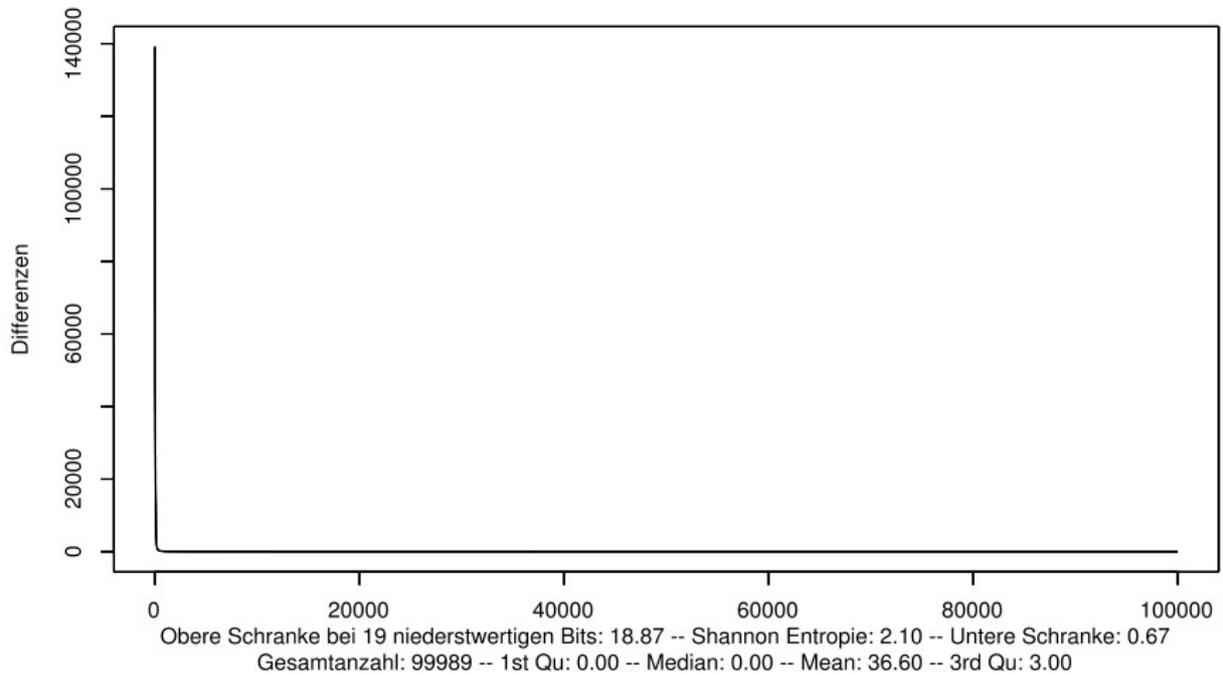
Die nächste Abbildung zeigt das Histogramm der empirischen Verteilung der Differenz von Jiffies. Wiederum sind die gleichen Werte wie in der vorangegangenen Abbildung aufgelistet.



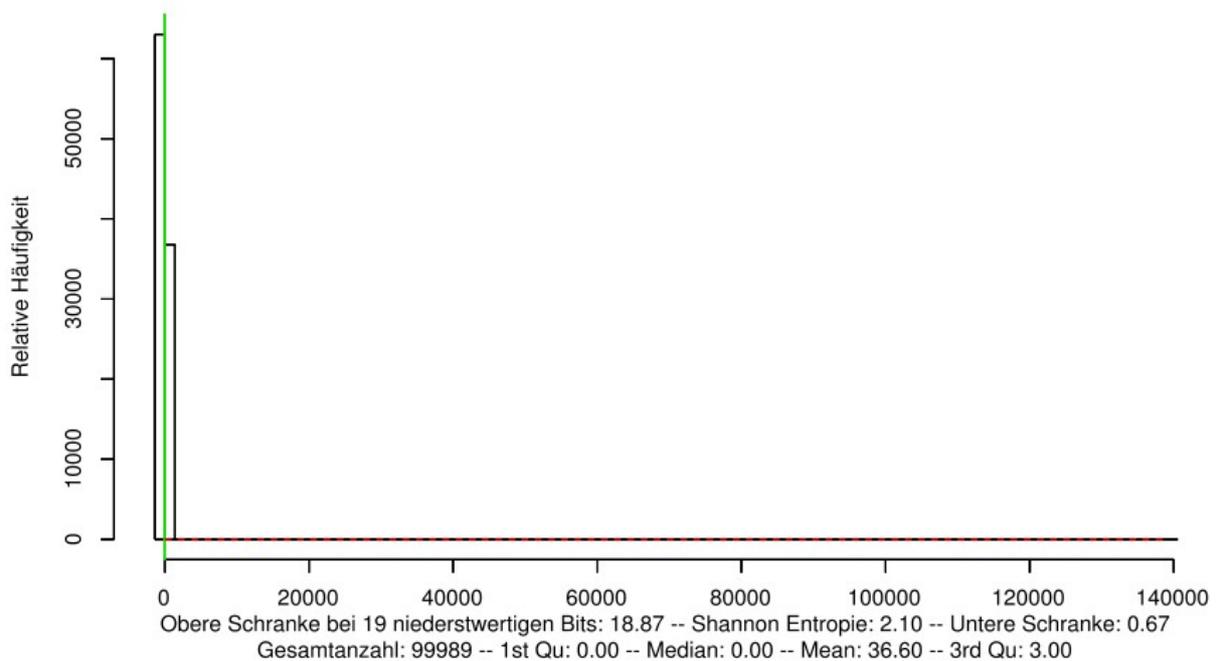
Die gleichen Graphen bezogen auf die 19 (64 - 45) niederwertigen Bits sind im Folgenden dargestellt.



Geordnete Verteilung der Differenzen der Jiffies (HID) - 19 low bits



Histogramm der Differenz der Jiffies (HID) - 19 low bits



6.2.5.2.3 Interpretation der Ergebnisse

Die Jiffies werden mit einer Variable aufgezeichnet, die

$$2^{32} * \frac{1}{250} \text{ sec} \approx \frac{2^{32}}{2^8} \text{ sec} = 2^{24} \text{ sec}$$

aufzeichnen kann. Aufgrund der Größe der Variable werden die höherwertigen Bits abhängig vom Beobachtungszeitraum vergleichsweise selten verändert. Entsprechend werden die höherwertigen Bits der Zeitvarianzen ebenfalls vergleichsweise selten verändert.

Wenn man diese höherwertigen Bits abschneidet und sich auf die niederwertigen Bits konzentriert, können folgende Aussagen getroffen werden:

- Die Verteilung der Differenzen der Jiffies ist relativ gleichförmig und zeigt wenige Verzerrungen. Es wird vermutet, dass die aufgezeichneten Spikes von der in Abschnitt 6.2.4.2.3 diskutierten Unterschiede im Auftreten der Ereignisse stammen.
- Auf Basis der beobachteten Daten für die Jiffies können die folgenden theoretischen Werte berechnet werden:
 - **Obere Schranke der Entropie: 23,53 Bits**
 - **Shannon-Entropie: 2,10 Bits**
 - **Untere Schranke der Entropie: 0,67 Bits**
- **Die theoretischen Entropiewerte sind sehr ähnlich zu denen der beobachteten Jiffies für die Blockgeräte.**

Die Vernachlässigung der höherwertigen Bits kann als akzeptabel eingestuft werden, da hierbei potentiell vorhandene Entropie einfach nicht betrachtet wird. Demzufolge ist das Weglassen eine Worst-Case-Abschätzung.

Für die praktische Analyse der Entropiequelle kann folgendes gesagt werden: Die Anzahl der niederwertigen Bits, welche für die Entropie-Berechnungen verwendet wird, hängt von der Größe der Zeitvarianzen ab. Wenn man annimmt, dass diese Varianzen maximal mehrere Tage sind (Annahme eines Headless-Serversystems, an dessen Konsole ein Administrator nur alle paar Tage geht) und minimal einige Millisekunden sind, sind kaum noch Bits in der Variable der Jiffies vorhanden, die für eine Entropie-Berechnung herangezogen werden können. Demzufolge ist die erwartete Entropie auf Basis der Jiffies relativ gering. Diese Überlegungen sind konsistent mit den theoretischen Entropiewerten.

6.2.5.3 Testresultate für Blockgeräte

Die gesetzte Stichprobenanzahl ist: $S = 100.000$

6.2.5.3.1 Verteilung der Bits der Jiffies

Die empirische Analyse der Verteilung der einzelnen Bits in den Jiffies ist in folgender Tabelle dargestellt. Dabei können folgende Informationen aus der Tabelle gelesen werden:

- Die linke Spalte spezifiziert das diskutierte Bit beginnend mit dem signifikantesten (d.h. dem linken Bit) in absteigender Reihenfolge.
- Die rechte Spalte listet die Wahrscheinlichkeit p_{1i} eines gesetzten Bitwertes auf.
- Auf Basis der rechten Spalte, kann man mittels $1 - p_{1i} = p_{0i}$ die Gegenwahrscheinlichkeit berechnen.

Diese Tabelle basiert auf den Daten in der Datei raw_entropy_disk_jiffies-histtable-64.txt.

Bitposition	Wahrscheinlichkeit für gesetztes Bit	Bitposition	Wahrscheinlichkeit für gesetztes Bit
1	0	33	0
2	0	34	0
3	0	35	0

Bitposition	Wahrscheinlichkeit für gesetztes Bit	Bitposition	Wahrscheinlichkeit für gesetztes Bit
4	0	36	0
5	0	37	0
6	0	38	1
7	0	39	0.873357467149343
8	0	40	0.126642532850657
9	0	41	0.842626852537051
10	0	42	0.899887997759955
11	0	43	0.715294305886118
12	0	44	0.5657513150263
13	0	45	0.410068201364027
14	0	46	0.406198123962479
15	0	47	0.522620452409048
16	0	48	0.469509390187804
17	0	49	0.536920738414768
18	0	50	0.517770355407108
19	0	51	0.469129382587652
20	0	52	0.587261745234905
21	0	53	0.584431688633773
22	0	54	0.4544990899818
23	0	55	0.536040720814416
24	0	56	0.501980039600792
25	0	57	0.496119922398448
26	0	58	0.50035000700014
27	0	59	0.496169923398468
28	0	60	0.49529990599812
29	0	61	0.498559971199424
30	0	62	0.499219984399688
31	0	63	0.49979999599992
32	1	64	0.501190023800476

Wie in der Tabelle zu sehen ist, sind die Bits Nr.47 bis Nr.64 in etwa gleichverteilt. Weiterhin zeigen die Bits Nr. 41 bis 44 einige Veränderungen. Die Bits Nr.1 bis Nr.41 ändern sich frühestens alle

$$2^{64-41} * 4 \text{ ms} = 2^{23} * 4 \text{ ms} \approx 33,500 \text{ sec} \approx 9 \text{ h}$$

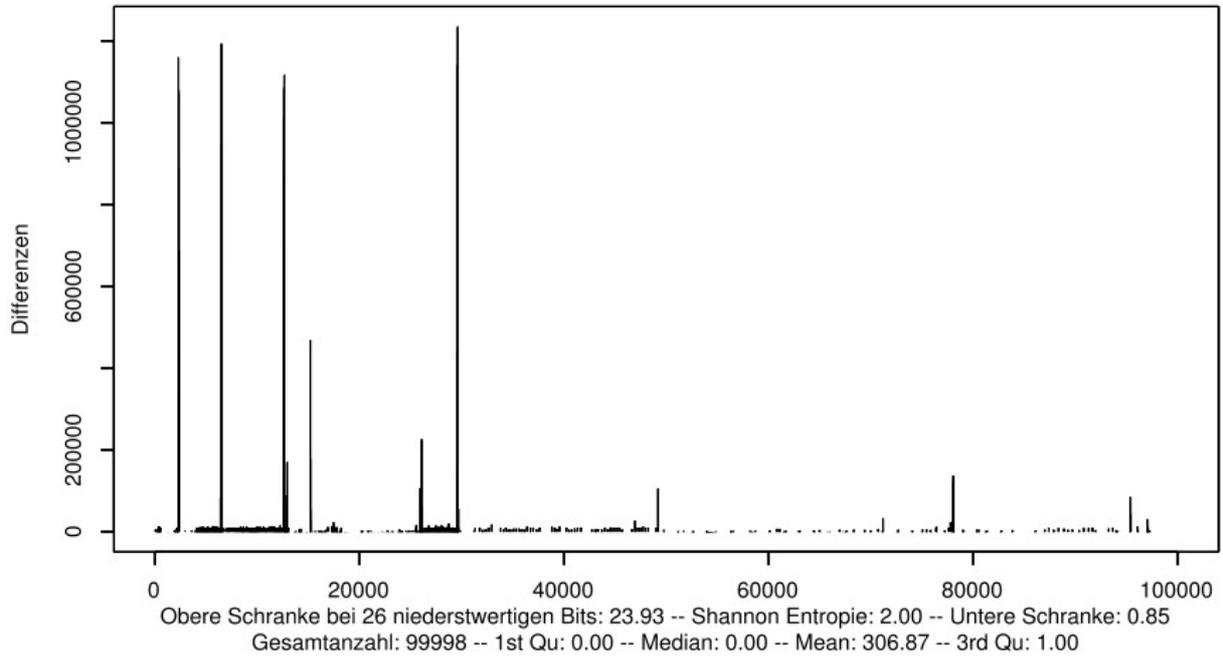
und bleiben deshalb im Betrachtungszeitraum konstant.

Der Testdurchlauf dauerte ca. 3 Stunden.

6.2.5.3.2 Differenzen der Jiffies

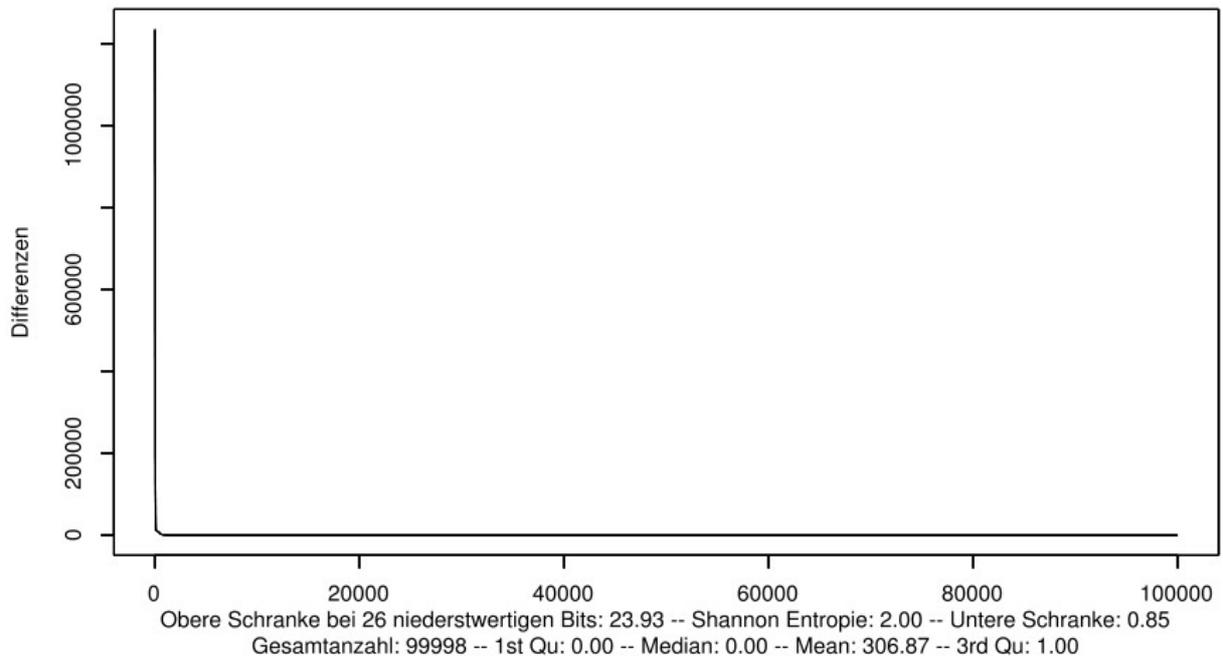
In einem weiteren Test werden die Differenzen der Jiffies bei direkt aufeinander folgenden Ereignissen untersucht. Der folgende Plot stellt diese Differenzen dar.

Verteilung der Differenzen der Jiffies (Disk) - 64 low bits



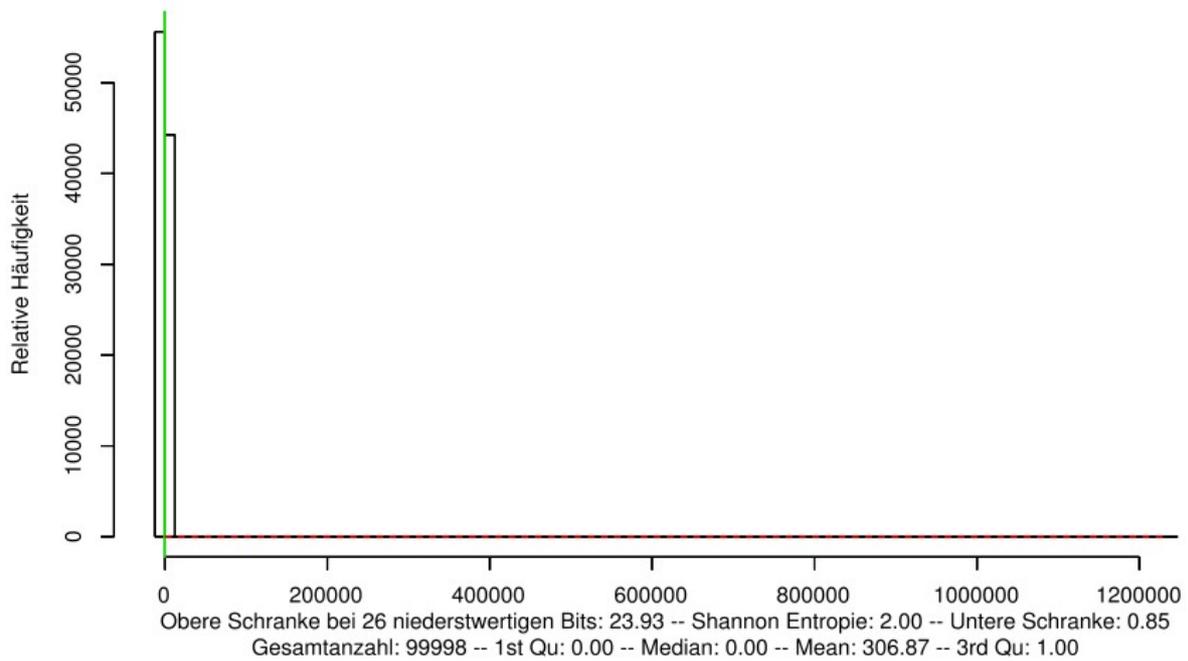
Zusätzlich sind nochmals die gleichen Werte wie in der vorangegangenen Abbildung aufgelistet.

Geordnete Verteilung der Differenzen der Jiffies (Disk) - 64 low bits



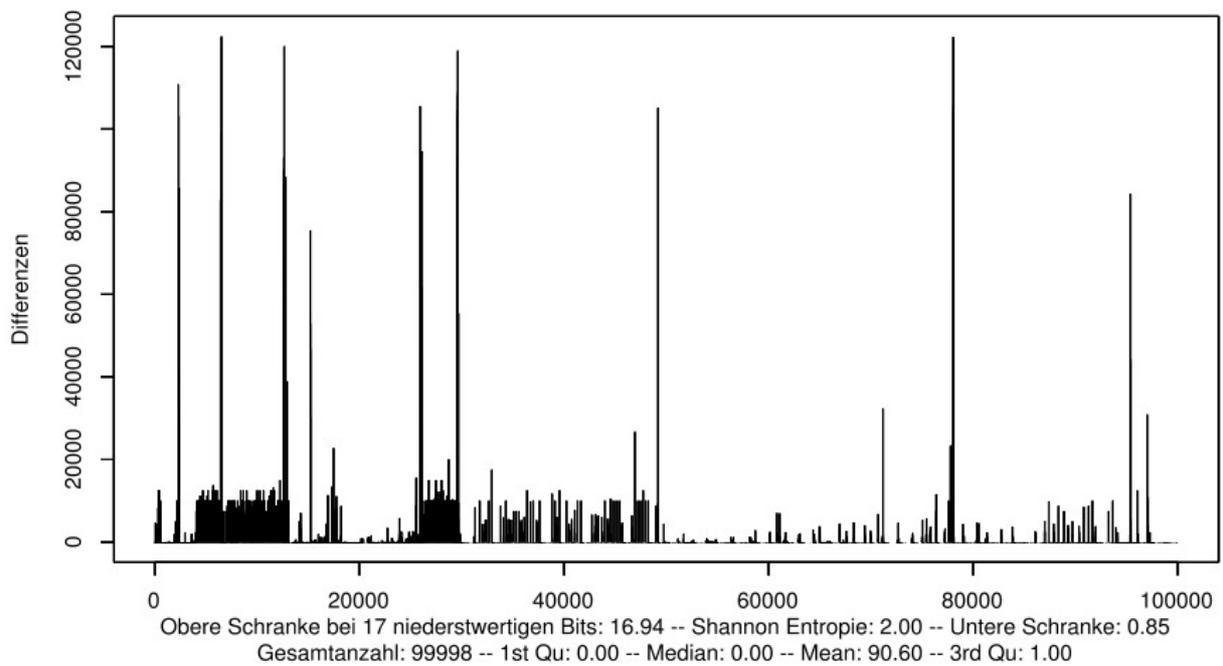
Die nächste Abbildung zeigt das Histogramm der empirischen Verteilung der Differenz von Jiffies. Wiederum sind die gleichen Werte wie in der vorangegangenen Abbildung aufgelistet.

Histogramm der Differenz der Jiffies (Disk) - 64 low bits

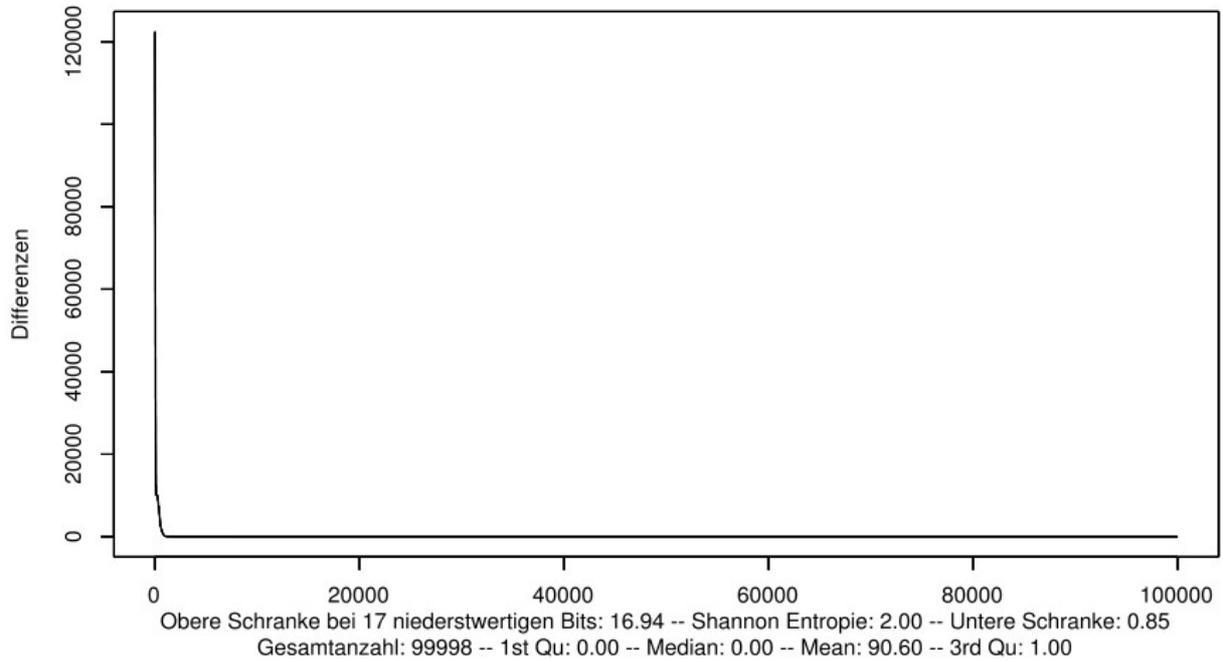


Die gleichen Graphen bezogen auf die 17 (64 - 47) niederwertigen Bits sind im Folgenden dargestellt.

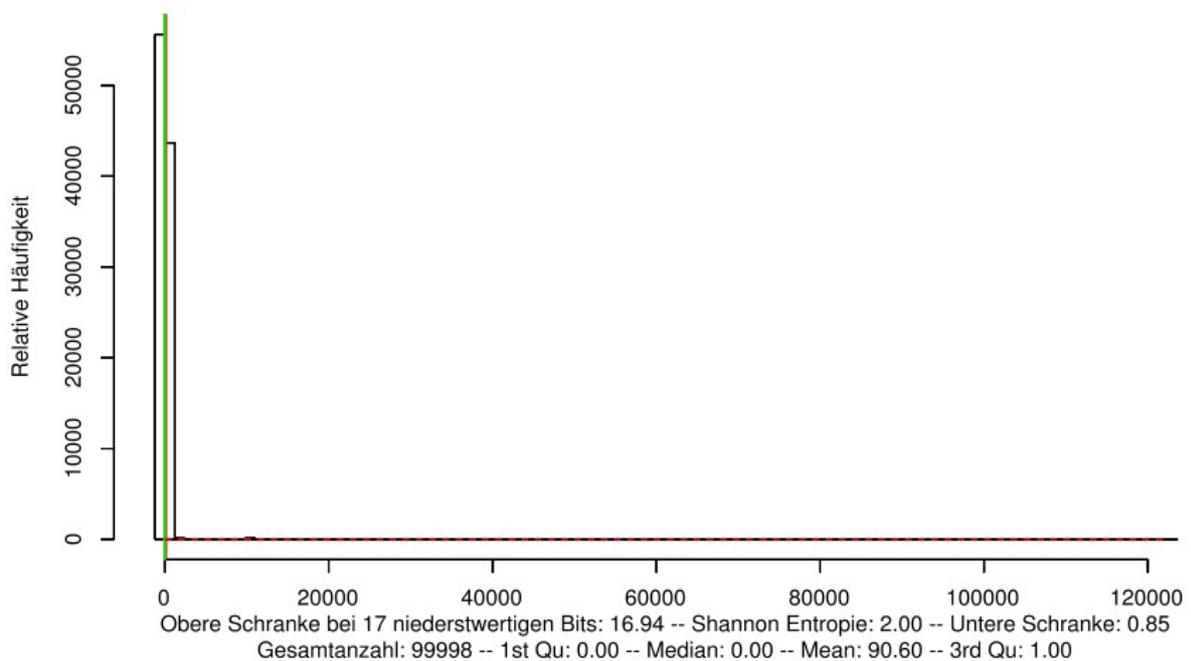
Verteilung der Differenzen der Jiffies (Disk) - 17 low bits



Geordnete Verteilung der Differenzen der Jiffies (Disk) - 17 low bits



Histogramm der Differenz der Jiffies (Disk) - 17 low bits



6.2.5.3.3 Interpretation der Ergebnisse

Die generelle Interpretation der Jiffies wird in Abschnitt 6.2.5.2.3 gegeben.

Wenn man die höherwertigen Bits ignoriert und sich auf die niederwertigen Bits konzentriert, können folgende Aussagen getroffen werden:

- Die Verteilung der Differenzen der Jiffies ist aufgrund der Art und Weise der Erzeugung der Messwerte sehr einseitig. Die Stichprobenanzahl muss erhöht werden und eine normale Nutzung des Systems muss erfolgen.
- Auf Basis der beobachteten Daten für die Jiffies können die folgenden theoretischen Werte berechnet werden:
 - **Obere Schranke der Entropie: 23,93 Bits**
 - **Shannon-Entropie: 2,0 Bits**
 - **Untere Schranke der Entropie: 0,85 Bits**

Die theoretischen Entropiewerte sind sehr ähnlich zu denen der beobachteten Jiffies für die Eingabegeräte.

Für die praktische Analyse der Entropiequelle, kann Folgendes gesagt werden: Die Anzahl der niederwertigen Bits, welche für die Entropie-Berechnungen verwendet wird, hängt von der Größe der Zeitvarianzen ab. Wenn man annimmt, dass diese Varianzen maximal eine Minute (der Kern führt mindestens einmal pro Minute eine Datensynchronisation mit der Festplatte durch) und minimal einige hundert Mikrosekunden sind, sind kaum noch Bits in der Variable der Jiffies vorhanden, die für eine Entropie-Berechnung herangezogen werden können. Demzufolge ist die erwartete Entropie auf Basis der Jiffies relativ gering. Diese Überlegungen sind konsistent mit den theoretischen Entropiewerten.

6.2.6 Untersuchung auf stochastische Unabhängigkeit

In diesem Abschnitt werden wir innerhalb von gemessenen Ereigniswerten stichprobenhaft Bitgruppen auf ihre stochastische Unabhängigkeit untersuchen. Da die vorangegangenen Tests gezeigt haben, dass die Prozessorzyklen einen Großteil der Entropie liefern, beschränken wir unsere Überprüfung darauf.

Den nachfolgenden Tests liegen die gleichen Daten wie in den Abschnitten 6.2.4.2 (Eingabegeräte) und 6.2.4.3 (Blockgeräte) zugrunde.

6.2.6.1 Erklärung der Tests

Für die Untersuchung auf stochastische Unabhängigkeit verschiedener Bitgruppen erstellen wir zunächst Kontingenztabelle und werten diese mit Hilfe des Chi-Quadrat-Tests aus. Zur Erstellung der Kontingenztabelle wählt man zuerst die beiden Bit-Gruppen aus, die auf stochastische Unabhängigkeit zu prüfen sind. Wir geben ein Beispiel für eine Kontingenztabelle zur Untersuchung der Bitgruppen (Bit 32, Bit 33) und (Bit 35, Bit 36, Bit 37) der Prozessorzyklen für Blockgeräte (siehe 6.2.4.3) an:

	000	001	010	011	100	101	110	111
00	2716	2803	2676	2627	2756	2633	2781	2660
01	3209	3111	3238	3428	3526	3086	3125	3376
10	3418	3334	3463	3476	3365	3572	3310	3198
11	3340	3240	3107	3072	2908	3002	3151	3146

Dabei sind in der linken Spalte alle möglichen Belegungen der Bits 32 und 33, den sogenannten Bedingungsbits, gegeben. In der ersten Zeile finden sich alle möglichen Belegungen für die Bits 35 bis 37, die Ratebits genannt werden. Die anderen Einträge geben an, wie häufig die entsprechende Kombination aufgetreten ist. Beispielsweise zeigt der Eintrag rechts unten, dass in 3146 Messungen sowohl Bit 32 und Bit 33 als auch die Bits 35, 36 und 37 gesetzt (d.h. mit 1 belegt) waren.

Wir erinnern daran, dass zwei Ereignisse A und B stochastisch unabhängig sind, wenn

$$P(A \cap B) = P(A) \cdot P(B)$$

gilt. Sind $P(A)$ und $P(B)$ größer als 0, so ist das gleichbedeutend mit

$$P(A|B) = \frac{P(A \cap B)}{P(B)} = P(A) \quad ,$$

wobei $P(A|B)$ die Wahrscheinlichkeit dafür ist, dass A eintritt unter der Voraussetzung, dass B bereits eingetreten ist (bedingte Wahrscheinlichkeit). Bezogen auf die obige Kontingenztabelle bedeutet das, dass die Bitgruppen (Bit 32, Bit 33) und (Bit 35, Bit 36, Bit 37) stochastisch unabhängig sind, wenn jede Belegung der Ratebits für eine feste Belegung der Bedingungsbits etwa gleich häufig vorkommt, d.h. die Zahlen in einer Zeile sind einigermaßen gleich groß²⁸.

Die letzte Aussage ist für Testzwecke natürlich zu vage. Als analytisches Werkzeug zur statistisch präzisen Prüfung, ob die Werte einer Kontingenztabelle gegen die stochastische Unabhängigkeit sprechen, wird hier der Chi-Quadrat-Test genutzt und der p-Wert berechnet, der die Wahrscheinlichkeit dafür angibt, dass man unter Voraussetzung der stochastischen Unabhängigkeit das gemessene oder ein „noch extremeres“ Messergebnis erhält. Ist der p-Wert ist sehr klein (beispielsweise kleiner als 0.01 oder 0.001), so geht man davon aus, dass keine stochastische Unabhängigkeit vorliegt.

6.2.6.2 Testdurchführung

Die Implementierung des oben beschriebenen Tests ist in der Datei independence.r im Verzeichnis independence zu finden. Für die Bestimmung des p-Wertes wurde die R-Funktion chisq.test verwendet. Wie bereits beschrieben, verwenden wir die Daten, die für die Tests in den Abschnitten 6.2.4.2 (Eingabegeräte) und 6.2.4.3 (Blockgeräte) gesammelt wurden.

Die Stichproben umfassen alle möglichen²⁹ Kombinationen der Parameter

- Startposition s aus {25, 33, ..., 56} – höhere Startpositionen können nicht gewählt werden, da das Maximum von Bedingungsbits, Ratebits und Verschiebung zusammen mit der Startposition den maximalen Bitwert von 64 umfassen
- Anzahl Bedingungsbits bb aus {1, 2, 3}
- Anzahl Ratebits rb aus {1, 2, 3}
- Verschiebung v aus {0, 1, 2}
- untere Schranke für den p-Wert (Signifikanzniveau): 0.01

die wie folgt erklärt sind:

Die Startposition s gibt die Bitposition des ersten Bedingungsbits an. Dieses und die folgenden bb-1 Bits bilden die Bedingungsbits. Darauf folgen v Bits, die nicht betrachtet werden. An diese schließen die rb Ratebits. Die folgende Tabelle

1.Bed.bit		bb.Bed.bit				1.Ratebit		rb.Ratebit
b_s	...	b_{s+bb-1}	b_{s+bb}	...	$b_{s+bb+v-1}$	b_{s+bb+v}	...	$b_{s+bb+v+rb-1}$

veranschaulicht die Bedeutung der einzelnen Parameter.

Bemerkungen:

- Die Prozessorzyklen auf einem Intel x86 System haben eine Auflösung von 1ns. Damit läuft ein 32 Bit Wert alle 4 Sekunden über. Obwohl die zugrunde gelegte Stichprobe ca. 3 Stunden umfasst, muss angenommen werden, dass je höher der Bitwertigkeit ist,

²⁸ Die Werte im Beispiel zeigen teils deutliche Schwankungen, sodass man hier Abhängigkeiten annimmt. Der nachfolgende Chi-Quadrat-Test bestärkt diese Vermutung.

²⁹ Möglich sind alle Kombinationen, in denen die Bitposition des letzten Ratebits nicht größer als 64 ist, da die Werte der Prozessorzyklen 64 Bit groß sind. Für die Startposition s=63 wird dementsprechend nur ein Test mit bb=1, rb=1 und v=0 durchgeführt, also eine Untersuchung der Bits 63 und 64 auf stochastische Abhängigkeit.

umso wahrscheinlicher sind Korrelationen zwischen den Beobachtungen. Dies liegt daran, dass die Stichprobe einen erheblich größeren Umfang hat, als Überläufe des 32 Bit Werts der Prozessorzyklen zu verzeichnen sind ($3h / 4s = 2700$ Überläufe). Damit haben viele Beobachtungen die gleichen, oder sehr nahe liegenden höherwertigen Bits.

- Die Bedeutung der obigen Parameter weicht in diesem Dokument von denen in [LLT07] ab. So folgen in [LLT07] die Ratebits direkt auf die Bedingungsbits, während hier durch den Parameter Bitverschiebung eine bestimmte Anzahl an Bits zwischen diesen Bitgruppen ausgelassen werden kann. In [LLT07] gibt es den Parameter für die Startposition nicht. Stattdessen wird Variabilität durch die Angabe einer Verschiebung vom niederwertigen Bit zum ersten Ratebit (in [LLT07] Bitverschiebung genannt) ermöglicht.

6.2.6.3 Testresultate und Interpretation für Eingabegeräte

Die folgende Tabelle listet alle Stichproben auf, für die ein p-Wert kleiner dem gewählten Signifikanzniveau gefunden wurde.

Die folgende Tabelle fasst alle weiteren Tests zusammen, in denen ein p-Wert kleiner als das Signifikanzniveau von 0.01 festgestellt wurde.

s	bb	rb	v	p-Wert	s	bb	rb	v	p-Wert
33	1	1	0	5.3569e-07	39	3	2	0	9.9952e-06
33	1	2	0	1.1251e-05	39	3	2	1	1.0922e-08
33	1	3	0	0.00025433	39	3	2	2	4.0282e-06
33	2	1	0	0.0076499	39	3	3	0	1.0749e-10
33	2	2	0	0.0047929	39	3	3	1	1.7241e-10
33	2	3	0	0.0081665	39	3	3	2	< 10 ⁻¹⁰
33	3	3	2	0.00067457	40	2	3	0	0.0093714
34	1	1	0	0.00072744	40	2	3	1	0.00046038
34	1	1	1	0.0021361	40	2	3	2	1.563e-07
34	1	2	0	9.3776e-05	40	3	1	2	0.0052171
34	1	3	0	0.00052654	40	3	2	0	0.00043576
34	2	1	0	0.0092104	40	3	2	1	9.8293e-06
34	2	3	2	0.00011983	40	3	2	2	3.4764e-05
34	3	2	2	1.712e-07	40	3	3	0	5.2161e-10
34	3	3	1	0.00011909	40	3	3	1	1.0701e-10
34	3	3	2	< 10 ⁻¹⁰	40	3	3	2	6.9365e-06
35	1	3	2	1.82e-05	41	1	3	1	0.0081109
35	2	2	2	4.9502e-06	41	1	3	2	3.6235e-05
35	2	3	1	0.00016557	41	2	2	0	0.00016478
35	2	3	2	4.7538e-09	41	2	2	1	0.0014347
35	3	1	2	< 10 ⁻¹⁰	41	2	2	2	9.382e-05
35	3	2	1	6.7707e-10	41	2	3	0	3.4706e-08
35	3	2	2	< 10 ⁻¹⁰	41	2	3	1	4.0512e-08
35	3	3	0	6.9099e-09	41	2	3	2	1.3845e-05
35	3	3	1	< 10 ⁻¹⁰	41	3	1	0	0.00098336

s	bb	rb	v	p-Wert	s	bb	rb	v	p-Wert
35	3	3	2	< 10 ⁻¹⁰	41	3	1	1	3.5102e-05
36	1	3	2	5.1178e-05	41	3	2	0	4.9872e-07
36	2	1	2	7.7009e-08	41	3	2	1	3.8489e-07
36	2	2	1	5.4954e-06	41	3	2	2	9.0284e-05
36	2	2	2	< 10 ⁻¹⁰	41	3	3	0	< 10 ⁻¹⁰
36	2	3	0	0.00013351	41	3	3	1	2.3517e-10
36	2	3	1	< 10 ⁻¹⁰	41	3	3	2	0.00075849
36	2	3	2	< 10 ⁻¹⁰	42	1	1	1	0.0037661
36	3	1	1	8.5728e-08	42	1	2	0	0.0021828
36	3	1	2	0.00020736	42	1	2	2	0.0092242
36	3	2	0	1.0202e-05	42	1	3	0	0.0020408
36	3	2	1	< 10 ⁻¹⁰	42	1	3	1	0.0049231
36	3	2	2	1.8119e-08	42	1	3	2	0.00053817
36	3	3	0	< 10 ⁻¹⁰	42	2	1	0	0.0013485
36	3	3	1	< 10 ⁻¹⁰	42	2	2	0	0.0032498
36	3	3	2	1.1203e-10	42	2	2	1	0.00078222
37	1	2	2	0.005884	42	2	2	2	5.8964e-05
37	1	3	2	0.0026255	42	2	3	0	0.00011107
37	2	1	2	0.0022288	42	2	3	1	7.3379e-08
37	2	2	1	0.0038246	42	2	3	2	5.0984e-05
37	2	2	2	2.2941e-08	42	3	2	0	0.0064947
37	2	3	1	< 10 ⁻¹⁰	42	3	2	1	7.0337e-07
37	2	3	2	4.1393e-08	42	3	3	0	6.889e-09
37	3	1	2	0.000164	42	3	3	1	5.3822e-07
37	3	2	1	2.2273e-06	43	1	2	2	0.0022262
37	3	2	2	< 10 ⁻¹⁰	43	1	3	1	0.00082405
37	3	3	0	1.1442e-07	43	2	2	1	0.0005255
37	3	3	1	< 10 ⁻¹⁰	43	2	3	0	0.0011176
37	3	3	2	< 10 ⁻¹⁰	43	2	3	1	0.0018264
38	1	1	1	0.0060986	43	3	2	0	0.00015054
38	1	2	2	4.9369e-05	43	3	3	0	0.00014775
38	1	3	1	2.3303e-05	44	1	2	1	0.0063847
38	1	3	2	1.4231e-05	44	1	3	1	0.0010211
38	2	1	2	1.9982e-05	44	2	2	0	0.0029862
38	2	2	1	0.00079943	44	2	3	0	0.0013092
38	2	2	2	< 10 ⁻¹⁰	44	3	1	0	0.0084038
38	2	3	0	0.0018565	44	3	2	0	0.0015533
38	2	3	1	< 10 ⁻¹⁰	44	3	3	0	0.0010268
38	2	3	2	< 10 ⁻¹⁰	48	1	2	1	0.0085427
38	3	1	1	7.5548e-05	48	1	3	0	0.0018785

s	bb	rb	v	p-Wert	s	bb	rb	v	p-Wert
38	3	1	2	1.1961e-08	48	2	1	1	0.0048198
38	3	2	0	0.0019647	48	2	2	0	0.0014741
38	3	2	1	< 10 ⁻¹⁰	48	3	1	0	0.0047607
38	3	2	2	6.7742e-10	53	2	1	2	0.0050445
38	3	3	0	< 10 ⁻¹⁰	53	2	3	2	0.005647
38	3	3	1	< 10 ⁻¹⁰	54	1	1	2	0.00037382
38	3	3	2	< 10 ⁻¹⁰	54	1	2	1	0.0046288
39	1	2	2	3.3775e-07	54	1	2	2	0.0031512
39	1	3	1	1.4239e-05	54	1	3	2	0.00047169
39	1	3	2	2.2592e-07	54	2	1	1	0.002953
39	2	1	2	2.0449e-07	54	2	2	0	0.0052694
39	2	2	1	5.2568e-06	54	2	2	1	0.009818
39	2	2	2	8.9657e-08	54	2	3	0	0.0068721
39	2	3	0	4.8113e-05	54	2	3	1	0.0034692
39	2	3	1	3.9823e-08	54	3	1	0	0.0019155
39	2	3	2	3.881e-08	54	3	2	0	0.0017988
39	3	1	1	1.833e-06	54	3	3	0	0.0019625
39	3	1	2	0.00039495	55	2	2	0	0.007325

Bei den Startpositionen 33 bis 42 wurden deutliche Hinweise gegen die stochastische Unabhängigkeit gefunden - wie bereits oben gemutmaßt. Diese Bits enthalten also nur wenig Entropie. Auch bei der Bitposition 43, 44, 48, 53 und 54 wurden derartige Hinweise festgestellt, wobei der Grund nicht erkennbar ist..

Somit scheint die stochastische Unabhängigkeit innerhalb der Bits 33 bis 64 nur bei etwa 17 Bits wahrscheinlich . Da der Entropieschätzer des LRNG jedes Ereignis ohnehin nur mit maximal 11 Bit bewertet, werden die nach diesen Tests vermuteten Abhängigkeiten nicht als problematisch eingestuft.

6.2.6.4 Testresultate und Interpretation für Blockgeräte

Die Ergebnisse und somit auch die Interpretation sind sehr ähnlich zu der im vorherigen Abschnitt 6.2.6.3. Die folgende Tabelle fasst alle weiteren Tests zusammen, in denen ein p-Wert kleiner als das Signifikanzniveau von 0.01 festgestellt wurde.

s	bb	rb	v	p-Wert	s	bb	rb	v	p-Wert
33	2	1	0	0.00039455	34	2	3	1	5.4457e-10
33	2	1	1	0.0015222	34	3	1	0	2.481e-10
33	2	2	0	< 10 ⁻¹⁰	34	3	1	1	4.0782e-06
33	2	2	1	2.6778e-05	34	3	1	2	0.00059117
33	2	2	2	1.4888e-06	34	3	2	0	< 10 ⁻¹⁰
33	2	3	0	< 10 ⁻¹⁰	34	3	2	1	8.5384e-09
33	2	3	1	< 10 ⁻¹⁰	34	3	3	0	< 10 ⁻¹⁰
33	2	3	2	2.6459e-10	34	3	3	1	2.5846e-06
33	3	1	0	1.1342e-08	35	1	1	1	0.0079421

33	3	1	1	1.7262e-05	35	1	2	0	0.0085917
33	3	1	2	3.215e-06	35	1	3	0	0.00051827
33	3	2	0	< 10^-10	35	2	1	0	0.008973
33	3	2	1	< 10^-10	35	2	1	1	0.0029254
33	3	2	2	2.0133e-06	35	2	2	0	0.0003817
33	3	3	0	< 10^-10	35	2	2	1	0.0010669
33	3	3	1	< 10^-10	35	2	3	0	0.00034299
33	3	3	2	3.5954e-05	35	2	3	1	0.0036014
34	1	1	0	0.00011898	35	3	1	0	0.0060753
34	1	2	0	< 10^-10	35	3	2	0	0.0034334
34	1	2	1	1.0715e-05	43	3	1	2	0.0099746
34	1	2	2	0.0019773	45	2	1	1	0.0093561
34	1	3	0	< 10^-10	52	2	1	1	0.0086581
34	1	3	1	< 10^-10	52	2	3	0	0.0069466
34	1	3	2	2.0992e-06	52	3	2	0	0.0043966
34	2	1	0	1.7888e-08	53	1	1	1	0.0034053
34	2	1	1	2.3537e-06	53	1	2	1	0.001312
34	2	1	2	0.0069882	53	1	3	0	0.0028582
34	2	2	0	< 10^-10	53	1	3	1	0.0085374
34	2	2	1	1.2934e-08	53	2	2	0	0.0030943
34	2	3	0	< 10^-10	53	3	1	0	0.0059158

Bei Startpositionen im Bereich von Bit 33 bis Bit 35 wurden Hinweise gegen die stochastische Unabhängigkeit gefunden - wie erwartet. Weiterhin wurden bei einigen niederwertigen Bits Auffälligkeiten entdeckt - der Grund hierfür ist nicht bekannt.

Hier bleibt festzuhalten, dass die stochastische Unabhängigkeit innerhalb der Bits 36 bis 64 mit wenigen Ausnahmen wahrscheinlich ist. Da der Entropieschätzer des LRNG jedes Ereignis ohnehin nur mit maximal 11 Bit bewertet, werden die nach diesen Tests vermuteten Abhängigkeiten nicht als problematisch eingestuft.

6.2.7 Vergleich theoretischer Entropie mit LRNG Abschätzungen

Wir haben ferner Testdaten gesammelt, die vom LINUX Zufallszahlengenerator mit dem Entropiewert 0, mit dem Entropiewert 1, und mit dem Entropiewert 8 bewertet wurden. Demzufolge wird der Test mehrfach durchgeführt, wobei die Tests den gleichen, im Folgenden beschriebenen Ansatz nutzen.

Es wird die vom LRNG berechnete Entropie für ein Ereignis mit dem gewünschten Entropiewert von 0 (entsprechend auch für 1 und 8) verglichen. Wenn das Ereignis einen anderen Entropiewert aufweist, wird es verworfen. Ansonsten wird der Ereigniswert aufgezeichnet.

Eine Besonderheit bei den Interrupt-Ereignissen ist zu nennen: da sich die gesamte Analyse mit den LRNG Entropie-Pools `input_pool`, `blocking_pool` und `nonblocking_pool` beschäftigt, werden die Eingaben in diese Pools analysiert. Bei den Interrupt-Ereignissen wird der `input_pool` nicht mehr direkt von den Hardware-Ereignissen gespeist (wie bei Eingabegeräten oder Blockgeräten), sondern aus den Werten des `fast_pools`. Dies berücksichtigt der vorliegende Test und nimmt pro Einmischen der Daten aus dem `fast_pool` alle 4 32 Bit Wörter des `fast_pool` und interpretiert diese als Ereigniswerte.

Wie bereits an anderer Stelle ersichtlich, hat der `fast_pool` bereits eine große Bandbreite an möglichen Werten und damit eine hohe Shannon-Entropie. Dies bedeutet, dass die Shannon-Entropie bei den Ereigniswerten, welche der LRNG mit einer heuristischen Entropie von 1 interpretiert eine signifikant höhere Shannon-Entropie als bei den anderen Entropiewerten zu erwarten ist – nur bei der heuristischen Entropie von 1 werden die Interrupt-Ereignisse aufgezeichnet.

Die Analyse der Entropie erfolgte analog zu Abschnitt 6.2.3, indem die Häufigkeiten der einzelnen Bits der Ereigniswerte berechnet werden. Weiterhin werden die theoretischen Entropiemaße

- untere Entropieschranke (bzw. minimale Entropie) und
- die Shannon-Entropie

aus den Ereigniswerten berechnet.

Das Ziel des Tests ist aufzuzeigen, dass die theoretischen Entropiemaße immer größer oder gleich der vom LRNG berechneten Entropie ist. Damit würde die Entropieberechnung des LRNG als konservativ angesehen werden.

Für diesen Test stammen alle Hardware-Ereignisse von allen Entropiesammelfunktionen.

Die Stichprobenanzahl bezieht sich immer auf die Anzahl der Ereignisse mit der gewünschten Entropie.

6.2.7.1 Testdurchführung

Ausgehend von diesen Anfangsüberlegungen wurde folgender Test für den Entropiewert von 0 erstellt:

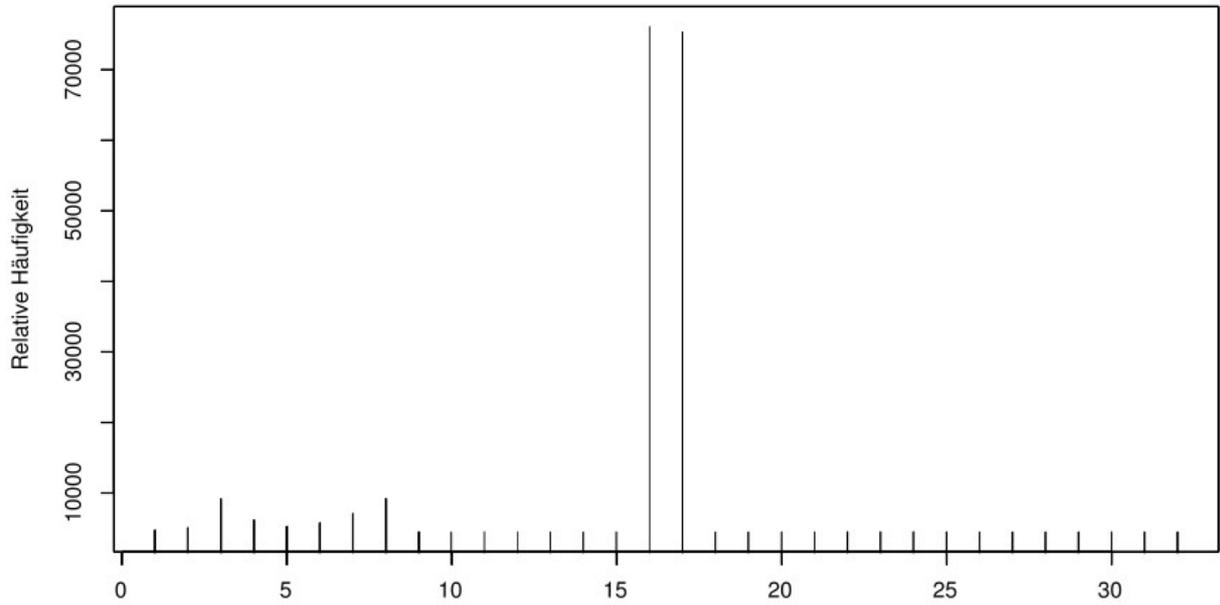
- Das SystemTap-Skript `compare_rng_shannon.stp` wurde für die gewünschte Entropie 0, 1 und 8 erstellt. Das Shell Skript `gendata.sh` enthält die Stichprobenanzahl mit der Variable `num_samplings`. Dieses Skript erzeugen eine Liste mit den beobachteten Ereigniswerten und einer Aufsummierung des Auftretens der einzelnen Bits der Ereigniswerte.
- Das SystemTap-Skript wird mit dem Bash-Skript `gendata.sh` initialisiert.
- Das R-Project-Programm `compare_rng_shannon.r` erstellen die folgenden Graphiken und Tabellen aus den erzeugten Datenreihen.

Die gesetzte Stichprobenanzahl ist $S=100.000$ für die Ereignisse für die der LRNG 0 Bit und 1 Bit Entropie annimmt. Hingegen ist die Stichprobengröße für die Ereignisse, für die der LRNG 8 Bit Entropie annimmt, $S=10.000$ aufgrund der Seltenheit solcher Ereignisse.

6.2.7.1.1 LRNG Entropieschätzung von 0 Bit pro Ereignis

Ausgehend von dieser Stichprobenanzahl wurde auf dem Testsystem eine Datenreihe mit Ereigniswerten erzeugt. Die Bitwerte der einzelnen Ereigniswerte werden in folgendem Histogramm aufgezeigt.

Verteilung der Bitwerte der Ereignisse mit 0 Bits



Shannon Entropie: 1.859594 -- Min Entropie 0.497038 -- Max Wert: 8388864 -- Max Anzahl: 70856 -- Gesamt 100000

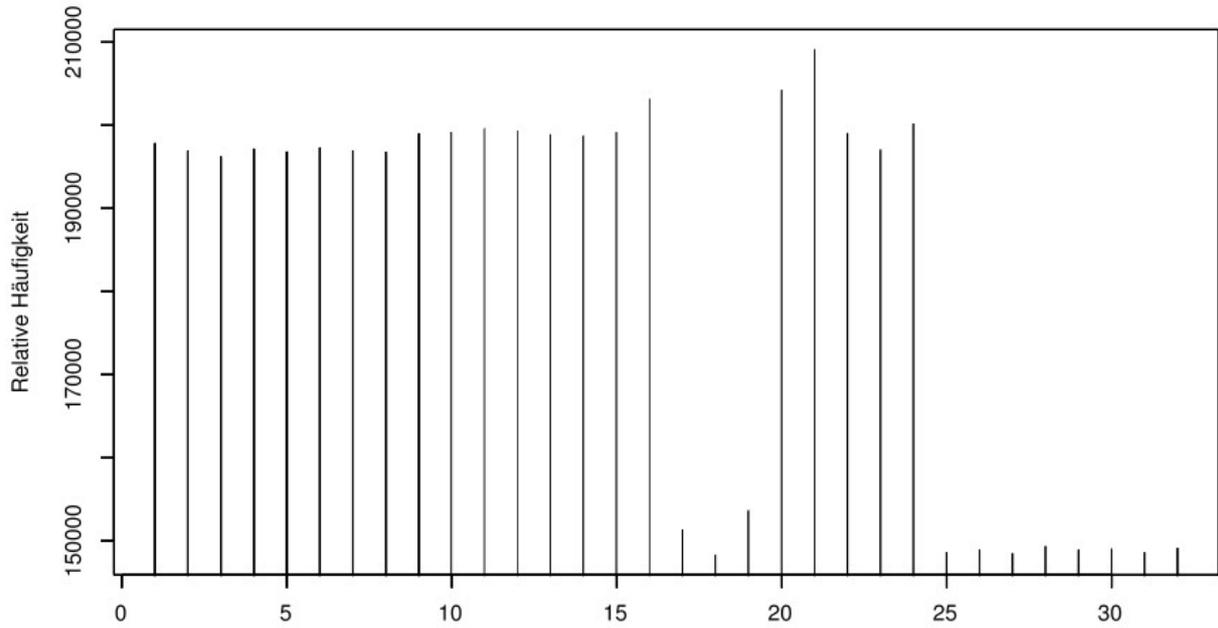
Des Weiteren sind in dem Histogramm folgende Werte bezogen auf die Ereigniswerte dargestellt:

- Ereigniswert, welcher am häufigsten auftritt und damit die untere Entropieschranke bestimmt
- Anzahl des am häufigsten auftretenden Ereigniswerts

6.2.7.1.2 LRNG Entropieschätzung von 1 Bit pro Ereignis

Ausgehend von dieser Stichprobenanzahl wurde auf dem Testsystem eine Datenreihe mit Ereigniswerten erzeugt. Die Bitwerte der einzelnen Ereigniswerte werden in folgendem Histogramm aufgezeigt.

Verteilung der Bitwerte der Ereignisse mit 1 Bits



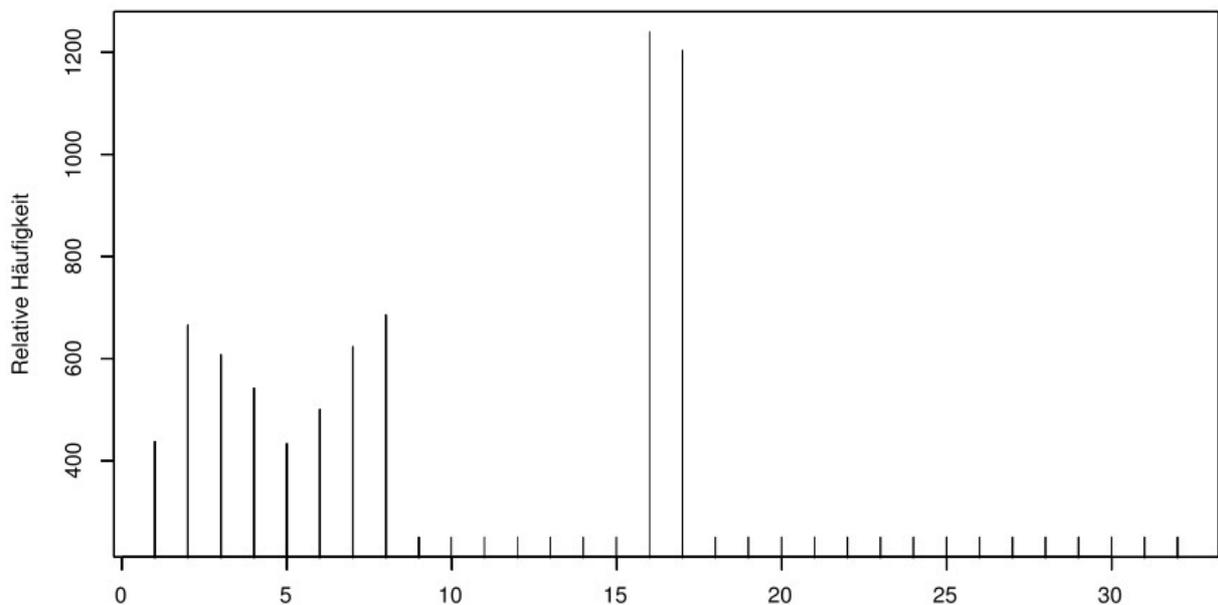
Shannon Entropie: 18.352631 -- Min Entropie 7.321928 -- Max Wert: 8388864 -- Max Anzahl: 2500 -- Gesamt 400000

Wieder ist mit „Max Wert“ der am häufigsten gemessene Wert und mit „Max Anzahl“ die Anzahl für das Auftreten von „Max Wert“ angegeben.

6.2.7.1.3 LRNG Entropieschätzung von 8 Bit pro Ereignis

Ausgehend von der genannten Stichprobenanzahl wurde auf dem Testsystem eine Datenreihe mit Ereigniswerten erzeugt. Die Bitwerte der einzelnen Ereigniswerte werden in folgendem Histogramm aufgezeigt.

Verteilung der Bitwerte der Ereignisse mit 8 Bits



Shannon Entropie: 3.394727 -- Min Entropie 1.113841 -- Max Wert: 8388864 -- Max Anzahl: 950 -- Gesamt 2056

Wieder ist mit „Max Wert“ der am häufigsten gemessene Wert und mit „Max Anzahl“ die Anzahl für das Auftreten von „Max Wert“ angegeben.

6.2.7.2 Interpretation der Ergebnisse

Die theoretische Entropie der Werte, welche im LRNG mit einer Entropie von Null bewertet werden, ist größer als Null. **Demzufolge ist der LRNG bei Ereignissen welche mit Null Bit Entropie bewertet werden, konservativer als eine theoretische Berechnung.**

Wie erwartet zeigen die theoretischen Entropiewerte der Ereignisse, welche im LRNG mit einer Entropie von eins bewertet werden, hohe Werte für die Shannon-Entropie und die minimale Entropie. **Demzufolge ist der LRNG bei Ereignissen welche mit ein Bit Entropie bewertet werden, konservativer als eine theoretische Berechnung.**

Hingegen ist die theoretische Entropie der Werte, welche der LRNG mit einer Entropie von 8 Bit bewertet, kleiner. **Demzufolge liegt bei Ereignissen welche mit 8 Bit Entropie bewertet werden, eine optimistischere Abschätzung des LRNG vor als eine theoretische Berechnung zulässt.**

7 Weitere Tests

7.1 Verwendung der Zufallsdaten

Neben der Analyse der Sammlung von Entropie und der Verarbeitung der Entropiedaten, ist natürlich auch die Verwendung der aus dem LRNG generierten Zufallszahlen interessant. In Abschnitt 6.1.2 wurde bereits auf den folgenden Test verwiesen. Er hat sich durch aufkommende Fragen zu dem unerwarteten Entropie-Verlauf im genannten Abschnitt angeboten.

Die Nutzer des LRNGs sind folgende:

- Kerninterne Dienste, die mit Hilfe der Funktion `get_random_bytes` Zufallsdaten aus dem LRNG extrahieren. Wie beschrieben nutzt diese Funktion den `nonblocking_pool`.
- User-Space welcher den LRNG mittels `/dev/random` ausliest. Diese Gerätedatei stellt den Zugang zum `blocking_pool` bereit.
- Weiterhin kann der User-Space auch den LRNG mittels `/dev/urandom` auslesen, wobei der `nonblocking_pool` verwendet wird.

7.1.1 Testdurchführung

Ausgehend von diesen Anfangsüberlegungen wurden folgende Tests für die Hardware-Klassen für Eingabegeräte und Blockgeräteaktivität definiert:

- Das SystemTap-Skript „`requested_entropy.stp`“ zeichnet die aufrufenden Funktionen und die angefragten Bytes auf. Die Stichprobengröße bezogen auf die Zeitdauer in Sekunden wird in der Probe `timer.s` in dem SystemTap-Skript festgelegt.
- Die SystemTap-Skripte werden mit dem Bash-Skript `gendata.sh` initialisiert. Dieses Skript installiert die notwendigen Programme, um das SystemTap-Skript beim Startvorgang aufzurufen und damit den Startvorgang zu beobachten. Weiterhin kann das SystemTap-Skript zur Laufzeit mittels dem Skript `/sbin/stap/requested_entropy.sh` aufgerufen werden. Die generierten Daten müssen aus dem Verzeichnis `$COLLECTDIR`, welches im Skript `requested_entropy.sh` definiert ist, übertragen werden.
- Das Perl-Programm `process_result.pl` erstellt die folgende Tabelle.

7.1.2 Testresultate im laufenden System

Wie zuvor wurden $S = 600$ Messungen im Abstand von jeweils 1 Sekunde durchgeführt.

Die folgende Tabelle listet die Nutzer der Zufallszahlen auf. Dabei können folgende Informationen aus der Tabelle gelesen werden:

- Die linke Spalte definiert das Subsystem, aus dem der Aufrufer stammt. Alles, was nicht als „User-Space“ markiert ist, sind kerninterne Subsysteme.
- Die mittlere Spalte listet die Kernfunktion oder die Gerätedatei, welche Anfragen an den LRNG stellt auf.
- Die rechte Spalte listet die gesamte Anzahl angefragter Bytes auf. Falls eine Funktion mehrere Anfragen stellt, werden die Bytes aller Anfragen aufsummiert.

Diese Tabelle basiert auf den Daten in der Datei `requested_entropy-1.txt`.

Subsystem	Anfragende Funktion	Angeforderte Bytes
Network	<code>tun_set_iff</code>	1
Network	<code>rt_cache_invalidate</code>	14

Subsystem	Anfragende Funktion	Angeforderte Bytes
Network	inet_frag_secret_rebuild	3
Network	__ipv6_regen_rndid	1
UserSpace	/dev/urandom	36
Misc	reqsk_queue_alloc	4
VFS	ecryptfs_write_headers_virt	108
VFS	ecryptfs_new_file_context	108
VFS	create_elf_tables	135

7.1.3 Interpretation der Ergebnisse

Zu diesem Test gibt es keine Interpretation, da er nur die Funktionen und Mechanismen auflistet, welche Entropie abgefragt haben.

7.2 Größe des Seeds

Die Diskussion des Designs des LRNG zeigt, dass beim Startvorgang ein Seed in die Entropie-Pools eingebracht werden sollte, um die initialen Stände der Pools zu durchmischen.

Es stellt sich nun die Frage, welche Größe dieser Seed haben sollte, wobei klar ist, dass es dafür keine obere Schranke gibt. Je größer der Seed ist, umso größer ist auch die Entropie der Pools. Im ungünstigsten Fall, indem ein Angreifer einfach bekannte Daten zu den Pools hinzufügt, erhöht sich die Entropie nicht - sie wird aber auch nicht verringert.

Demnach muss die minimale Größe des Seeds diskutiert werden. Als Vorbetrachtung erinnern wir uns an die in Abschnitt 2.5.1.8 besprochene Implementierung der Schreibfunktionen für die Gerätedateien /dev/random und /dev/urandom. Diese Schreibfunktion fügt die erhaltenen Daten in den nonblocking_pool und den blocking_pool ein. Der input_pool wird nicht verändert.

Bei dem Hinzufügen von Daten in die Pools wird, bedingt durch die byteweise Verarbeitung von Eingabedaten, ein Eingabe-Byte auf 4 Bytes vergrößert (siehe Abschnitt 2.5.2, Schritt 1). Diese Vergrößerung wird einfach durch ein Cast von char * auf __u32 gewährleistet. Bei diesem Cast werden 24 Null-Bits zu dem Byte hinzugefügt. Des Weiteren wird der so vergrößerte Puffer um einen Wert nach links verschoben.

Es sollte also sichergestellt werden, dass alle Bits des Pools durch das Einmischen des Seeds einer potentiellen Veränderung unterworfen werden, d.h. dass der Bit-Rotation alle $32 * 32$ Bits unterworfen werden..

Das Verhalten der Rotation kann anhand des folgenden kleinen Programms illustriert werden:

```

#include <stdio.h>

main()
{
    // Start rotation analysis at some point
    int input_rotate = 0;
    // Well-crafted size of the input
    int nbytes = 1024;
    // pointer into the entropy pool
    int i = 0;
    // word mask from the entropy pool
    int wordmask = 32 - 1;

    // taken from mix_pool_bytes_extract
    while(nbytes--)
    {
        printf("%d ", (input_rotate & 31));

        i = (i - 1) & wordmask;
        input_rotate += i ? 7 : 14;

        if(!(nbytes % 32))
            printf("\n");
    }
}

```

Quellcode 35: Analyse der Rotation der Eingabedaten

Dieser Code kopiert die Verarbeitung der input_rotate Variable aus der LRNG-Funktion `_mix_pool_bytes`.

Mittels dieses Programms erhält man folgende Matrix:

0	7	14	21	28	3	10	17	24	31	6	13	20	27	2	9	16	23	30	5	12	19	26	1	8	15	22	29	4	11	18	25
7	14	21	28	3	10	17	24	31	6	13	20	27	2	9	16	23	30	5	12	19	26	1	8	15	22	29	4	11	18	25	0
14	21	28	3	10	17	24	31	6	13	20	27	2	9	16	23	30	5	12	19	26	1	8	15	22	29	4	11	18	25	0	7
21	28	3	10	17	24	31	6	13	20	27	2	9	16	23	30	5	12	19	26	1	8	15	22	29	4	11	18	25	0	7	14
28	3	10	17	24	31	6	13	20	27	2	9	16	23	30	5	12	19	26	1	8	15	22	29	4	11	18	25	0	7	14	21
3	10	17	24	31	6	13	20	27	2	9	16	23	30	5	12	19	26	1	8	15	22	29	4	11	18	25	0	7	14	21	28
10	17	24	31	6	13	20	27	2	9	16	23	30	5	12	19	26	1	8	15	22	29	4	11	18	25	0	7	14	21	28	3
17	24	31	6	13	20	27	2	9	16	23	30	5	12	19	26	1	8	15	22	29	4	11	18	25	0	7	14	21	28	3	10
24	31	6	13	20	27	2	9	16	23	30	5	12	19	26	1	8	15	22	29	4	11	18	25	0	7	14	21	28	3	10	17
31	6	13	20	27	2	9	16	23	30	5	12	19	26	1	8	15	22	29	4	11	18	25	0	7	14	21	28	3	10	17	24
6	13	20	27	2	9	16	23	30	5	12	19	26	1	8	15	22	29	4	11	18	25	0	7	14	21	28	3	10	17	24	31
13	20	27	2	9	16	23	30	5	12	19	26	1	8	15	22	29	4	11	18	25	0	7	14	21	28	3	10	17	24	31	6
20	27	2	9	16	23	30	5	12	19	26	1	8	15	22	29	4	11	18	25	0	7	14	21	28	3	10	17	24	31	6	13
27	2	9	16	23	30	5	12	19	26	1	8	15	22	29	4	11	18	25	0	7	14	21	28	3	10	17	24	31	6	13	20
2	9	16	23	30	5	12	19	26	1	8	15	22	29	4	11	18	25	0	7	14	21	28	3	10	17	24	31	6	13	20	27
9	16	23	30	5	12	19	26	1	8	15	22	29	4	11	18	25	0	7	14	21	28	3	10	17	24	31	6	13	20	27	2
16	23	30	5	12	19	26	1	8	15	22	29	4	11	18	25	0	7	14	21	28	3	10	17	24	31	6	13	20	27	2	9
23	30	5	12	19	26	1	8	15	22	29	4	11	18	25	0	7	14	21	28	3	10	17	24	31	6	13	20	27	2	9	16
30	5	12	19	26	1	8	15	22	29	4	11	18	25	0	7	14	21	28	3	10	17	24	31	6	13	20	27	2	9	16	23

5	12	19	26	1	8	15	22	29	4	11	18	25	0	7	14	21	28	3	10	17	24	31	6	13	20	27	2	9	16	23	30
12	19	26	1	8	15	22	29	4	11	18	25	0	7	14	21	28	3	10	17	24	31	6	13	20	27	2	9	16	23	30	5
19	26	1	8	15	22	29	4	11	18	25	0	7	14	21	28	3	10	17	24	31	6	13	20	27	2	9	16	23	30	5	12
26	1	8	15	22	29	4	11	18	25	0	7	14	21	28	3	10	17	24	31	6	13	20	27	2	9	16	23	30	5	12	19
1	8	15	22	29	4	11	18	25	0	7	14	21	28	3	10	17	24	31	6	13	20	27	2	9	16	23	30	5	12	19	26
8	15	22	29	4	11	18	25	0	7	14	21	28	3	10	17	24	31	6	13	20	27	2	9	16	23	30	5	12	19	26	1
15	22	29	4	11	18	25	0	7	14	21	28	3	10	17	24	31	6	13	20	27	2	9	16	23	30	5	12	19	26	1	8
22	29	4	11	18	25	0	7	14	21	28	3	10	17	24	31	6	13	20	27	2	9	16	23	30	5	12	19	26	1	8	15
29	4	11	18	25	0	7	14	21	28	3	10	17	24	31	6	13	20	27	2	9	16	23	30	5	12	19	26	1	8	15	22
4	11	18	25	0	7	14	21	28	3	10	17	24	31	6	13	20	27	2	9	16	23	30	5	12	19	26	1	8	15	22	29
11	18	25	0	7	14	21	28	3	10	17	24	31	6	13	20	27	2	9	16	23	30	5	12	19	26	1	8	15	22	29	4
18	25	0	7	14	21	28	3	10	17	24	31	6	13	20	27	2	9	16	23	30	5	12	19	26	1	8	15	22	29	4	11
25	0	7	14	21	28	3	10	17	24	31	6	13	20	27	2	9	16	23	30	5	12	19	26	1	8	15	22	29	4	11	18

Eine Zeile dieser Matrix enthält genau 32 Einträge entsprechend der Verarbeitung von 32 Eingabe-Bytes. D.h. mit einer Zeile werden alle 32 Wörter der Pools verändert. Schaut man sich aber die Veränderung des Rotationswerts an, so sieht man, dass sich bei jedem Erreichen von 32 der Rotationswert um eins nach links schiebt. Der Grund hierfür ist folgender Code:

```
input_rotate += i ? 7 : 14;
```

Anhand der Matrix kann man sehr gut sehen, dass die Werte nach $32 * 32$ Rotierungen alle möglichen Bitwerte spaltenweise und zeilenweise angenommen haben. D.h. erst wenn

$$32 * 32 = 1024 \text{ Bytes}$$

verarbeitet wurden, ist definitiv sichergestellt, dass für alle Bits des blocking_pool und nonblocking_pool die theoretische Möglichkeit bestand, verändert zu werden.

Demzufolge ist die untere Schranke für den Seed 1024 Bytes.

7.3 Test Procedure A

Die folgenden Tests stammen aus der [AIS2031] Test Procedure A.

Um den Test auf eine ausreichend große Stichprobe anzuwenden, wurden Zufallszahlen ausschließlich aus /dev/urandom verarbeitet. Designbedingt sind /dev/random und /dev/urandom mit der Ausnahme der Entropieschätzung/anforderung identisch. Auf Basis der Gemeinsamkeiten können folgende Aussagen getroffen werden:

- /dev/urandom enthält weniger oder maximal gleich viel Entropie wie /dev/random.
- Die Messungen mit Hilfe der BSI Test Suite A prüfen die Verteilung der Ausgabedaten. Diese Verteilung wird alleine von dem verwendeten SHA-1 Hash bestimmt. Die Entropie hat keinen Einfluss auf die Tests.

7.3.1 Testdurchführung

Als Stichprobengröße wird $S=100\text{MByte}$ verwendet.

Die Zufallszahlen aus dem LRNG wurden mittels folgendem Kommando erzeugt:

```
dd if=/dev/urandom of=testsuiteA-urandom.data bs=1024 count=100000
```

Die erzeugte Datei mit den Zufallszahlen wird in den folgenden Tests verwendet.

Des Weiteren wurde das Perl-Programm ntg.1.5/test_proc_A.pl erstellt, das die in [AIS2031], Test Procedure A, definierten Tests implementiert. Dieses Perl-Programm wurde mit den generierten Zufallszahlen wie folgt aufgerufen:

```
perl ./test_proc_A.pl testsuiteA-urandom.data
```

Es extrahiert die ersten 20.000 Bits aus der Datei mit den Zufallszahlen. Diese Stichprobe ist die Grundlage für die in Test Procedure A definierten Tests.

7.3.2 Testresultate

Der Aufruf des Perl Programms erzeugte folgende Resultate:

```
Test T0 passed
Test T1 passed
Test T2 passed
Test T3 passed
Test T4 passed
Test T5 passed
```

Damit sind alle Tests aus der Test Procedure A erfüllt worden.

7.4 BSI Test Suite A

Zur Unterstützung der NTG.1.5 Analyse wurde die BSI Test Suite A verwendet, um die Ausgabe des LRNG zu prüfen.

Die Analyse wurde ebenfalls, wie in Abschnitt 7.3 beschrieben, nur für /dev/urandom durchgeführt und ist damit eine untere Abschätzung für /dev/random.

7.4.1 Testdurchführung

Als Stichprobengröße wird $S=100\text{ MByte}$ verwendet.

Die Zufallszahlen aus dem LRNG wurden mittels folgendem Kommando erzeugt:

```
dd if=/dev/urandom of=testsuiteA-urandom.data bs=1024 count=100000
```

Die erzeugte Datei mit den Zufallszahlen wird mittels dem BSI-Analyseprogramm verarbeitet.

7.4.2 Testresultate

Das BSI-Analyseprogramm hat keine Auffälligkeiten feststellen können. Damit erfüllen die Zufallszahlen die vom BSI-Analyseprogramm gestellten Qualitätsanforderungen.

Die genauen Testresultate sind in der Datei ntg.1.5/AIS 31.log zu finden.

7.5 LRNG Ausgabe und „dieharder“-Test

Zur Unterstützung der NTG.1.5-Analyse wurde die „dieharder“³⁰ Test Suite verwendet, um die Ausgabe des LRNG zu prüfen.

Die Analyse wurde ebenfalls, wie in Abschnitt 7.3 beschrieben, nur für /dev/urandom durchgeführt und ist damit eine untere Abschätzung für /dev/random.

7.5.1 Testdurchführung

Der Test wurde mittels folgendem Kommando durchgeführt: ntg.1.5/dieharder-urandom.sh.

7.5.2 Testresultate

Folgende Resultate (die auch in der Datei ntg.1.5/dieharder-urandom-result.txt zu finden sind) sind von dem Test erzeugt wurden:

```
#####
#           dieharder version 3.31.1 Copyright 2003 Robert G. Brown           #
#####
#   rng_name   |rands/second|   Seed   |
stdin_input_raw|  1.49e+06  |1086582434|
#####
#           test_name   |ntup| tsamples |psamples|   p-value |Assessment
#####
```

30 Siehe <http://www.phy.duke.edu/~rgb/General/dieharder.php>.

diehard_birthdays	0	100	100 0.31536555	PASSED
diehard_operm5	0	1000000	100 0.67725614	PASSED
diehard_rank_32x32	0	40000	100 0.69686688	PASSED
diehard_rank_6x8	0	100000	100 0.04824161	PASSED
diehard_bitstream	0	2097152	100 0.49047865	PASSED
diehard_opso	0	2097152	100 0.71011245	PASSED
diehard_oqso	0	2097152	100 0.54516007	PASSED
diehard_dna	0	2097152	100 0.84201170	PASSED
diehard_count_ls_str	0	256000	100 0.97082212	PASSED
diehard_count_ls_byt	0	256000	100 0.01596907	PASSED
diehard_parking_lot	0	12000	100 0.75842967	PASSED
diehard_2dsphere	2	8000	100 0.87060588	PASSED
diehard_3dsphere	3	4000	100 0.47992921	PASSED
diehard_squeeze	0	100000	100 0.44846381	PASSED
diehard_sums	0	100	100 0.11426153	PASSED
diehard_runs	0	100000	100 0.39331276	PASSED
diehard_runs	0	100000	100 0.62035847	PASSED
diehard_craps	0	200000	100 0.71670127	PASSED
diehard_craps	0	200000	100 0.49466306	PASSED
marsaglia_tsang_gcd	0	10000000	100 0.64294019	PASSED
marsaglia_tsang_gcd	0	10000000	100 0.78617153	PASSED
sts_monobit	1	100000	100 0.36187950	PASSED
sts_runs	2	100000	100 0.03106116	PASSED
sts_serial	1	100000	100 0.13669643	PASSED
sts_serial	2	100000	100 0.55496441	PASSED
sts_serial	3	100000	100 0.82997651	PASSED
sts_serial	3	100000	100 0.81887292	PASSED
sts_serial	4	100000	100 0.53442840	PASSED
sts_serial	4	100000	100 0.72290465	PASSED
sts_serial	5	100000	100 0.52142057	PASSED
sts_serial	5	100000	100 0.12917803	PASSED
sts_serial	6	100000	100 0.23589163	PASSED
sts_serial	6	100000	100 0.03031741	PASSED
sts_serial	7	100000	100 0.62901409	PASSED
sts_serial	7	100000	100 0.86621233	PASSED
sts_serial	8	100000	100 0.49603287	PASSED
sts_serial	8	100000	100 0.08757404	PASSED
sts_serial	9	100000	100 0.11066353	PASSED
sts_serial	9	100000	100 0.28073129	PASSED
sts_serial	10	100000	100 0.28190901	PASSED
sts_serial	10	100000	100 0.95473365	PASSED
sts_serial	11	100000	100 0.89068288	PASSED
sts_serial	11	100000	100 0.57411409	PASSED
sts_serial	12	100000	100 0.89548510	PASSED
sts_serial	12	100000	100 0.22948460	PASSED
sts_serial	13	100000	100 0.94840423	PASSED
sts_serial	13	100000	100 0.71772291	PASSED
sts_serial	14	100000	100 0.11055728	PASSED
sts_serial	14	100000	100 0.08125634	PASSED
sts_serial	15	100000	100 0.57779319	PASSED
sts_serial	15	100000	100 0.89835181	PASSED
sts_serial	16	100000	100 0.86220685	PASSED
sts_serial	16	100000	100 0.29394587	PASSED
rgb_bitdist	1	100000	100 0.03898038	PASSED
rgb_bitdist	2	100000	100 0.99019346	PASSED
rgb_bitdist	3	100000	100 0.13567320	PASSED
rgb_bitdist	4	100000	100 0.13766308	PASSED

```

    rgb_bitdist| 5| 100000| 100|0.95776240| PASSED
    rgb_bitdist| 6| 100000| 100|0.75690297| PASSED
    rgb_bitdist| 7| 100000| 100|0.82007452| PASSED
    rgb_bitdist| 8| 100000| 100|0.63679309| PASSED
    rgb_bitdist| 9| 100000| 100|0.90348914| PASSED
    rgb_bitdist| 10| 100000| 100|0.27844131| PASSED
    rgb_bitdist| 11| 100000| 100|0.42111074| PASSED
    rgb_bitdist| 12| 100000| 100|0.56348314| PASSED
rgb_minimum_distance| 2| 10000| 1000|0.63788529| PASSED
rgb_minimum_distance| 3| 10000| 1000|0.55704944| PASSED
rgb_minimum_distance| 4| 10000| 1000|0.76344526| PASSED
rgb_minimum_distance| 5| 10000| 1000|0.02953872| PASSED
    rgb_permutations| 2| 100000| 100|0.42197498| PASSED
    rgb_permutations| 3| 100000| 100|0.99951875| WEAK
    rgb_permutations| 4| 100000| 100|0.36724237| PASSED
    rgb_permutations| 5| 100000| 100|0.78048531| PASSED
    rgb_lagged_sum| 0| 1000000| 100|0.86434351| PASSED
    rgb_lagged_sum| 1| 1000000| 100|0.13086685| PASSED
    rgb_lagged_sum| 2| 1000000| 100|0.13857371| PASSED
    rgb_lagged_sum| 3| 1000000| 100|0.07721072| PASSED
    rgb_lagged_sum| 4| 1000000| 100|0.58458552| PASSED
    rgb_lagged_sum| 5| 1000000| 100|0.27657593| PASSED
    rgb_lagged_sum| 6| 1000000| 100|0.40551988| PASSED
    rgb_lagged_sum| 7| 1000000| 100|0.86343493| PASSED
    rgb_lagged_sum| 8| 1000000| 100|0.38415504| PASSED
    rgb_lagged_sum| 9| 1000000| 100|0.94738402| PASSED
    rgb_lagged_sum| 10| 1000000| 100|0.14885456| PASSED
    rgb_lagged_sum| 11| 1000000| 100|0.95748401| PASSED
    rgb_lagged_sum| 12| 1000000| 100|0.32835960| PASSED
    rgb_lagged_sum| 13| 1000000| 100|0.01438556| PASSED
    rgb_lagged_sum| 14| 1000000| 100|0.96801886| PASSED
    rgb_lagged_sum| 15| 1000000| 100|0.66719287| PASSED
    rgb_lagged_sum| 16| 1000000| 100|0.46693451| PASSED
    rgb_lagged_sum| 17| 1000000| 100|0.20911935| PASSED
    rgb_lagged_sum| 18| 1000000| 100|0.92160709| PASSED
    rgb_lagged_sum| 19| 1000000| 100|0.95330337| PASSED
    rgb_lagged_sum| 20| 1000000| 100|0.57750745| PASSED
    rgb_lagged_sum| 21| 1000000| 100|0.65592667| PASSED
    rgb_lagged_sum| 22| 1000000| 100|0.43698596| PASSED
    rgb_lagged_sum| 23| 1000000| 100|0.42320983| PASSED
    rgb_lagged_sum| 24| 1000000| 100|0.90257721| PASSED
    rgb_lagged_sum| 25| 1000000| 100|0.84688128| PASSED
    rgb_lagged_sum| 26| 1000000| 100|0.29661338| PASSED
    rgb_lagged_sum| 27| 1000000| 100|0.53637839| PASSED
    rgb_lagged_sum| 28| 1000000| 100|0.91801343| PASSED
    rgb_lagged_sum| 29| 1000000| 100|0.52916862| PASSED
    rgb_lagged_sum| 30| 1000000| 100|0.86248388| PASSED
    rgb_lagged_sum| 31| 1000000| 100|0.70107788| PASSED
    rgb_lagged_sum| 32| 1000000| 100|0.46127037| PASSED
    rgb_kstest_test| 0| 10000| 1000|0.31637582| PASSED
    dab_bytedistrib| 0| 51200000| 1|0.62077482| PASSED
    dab_dct| 256| 50000| 1|0.15827817| PASSED
Preparing to run test 207. ntuple = 0
    dab_filltree| 32| 15000000| 1|0.89664680| PASSED
    dab_filltree| 32| 15000000| 1|0.76718916| PASSED
Preparing to run test 208. ntuple = 0
    dab_filltree2| 0| 5000000| 1|0.06799260| PASSED

```

```
dab_filltree2| 1| 5000000| 1|0.93894852| PASSED
Preparing to run test 209. ntuple = 0
dab_monobit2| 12| 65000000| 1|0.25284199| PASSED
```

Alle Tests sind positiv ausgefallen, d.h. es wurden keine Hinweise gegen die Qualität der Zufallszahlen gefunden,

Dennoch ist ein Testresultat als „weak“ markiert. Dieses schwache Resultat ist bei Testwiederholungen aufgetaucht. Die Aussage der Tests, die schwache Resultate lieferten, ist nicht abschließend geklärt.

7.6 Test des Referenzsystems

Entsprechend den Vorgaben des BSIs wurden zusätzliche Tests auf folgendem Referenzsystem durchgeführt:

- Hersteller Dell
- Modell Optiplex 780
- Prozessor Intel Core 2 Duo
- Taktfrequenz 3000 MHz
- Arbeitsspeicher 4096 MB
- Festplatte (HDD) 160 GB
- Optisches - Laufwerk DVD-RW
- Chipsatz Intel Q45 Express
- Modell (Grafik) GMA 4500
- Anschlüsse Displayport / VGA
- Prozessortyp E8400
- Technologie Dual-Core
- Socket Socket 775
- Front-Side-Bus 1333 MHz
- Taktfrequenz 3000 MHz
- L2 Cache 6144KB
- Besonderheiten CPU 64bit-fähig, Intel VT-x/d
- Arbeitsspeicher 4096 MB
- max. Speichergröße 8192 MB
- Speichertaktung 1066 MHz
- Speicherspezifikation DDR3-SDRAM
- Speicherbänke 4
- HDD - Schnittstelle SATA
- HDD - Formfaktor 8,9cm / 3.5 Zoll
- Optisches - Laufwerk DVD-RW

Das System wurde mit Ubuntu Linux betrieben, welches in der folgenden Liste spezifiziert ist:

- 32 Bit Version von Kubuntu 12.10 (Ubuntu mit KDE Desktop)
- Ubuntu Kernel 3.5
- Desktop Installation mit LibreOffice, Firefox, KMail

Während der Durchführung der Tests wurden ein normaler Büroalltag simuliert, indem auf dem PC die folgenden Aktionen durchgeführt wurden:

- 0,5h mittels rekonq (Standardbrowser Kubuntu) Internet-Recherche durchgeführt
- 0,3h E-Mail Adresse in KMail einrichten und E-Mail aus einem Webformular an diese schicken. E-Mail wurde empfangen, gelesen und bearbeitet.
- 0,3h Route auf Google Maps planen und günstigste Strecke in LibreOffice Calc berechnen.
- 0,5h Abtippen eines Zeitungsartikels in OpenOffice Writer und anschließend auf der Festplatte speichern. Mit KCalc ein paar Werte des Artikels nach-berechnen.
- 0,5h Videos von Youtube mit Firefox betrachtet
- 0,5h Mails im KMail lesen, schreiben.
- 0,1h Abtippen eines Zeitungsartikels in LibreOffice Writer und anschließend auf der Festplatte speichern.
- 1,0h Surfen mit Firefox (u.a. Social Networks)
- 1,0h Abtippen eines Zeitungsartikels in LibreOffice Writer und anschließend auf der Festplatte speichern.
- 1,5h Pause (PC unbesetzt)

The folgenden Unterabschnitte beinhalten die Beschreibung und die Testresultate der durchgeführten Tests.

7.6.1 Test entsprechend Abschnitt 7.3

Auf dem Referenzsystem wurde eine Stichprobe $S=200\text{ MByte}$ gewählt.

Der Aufruf des Perl Programms erzeugte folgende Resultate:

```
Test T0 passed
Test T1 passed
Test T2 passed
Test T3 passed
Test T4 passed
Test T5 passed
```

Damit sind alle Tests aus der Test Procedure A erfüllt worden.

7.6.2 Test entsprechend Abschnitt 7.4

Auf dem Referenzsystem wurde eine Stichprobe $S=200\text{ MByte}$ gewählt.

Die genauen Testresultate sind in der Datei reference_system/AIS 31.log zu finden.

7.6.3 Test entsprechend Abschnitt 7.5

Das Referenzsystem zeigte folgende Resultate:

```
#=====#
#          dieharder version 3.31.1 Copyright 2003 Robert G. Brown          #
#=====#
  rng_name   |rands/second|   Seed   |
stdin_input_raw| 2.59e+06 | 274628091|
#=====#
  test_name  |ntup| tsamples |psamples|  p-value |Assessment
#=====#
  diehard_birthdays|  0|    100|    100|0.41171179| PASSED
  diehard_operm5|  0| 1000000|    100|0.49149457| PASSED
  diehard_rank_32x32|  0|   40000|    100|0.80357258| PASSED
  diehard_rank_6x8|  0|  100000|    100|0.94991420| PASSED
  diehard_bitstream|  0| 2097152|    100|0.13106722| PASSED
```

diehard_opso	0	2097152	100 0.57420686	PASSED
diehard_oqso	0	2097152	100 0.96891939	PASSED
diehard_dna	0	2097152	100 0.09019472	PASSED
diehard_count_ls_str	0	256000	100 0.67751697	PASSED
diehard_count_ls_byt	0	256000	100 0.17507871	PASSED
diehard_parking_lot	0	12000	100 0.31899023	PASSED
diehard_2dsphere	2	8000	100 0.67842038	PASSED
diehard_3dsphere	3	4000	100 0.20818993	PASSED
diehard_squeeze	0	100000	100 0.73862645	PASSED
diehard_sums	0	100	100 0.43270809	PASSED
diehard_runs	0	100000	100 0.74701871	PASSED
diehard_runs	0	100000	100 0.84619408	PASSED
diehard_craps	0	200000	100 0.52566774	PASSED
diehard_craps	0	200000	100 0.96754452	PASSED
marsaglia_tsang_gcd	0	10000000	100 0.16097194	PASSED
marsaglia_tsang_gcd	0	10000000	100 0.11097988	PASSED
sts_monobit	1	100000	100 0.33979824	PASSED
sts_runs	2	100000	100 0.67239521	PASSED
sts_serial	1	100000	100 0.97239261	PASSED
sts_serial	2	100000	100 0.65336889	PASSED
sts_serial	3	100000	100 0.85283530	PASSED
sts_serial	3	100000	100 0.07645680	PASSED
sts_serial	4	100000	100 0.84630512	PASSED
sts_serial	4	100000	100 0.11841305	PASSED
sts_serial	5	100000	100 0.68829028	PASSED
sts_serial	5	100000	100 0.69453446	PASSED
sts_serial	6	100000	100 0.93238140	PASSED
sts_serial	6	100000	100 0.56928225	PASSED
sts_serial	7	100000	100 0.16831358	PASSED
sts_serial	7	100000	100 0.23848256	PASSED
sts_serial	8	100000	100 0.23085872	PASSED
sts_serial	8	100000	100 0.52827515	PASSED
sts_serial	9	100000	100 0.24113164	PASSED
sts_serial	9	100000	100 0.47464511	PASSED
sts_serial	10	100000	100 0.31281933	PASSED
sts_serial	10	100000	100 0.43507879	PASSED
sts_serial	11	100000	100 0.16246896	PASSED
sts_serial	11	100000	100 0.43790007	PASSED
sts_serial	12	100000	100 0.12820566	PASSED
sts_serial	12	100000	100 0.94178175	PASSED
sts_serial	13	100000	100 0.00729859	PASSED
sts_serial	13	100000	100 0.40317689	PASSED
sts_serial	14	100000	100 0.04106450	PASSED
sts_serial	14	100000	100 0.98180431	PASSED
sts_serial	15	100000	100 0.96062894	PASSED
sts_serial	15	100000	100 0.43722624	PASSED
sts_serial	16	100000	100 0.47379827	PASSED
sts_serial	16	100000	100 0.54448799	PASSED
rgb_bitdist	1	100000	100 0.96102556	PASSED
rgb_bitdist	2	100000	100 0.29808276	PASSED
rgb_bitdist	3	100000	100 0.62253926	PASSED
rgb_bitdist	4	100000	100 0.29133215	PASSED
rgb_bitdist	5	100000	100 0.71247694	PASSED
rgb_bitdist	6	100000	100 0.71386568	PASSED
rgb_bitdist	7	100000	100 0.04669202	PASSED
rgb_bitdist	8	100000	100 0.16674610	PASSED
rgb_bitdist	9	100000	100 0.13097007	PASSED

```

    rgb_bitdist| 10| 100000| 100|0.62659937| PASSED
    rgb_bitdist| 11| 100000| 100|0.24884007| PASSED
    rgb_bitdist| 12| 100000| 100|0.66720326| PASSED
rgb_minimum_distance| 2| 10000| 1000|0.26536325| PASSED
rgb_minimum_distance| 3| 10000| 1000|0.52497826| PASSED
rgb_minimum_distance| 4| 10000| 1000|0.09477451| PASSED
rgb_minimum_distance| 5| 10000| 1000|0.89953346| PASSED
    rgb_permutations| 2| 100000| 100|0.83974995| PASSED
    rgb_permutations| 3| 100000| 100|0.59791278| PASSED
    rgb_permutations| 4| 100000| 100|0.32743510| PASSED
    rgb_permutations| 5| 100000| 100|0.43420079| PASSED
    rgb_lagged_sum| 0| 1000000| 100|0.93356574| PASSED
    rgb_lagged_sum| 1| 1000000| 100|0.29750623| PASSED
    rgb_lagged_sum| 2| 1000000| 100|0.71921232| PASSED
    rgb_lagged_sum| 3| 1000000| 100|0.15983057| PASSED
    rgb_lagged_sum| 4| 1000000| 100|0.18568327| PASSED
    rgb_lagged_sum| 5| 1000000| 100|0.93199815| PASSED
    rgb_lagged_sum| 6| 1000000| 100|0.22197795| PASSED
    rgb_lagged_sum| 7| 1000000| 100|0.97772426| PASSED
    rgb_lagged_sum| 8| 1000000| 100|0.26137247| PASSED
    rgb_lagged_sum| 9| 1000000| 100|0.77798276| PASSED
    rgb_lagged_sum| 10| 1000000| 100|0.61172347| PASSED
    rgb_lagged_sum| 11| 1000000| 100|0.82865252| PASSED
    rgb_lagged_sum| 12| 1000000| 100|0.81911633| PASSED
    rgb_lagged_sum| 13| 1000000| 100|0.84575709| PASSED
    rgb_lagged_sum| 14| 1000000| 100|0.57885648| PASSED
    rgb_lagged_sum| 15| 1000000| 100|0.88869867| PASSED
    rgb_lagged_sum| 16| 1000000| 100|0.96680764| PASSED
    rgb_lagged_sum| 17| 1000000| 100|0.91155531| PASSED
    rgb_lagged_sum| 18| 1000000| 100|0.81421404| PASSED
    rgb_lagged_sum| 19| 1000000| 100|0.98135817| PASSED
    rgb_lagged_sum| 20| 1000000| 100|0.90092963| PASSED
    rgb_lagged_sum| 21| 1000000| 100|0.93823822| PASSED
    rgb_lagged_sum| 22| 1000000| 100|0.61468678| PASSED
    rgb_lagged_sum| 23| 1000000| 100|0.00210512| WEAK
    rgb_lagged_sum| 24| 1000000| 100|0.87016705| PASSED
    rgb_lagged_sum| 25| 1000000| 100|0.42547854| PASSED
    rgb_lagged_sum| 26| 1000000| 100|0.79471011| PASSED
    rgb_lagged_sum| 27| 1000000| 100|0.68958296| PASSED
    rgb_lagged_sum| 28| 1000000| 100|0.86268600| PASSED
    rgb_lagged_sum| 29| 1000000| 100|0.01538072| PASSED
    rgb_lagged_sum| 30| 1000000| 100|0.32842785| PASSED
    rgb_lagged_sum| 31| 1000000| 100|0.80329559| PASSED
    rgb_lagged_sum| 32| 1000000| 100|0.75082432| PASSED
    rgb_kstest_test| 0| 10000| 1000|0.47865632| PASSED
    dab_bytedistrib| 0| 51200000| 1|0.44791670| PASSED
    dab_dct| 256| 50000| 1|0.71113256| PASSED
Preparing to run test 207. ntuple = 0
    dab_filltree| 32| 15000000| 1|0.76420660| PASSED
    dab_filltree| 32| 15000000| 1|0.16485233| PASSED
Preparing to run test 208. ntuple = 0
    dab_filltree2| 0| 5000000| 1|0.72012991| PASSED
    dab_filltree2| 1| 5000000| 1|0.17161987| PASSED
Preparing to run test 209. ntuple = 0
    dab_monobit2| 12| 65000000| 1|0.98400083| PASSED

```

Die Testergebnisse sind konsistent mit denen aus Abschnitt 7.5.2.

8 Zusammenfassung und Ausblick

8.1 Zusammenfassung

Wir fassen die Ergebnisse dieses Berichts kurz zusammen:

- Der Linux-Kern stellt mit `/dev/random` und `/dev/urandom` zwei Gerätedateien für die Extraktion von Zufallszahlen zur Verfügung. Die Analyse in Kapitel 4 hat gezeigt, dass `/dev/random` alle Anforderungen von NTG.1 erfüllt, während `/dev/urandom` alle Anforderungen von NTG.1 bis auf NTG.1.1, also insbesondere auch DRG.3 erfüllt.
- Die Entropie-Pools wurden in Kapitel 5 einer umfangreichen Analyse unterzogen. Die durchgeführten Test weisen im Allgemeinen auf eine ausreichend gleichmäßige Verteilung von Daten innerhalb des Pools sowie eine ausreichend gleichmäßige Einmischung neuer Entropie hin.
- Als problematisch kann die geringe und zudem vorhersagbare Entropie beim Systemstart, insbesondere bei Headless-Systemen gesehen werden (siehe dazu Kapitel 3 und die Abschnitte 5.2.5, 5.3.5, 5.4.6). Ein Speichern des aktuellen Zustands der Entropie-Pools und (Re-)Seeding beim Booten sollte dringend erfolgen. Für Live-Systeme empfiehlt es sich deshalb, sicherheitskritische Anwendungen erst eine gewisse Zeit nach Systemstart durchzuführen, sodass genügend Entropie gesammelt werden konnte.
- Die Größe des Input-Pools ist nach Abschnitt 6.1 für die gesammelte Entropie ausreichend.
- Ein interessanter Aspekt ist in Abschnitt 6.1.2 aufgefallen: Bei jedem Starten eines Prozesses werden für Address-Space-Layout-Randomization (ASLR) und als Seed für interne DRNG Zufallszahlen vom LRNG angefordert, obwohl für ASLR schwächere Zufallszahlen ausreichen sollten und für weitere Zwecke eine On-Demand-Lösung deutlich ressourcensparender wäre.
- Nach Abschnitt 6.2.1 liefern die Ereigniswerte der verschiedenen Hardware-Quellen kaum Entropie. Theoretisch kann dies untermauert werden, indem man sich die geringe Anzahl von Ausprägungen vor Augen führt. Pro Blockgerät gibt es einen Wert, für HID gibt es einen Wert pro Tastendruck/Taste Loslassen/Bewegungsrichtung der Maus.
- Die Analyse in 6.2.4 hat jedoch gezeigt, dass die Prozessorzyklen pro Ereigniswert einen Großteil der Entropie liefern, da die Auflösung des Zeitgebers sehr hoch ist. Demzufolge können die Differenzen der Prozessorzyklen eine große Bandbreite einnehmen, welches den Grad der Ungewissheit, und damit die Entropie vergrößert.
- In 6.2.5 wurde der Entropiegewinn durch Jiffies betrachtet. Als Ergebnis lässt sich festhalten, dass der Jiffie-Wert eines Ereignisses etwas Entropie liefert. Da die Größenordnung der Differenzen der Jiffie-Werte aufgrund der groben Auflösung relativ gering ist, ist damit die gemessene und theoretische Entropie ebenfalls gering.
- In Abschnitt 6.2.6 wurden stichprobenhaft Ereigniswerte (Prozessorzyklen bei Eingabegeräten und Blockgeräten) auf stochastische Unabhängigkeit geprüft. Dabei wurden zwar einige mögliche Abhängigkeiten entdeckt, die jedoch aufgrund der oberen Schranke für die Entropiebewertung einzelner Ereignisse (siehe 2.5.3) für unproblematisch gehalten werden.
- Nach der Untersuchung in 6.2.7 kann gesagt werden, dass für die meisten Ereignisse ist die Entropieschätzung des LRNG als sehr konservativ zu bezeichnen ist. Dies ist eine grundsätzlich wünschenswerte Eigenschaft. Eine ausführliche Bemerkung zur Verfahrensweise bei der Änderung des Entropiezählers ist im nachfolgenden Abschnitt zu finden.
- Die Frage, ob 100 Bits Entropie direkt nach dem Starten verfügbar sind lässt sich relativ schnell beantworten: wenn `/dev/random` betrachtet wird, enthält jedes generierte Bit (fast) ein Bit an Entropie, wenn der Seed in `/dev/random` eingemischt

wurde. Dieser Seed stellt sicher, dass die gleichförmigen Bootvorgänge mit Entropie des vorherigen Systems vermengt werden. Diese Eigenschaft, dass jedes Ausgabebit in etwa ein Bit an Entropie besitzt, wird durch das Blockieren des Lesens auf Basis der Entropieschätzung sichergestellt.

Gesamturteil: Die Untersuchungen lassen insgesamt auf eine sehr hohe Qualität des Linux-Zufallszahlengenerators schließen. Aufgrund der Ergebnisse der Tests zum Systemstart gilt jedoch die Empfehlung, dass der LRNG für sicherheitskritische Anwendungen erst nach dem Seeding verwendet wird.

8.2 Ausblick und offene Fragen

- Die RDRAND-Instruktion neuerer Intel x86 Prozessoren wurde in diesem Dokument nicht untersucht. Die Autoren empfehlen, den LRNG nicht vollständig durch den Hardware-RNG zu ersetzen, sondern diesen nur als zusätzliche Entropie-Quelle zu nutzen und bei der Nutzung auf das Erhöhen der geschätzten Entropie zu verzichten. Auf diese Weise sind eventuell auftretende Sicherheitslücken sowie ein zeitliches Abbauen der Qualität des Hardware-RNG nicht als kritisch anzusehen. Die Änderungen im Kern 3.6 verknüpfen bei der Ausgabe von Zufallszahlen die von `/dev/random` oder `/dev/urandom` erzeugten Daten mit einer gleichen Anzahl von Daten von RDRAND mittels XOR.
- Das Einbinden der Interrupts als weitere, wenn auch sehr beschränkte Rauschquelle ab Version 3.6 ist grundsätzlich als positiv zu betrachten. Dabei sind jedoch folgende Dinge zu diskutieren: Sollte ein Angreifer von Beginn des Systemstarts an den Rechner mit Netzwerkpaketen fluten, wäre es möglich, dass er größtenteils Kontrolle über den Inhalt des `input_pool` bekommt? Um die Antwort vorweg zu nehmen: Die Wahrscheinlichkeit für diesen Effekt ist kaum gegeben, da die Zeitauflösung der Messungen im Nanosekundenbereich liegt. Diese Messgenauigkeit ist um mehrere Potenzen größer, als die Genauigkeit eines Angreifers aufgrund der Netzwerklatenzen (Router, Switches, Netzwerkstacks, etc.). Aus den obigen Ausführungen geht hervor, dass genau der Unterschied zwischen den Messgenauigkeiten eines Angreifers und des LRNG die Entropie bezogen wird.
- Die zusätzliche Integration weiterer Entropie-Quellen, wie etwa Lüfterdrehzahlen, wäre ebenfalls wünschenswert. Auf Android-Systemen (welche einen Linux-Kern verwenden) wird heute schon der LRNG mit den Ereignissen von vorhandenen Gyroskopen, Umgebungslichtsensoren, GPS Sensoren, etc. gefüttert.
- Die Verfahrensweise bei der Änderung des Entropiezählers ist aus Sicht der Autoren zumindest als extrem konservativ einzuschätzen: Werden beispielsweise 80 Bit von `/dev/random` angefordert, so wird der entsprechende Entropiezähler um eben diesen Wert erniedrigt. Aufgrund des Designs der Extraktionsfunktion (zweimalige Anwendung von SHA-1) scheint die Kenntnis der Ausgabe-Bits jedoch keinerlei Rückschlüsse auf den Inhalt des Pools oder nachfolgende/vorangegangene Ausgaben zuzulassen. Hier wäre eine tiefergehende Untersuchung wünschenswert. Ein denkbarer Ansatz wäre, beispielsweise `/dev/urandom` als deterministischen RNG zu betrachten und zu berechnen, wie viele Zufallsbits vor einem notwendigen Reseeding nach den Anforderungen von AIS20 ausgelesen werden können. Es wird vermutet, dass dieser Wert den maximalen Stand des Entropiezählers (= 1024) weit übertrifft.

Appendix A Technische Aspekte und Implementierung

Um Aussagen über die Qualität der Zufallszahlen aus dem LRNG zu treffen, muss sein Verhalten zur Laufzeit überwacht werden. Dazu bedarf es einer Instrumentierung des Kerns, mit der die interessanten Parameter ausgelesen werden können, ohne die Aussagekraft dieser Werte wesentlich zu beeinträchtigen.

Der Linux-Kern in der vorliegenden Version implementiert eine Reihe von Tracing-Mechanismen, die zur Laufzeit des Kerns angewendet werden können. Folgende Tracing-Mechanismen sind möglich:

- SystemTap
- Ftrace
- KGDB
- Manuelle Instrumentierung des Quellcodes mittels `printk()`
- `ptrace()` für die Analyse von Systemaufrufen

Im Folgenden werden die angewendeten Mechanismen erläutert. Die Erklärung diskutiert ebenfalls den Einfluss der Messungen auf die Resultate. Da alle Messungen das Zeitverhalten des Linux-Kerns und damit des LRNGs beeinflussen und der Umstand, dass die Entropieabschätzungen des LRNG auf Zeitvarianzen basieren, haben alle Messreihen immer einen Einfluss auf die Ergebnisse.

Es bestehen grundsätzlich viele Möglichkeiten, ein Tracing im Linux-Kern zu implementieren. Selbst wenn der Linux-Kern von Haus aus keine Tracing-Mechanismen implementieren würde, bestünde immer noch die Möglichkeit, den Linux-Kern zu modifizieren, neu zu übersetzen und die Messungen durchzuführen. Solch ein Ansatz hat allerdings gravierende Unzulänglichkeiten bezüglich der Auswahlkriterien für einen angemessenen Tracing-Mechanismus:

- Der Einfluss des Tracing-Mechanismus auf die Messreihen und die Aussagekraft der Messreihen für einen Kern im Normalbetrieb muss vernachlässigbar sein.
- Die Messreihen sollten an einem laufenden System erstellbar sein. D.h., das Einspielen von Patches in den Quellcode des Kerns, ein erneutes Übersetzen und Starten des Kerns sollte vermieden werden. Der Hintergrund ist, dass ein erneutes Übersetzen zwangsläufig zu einem anderen Binärkode führt, da die distributionsspezifische Bauumgebung des originalen Kerns kaum im Labor wiederherstellbar ist.
- Die Messreihen sollen ohne großen Aufwand auf neueren Kernversionen wiederholbar sein.

Im Folgenden werden die verwendeten Tracing-Mechanismen näher erläutert.

A.1 SystemTap

SystemTap stellt eine Infrastruktur im Linux-Kern bereit, um die Sammlung von Informationen über den Linux-Kern zu vereinfachen. Im Gegensatz zu althergebrachten Werkzeugen (Methoden) zur Kernanalyse entfällt bei SystemTap für den Entwickler die Notwendigkeit, Code im Linux-Kern zu instrumentieren, neu zu übersetzen und zu installieren, und einen Neustart mit dem geänderten Code durchzuführen.

SystemTap stellt ein einfaches Kommandozeilen-Programm und eine Skriptsprache zur Verfügung, um Kern-Instrumentierungen für einen laufenden Kern zu schreiben.

Technisch gesehen wird aus dem Skript von dem Kommandozeilen-Programm ein Kernmodul erstellt, welches zusätzliche Servicefunktionen enthält. Dazu gehört ein Mechanismus, Daten vom Kernel-Space zum User-Space zu transportieren. SystemTap verwendet hier DebugFS, das Pufferspeicher effizient in den Speicher von User-Space-Anwendungen kopieren kann. Beim Starten des Tracings wird das erstellte Kernmodul geladen.

SystemTap-Skripte bestehen aus Funktionen, die beim Eintreten bestimmter Ereignisse ausgeführt werden. Auslösende Ereignisse können z.B. sein:

- Aufruf einer spezifizierten Kernfunktion
- Ablauf eines zeitgesteuerten Alarms

Die SystemTap-Infrastruktur ist Teil der betrachteten Version 3.8 des Linux-Kerns.

A.1.1 Messfehler mit SystemTap

Folgende Probleme beim Einsatz von SystemTap-Skripten müssen unbedingt berücksichtigt werden, da diese Messfehler verursachen. Der Kern läuft auf einem Mehrprozessorsystem. Demzufolge besteht immer die Möglichkeit, dass Codesequenzen in `random.c` parallel ausgeführt werden. SystemTap hat keine Möglichkeit, auf diese Parallelität zu reagieren. Damit sind in den Testskripten immer Race-Conditions vorhanden. Vor allem treten diese auf, wenn ein SystemTap-Testskript die Quelle für eine Operation des LRNG feststellen muss. Anhand des Designs wissen wir, dass die zentralen Funktionen für das Einmischen von Daten in einen Entropie-Pool oder das Verändern der Entropieschätzung nicht nur von den Funktionen, welche die Rauschquellen überwachen, aufgerufen werden, sondern auch, wenn Zufallszahlen aus den Pools gelesen werden.

Betrachten wir folgende Testsequenz, welche immer wieder in den Testskripten auftritt – wir nehmen hier an, dass die Entropie eines HID-Ereignisses ermittelt werden soll:

- Ereignis stößt `add_input_randomness` an. Das Testskript setzt einen globalen Statuswert, der anzeigt, dass ein relevantes Ereignis vorliegt.
- Als zweiter Schritt wird die Funktion zum Einmischen aufgerufen. Das Testskript wertet die in Schritt 1 gesetzte globale Variable aus und führt die zentrale Testlogik nur aus, wenn die globale Variable entsprechend gesetzt ist.
- Am Schluss wird die globale Variable zurückgesetzt.

Wenn nun zwischen Schritt 1 und 2 ein Lesen von, z.B. `/dev/random` im normalen Betrieb auftaucht, ist die globale Variable gesetzt, aber die Mischfunktion wurde wegen der Leseanforderung aufgerufen. Das Testskript zeichnet dennoch diese Leseanforderung als validen Test auf. Dies alleine verfälscht die Testergebnisse.

Weiterhin kann vor dem Erreichen von Schritt 3 die Mischfunktion von dem eigentlich zu beobachteten Ereignis aufgerufen werden. Der Test zeichnet wiederum diesen Aufruf auf. Damit haben wir 2 Ereignisse aufgezeichnet anstelle eines Ereignisses.

Dieses Problem ist klar ersichtlich in den Graphen in Abschnitt 6.2.4.3.2 – die Graphen resultierend aus dem Test zeigen mit Abstand die größte Anzahl an aufgefangenen Race-Conditions: die „Gesamtzahl“ der betrachteten Ereignisse ist geringer als die Stichprobenzahl. Der Fehler liegt bei ca. 0,15%. Damit ist die Messgenauigkeit in einem akzeptablen Bereich³¹.

A.1.2 Voraussetzungen für den Einsatz

Für den Einsatz von SystemTap müssen folgende Voraussetzungen erfüllt sein:

- Der Kernel-Quellcode muss für SystemTap erreichbar sein. Normale Linuxdistributionen stellen hierfür sogenannte „Development“-Pakete³² bereit:
 - Ubuntu: `linux-headers-generic`

31 Die Entfernung der Messfehler ist relativ einfach, da die Fehler klar ersichtlich sind: die aufgezeichneten Ereigniswerte der Fehler weichen erheblich von den erwarteten Ereigniswerten ab: Für HID Ereignisse sind die Ereigniswerte entweder kleiner als 1000 (Tastatur) oder knapp unterhalb 2³² (Maus); bei Blockgeräten gibt es nur einen Ereigniswert pro vorhandenem Blockgerät – demzufolge sind alle Ereigniswerte, die abweichen, Messfehler.

32 Diese „Development“-Pakete sind ausreichend, auch wenn diese nicht den gesamten Quellcode enthalten. Sie stellen alle nötigen Header-Dateien und andere Dateien zur Verfügung, die für das Erstellen von Kernmodulen nötig sind.

- Red Hat: kernel-devel

Für selbstübersetzte Kerne muss der Quellcode unter `/lib/module/$(uname -r)/build` zu finden sein.

- Die Debugsymbole für den laufenden Kern müssen installiert sein. Normale Linuxdistributionen stellen hierfür sogenannte „Debug“-Pakete bereit:
 - Ubuntu: `linux-image-$(uname -r)-dbgsym`
 - Red Hat: `kernel-debuginfo`

Für selbstübersetzte Kerne muss die Kernkonfigurationsoption `CONFIG_DEBUG_INFO=y` gesetzt sein. Diese Option ist unter „Kernel Hacking“ → „Compile the kernel with debug info (DEBUG_INFO) [N/y/?]“ → „Kernel debugging (DEBUG_KERNEL) [Y/n/?]“ zu finden.

Falls eine Debian oder Ubuntu Distribution verwendet wird, kann das Paket „kernel-package“ installiert werden und ein neues Debian-Paket aus den Kernelquellen mit folgendem Aufruf erstellt werden:

```
fakeroot make-kpkg --initrd --append-to-version=-upstream kernel_image kernel_headers kernel_debug
```

Anschließend müssen alle generierten Pakete installiert und danach das System neu gestartet werden.

- Die Binärprogramme und Bibliotheken von SystemTap sind zu installieren. Diese sind entweder von der [SystemTap-Homepage](#) oder via folgende Pakete zu beziehen:
 - Ubuntu: `systemtap`
 - Red Hat: `stap`

A.1.3 Einfluss der Messungen auf die Resultate

Aufgrund folgender Eigenschaften der LRNG-Implementierung kann davon ausgegangen werden, dass die Nutzung von SystemTap-Skripten die Entropieabschätzung nicht so verändert, dass die Messergebnisse nicht mehr die erforderliche Aussagekraft haben:

Die Zeitvarianzen, auf welchen die Entropieabschätzung basiert, werden mit folgendem Code implementiert³³:

```
static void add_timer_randomness(struct timer_rand_state *state, unsigned num)
{
    ...
    sample.jiffies = jiffies;
    sample.cycles = random_get_entropy();
    ...
}
```

Quellcode 36: `drivers/char/random.c`

Das Zeitverhalten und damit die Zeitvarianzen eines Ereignisses werden natürlich verändert, wenn SystemTap-Skripte aktiviert werden, die im Code-Pfad des LRNG liegen. Es ist aber zu bedenken, dass das SystemTap-Skript statischen Code enthält und damit einen immer genau gleichen Codeblock zum Zeitverhalten hinzufügt.

Diese Änderung des Zeitverhaltens kann mit dem Addieren einer Konstante auf den beobachteten Zeitwert verglichen werden. Die Entropieberechnung basiert auf Zeitvarianzen zwischen dem aktuellen Ereignis und den drei davor aufgenommenen Ereignissen. Die Schlussfolgerung ist damit: Wenn man auf eine Ereigniszeit eine Konstante aufaddiert, werden die Messungen der Zeitvarianzen damit nicht verändert. Somit ändert sich die Entropieabschätzung nicht.

³³ Die Beschreibung des Quellcodes setzt voraus, dass der Leser bereits Kapitel 2 gelesen hat.

Weiterhin ändert SystemTap nichts an dem zu überprüfenden Code, außer das Skript enthält C-Code, der in die Funktionsweise des Kerns eingreift. Die Skripte für die Testreihen werden aber ohne solchen C-Code implementiert.

Demzufolge ist das Hinzufügen von SystemTap-Skripten unkritisch für die Aussagekraft der Messergebnisse.

Es muss aber sichergestellt werden, dass die Ergebnisse der SystemTap-Skripte erst nach der Messung auf die Festplatte geschrieben werden. Ansonsten führt der Schreibvorgang zu Hardware-Ereignissen, die vom LRNG aufgefangen werden und als Entropie verarbeitet werden. Im Endeffekt wird dabei das Messergebnis verfälscht. Das Verhindern des Schreibens von Messwerten auf die Festplatte kann durch folgende Maßnahmen sichergestellt werden:

- Die SystemTap-Skripte speichern die Messwerte in internen Puffern und übergeben diese Werte erst an den User-Space, nachdem die Messung beendet ist.
- Der User-Space-Prozess, welcher die Messwerte des SystemTap-Skripts aufzeichnet, schreibt die Werte in ein TmpFS Dateisystem. Dieses Dateisystem wird im RAM ohne Blockgeräte abgebildet und erzeugt demzufolge keine Blockgeräte-Ereignisse.

A.2 Testdurchführung

A.2.1 Vorbetrachtungen

Die Aussagen der im Folgenden aufgeführten Testreihen zur Messung der Zustände im LRNG gelten zur Zeit nur, wenn keine Hardware-RNGs verwendet werden. Wie in Abschnitt 2.7 dargelegt, existiert zur Zeit nur die Unterstützung für die RDRAND-Instruktion von Intel. Um auszuschließen, dass die aus den SystemTap-Skripten generierten Daten von der RDRAND-Instruktion beeinflusst werden, sind alle Skripte so programmiert, dass sie mit einem Fehler abbrechen, wenn in der CPU die RDRAND-Instruktion gefunden wurde.

Für jede generierte Testreihe wird das verwendete SystemTap-Skript referenziert.

Die Graphen wurden alle mit dem Programm R-Project³⁴ erzeugt. Dieses Programm nutzt eine Programmiersprache, um Daten zu verarbeiten, statistische Analysen zu erstellen und graphisch aufzubereiten. Jeder Graph wurde mit einem eigenen R-Project-Programm aus den Rohdaten der SystemTap-Skripte erzeugt. Um die Zusammengehörigkeit der SystemTap-Skripte und die R-Project-Programme kenntlich zu machen, wurden die gleichen Dateinamen verwendet, die nur einen unterschiedlichen Suffix haben.

A.2.2 Aufruf von SystemTap-Skripten

Der Aufruf der einzelnen SystemTap-Skripte ist als Kommentar in jedem Skript enthalten. Die generierten Daten müssen in einer zu dem Skriptnamen gleichnamigen Datei mit dem Suffix „.data“ im selben Verzeichnis gespeichert werden, damit sie von den R-Project-Programmen gelesen werden können.

A.2.3 Aufruf von R-Project Programmen

Die Autoren stellen das Programm „update-graphs.sh“ zur Verfügung, welches alle Graphen in allen vorhandenen Unterverzeichnissen neu erstellt.

Alle im Folgenden eingebundenen Graphen sind dynamische Links. Demzufolge werden alle neu erzeugten Graphen automatisch importiert.

A.2.4 Verwendete Hard- und Software

Die vorliegenden Testreihen wurden auf folgender Hardware durchgeführt:

- Thinkpad T520
- 2-Kern-CPU mit jeweils 2 Hyperthreads
- Intel(R) Core(TM) i7-2620M CPU @ 2.70GHz

³⁴ <http://www.r-project.org/>

- CPU Stepping: 7

und

- Thinkpad T500
- 2-Kern-CPU mit jeweils 2 Hyperthreads
- Intel(R) Core(TM) i7-620M CPU @ 2.10GHz
- CPU Stepping: 7

Es wurde folgender Kern verwendet:

- Ubuntu Oneric Ocelot 3.2.0-26.4.1
- 64 Bit Wortgröße

A.2.5 Nachbildung der Einsatzumgebung

Das Kapitel 2 beschreibt den LRNG als einen Mechanismus, der auf Hardware-Ereignisse reagiert. Das Eintreffen von Hardware-Ereignissen hängt direkt davon ab, wie das Betriebssystem verwendet wird.

Zum Beispiel ist bei dem Einsatz von Linux als Desktop-PC mit einer X11-Oberfläche eine große Anzahl an Entropiequellen vorhanden, da die Eingabegeräte wie Maus oder Tastatur häufig verwendet werden. Dem gegenüber steht ein „Headless“-Serversystem (d.h. ein System ohne angeschlossene Eingabegeräte), wie es in einem Serverraum stehen könnte. Hier kann keine Entropie über die Eingabegeräte abgegriffen werden.

Für die Testreihen wird demnach folgende Einsatzumgebung definiert und entsprechend simuliert:

- Ubuntu Desktop Installation mit KDE
- Vorhandensein von typischen Büroanwendungen (LibreOffice, Thunderbird, KMail, Firefox)
-

Appendix B Neuerungen im LRNG

Das vorliegende Dokument befasst sich mit einer genau definierten Version des Linux-Kerns: 3.8. Für neue Kernversionen muss dieses Dokument einer Überprüfung unterzogen werden. Dabei ist der Kern auf folgende Änderungen hin zu überprüfen.

- Alle Änderungen der in Kapitel 1 genannten Dateien sind zu analysieren.
- Änderungen in den Aufrufen der in Abschnitt 2.5.1.1, 2.5.1.2, und 2.5.1.3 besprochenen Funktionen zum Sammeln von Entropie sind zu analysieren. Insbesondere sind neue Bedingungen beim Aufruf dieser Funktionen zu prüfen.
- Es ist zu prüfen, ob gegebenenfalls neue Funktionen in `random.c`, welche nicht als „static“ markiert sind, im restlichen Kern verwendet werden. Diese Funktionen sind möglicherweise neue Schnittstellen, sowohl zum Sammeln von Entropie, als auch zum Extrahieren von Zufallszahlen.

Wenn einer der genannten Punkte eine Änderung aufweist, ist diese in den folgenden Kapiteln zu dokumentieren und deren Einfluss auf die in diesem Dokument stehenden Aussagen zu erklären.

B.1 Linux Kern 3.5

Vorherige Kernversion: Ubuntu Oneric Ocelot 3.2.0-26.4.1

Derzeitige Kernversion: [Linux-Kern 3.5](#)

Die folgenden Abschnitte enthalten die Prüfung auf Änderungen.

B.1.1 Änderung von in Kapitel 1 genannten Dateien

B.1.1.1 `drivers/char/random.c`

Diff: siehe `kernelupdates/3.5/random.c.diff`

Folgende Änderungen sind im Diff ersichtlich:

- Änderung der Debug-Variable von `int` zu `bool`: Kein Einfluss, da diese Variable in der Anwendung nur 0 oder 1 enthält.
Die Änderung der Größe des verwendeten Speichers ist irrelevant für den LRNG.
- `add_timer_randomness`: Die Änderung des Typs von `cycles_t` nach `unsigned int` wird für den Einsatz von `get_cycles` gebraucht. `cycles_t` war entweder als `unsigned long long` (x86) oder `unsigned long` (alle anderen Architekturen) definiert. Damit impliziert die Änderung des Typs eine Verkleinerung der Zeitvariable im Verhältnis zu den anderen Daten, die als Roh-Entropie zum `input_pool` hinzugefügt werden.

Damit ist eine Änderung des Verhaltens des LRNG theoretisch möglich. Es ist aber nicht davon auszugehen, dass diese Änderung eine Auswirkung auf die Qualität der Zufallszahlen hat. Folgende Gründe sind zu nennen:

- 2^{32} Nanosekunden entsprechen etwa 4 Sekunden – dies ist die Genauigkeit der Variable unter der Annahme eines 1 GHz Prozessors.
- Die Funktion `add_timer_randomness` verknüpft den Nanosekundenwert mit den Jiffies mittels Konkatenation. Damit werden die sich schnell ändernden Bits mit langsam ändernden Bits. Dies bedeutet, dass der resultierende Wert immer noch einen großen Zeitbereich abdeckt.
- Die Jiffies verändern sich alle 4ms oder 1ms. Damit wird es in einer für den LRNG zur Entropie-Abschätzung relevanten Zeit keinen Überlauf der Jiffies-Variable geben.
- Die Konkatenation von Jiffies und Prozessorzyklen stellt nun sicher, dass die sich schnell ändernden Prozessorzyklen mit den sich langsamen ändernden Jiffies verknüpft werden. Vor der Änderung wurden auch die sich langsam ändernden Prozessorzyklen mit den Jiffies verknüpft, welches eine doppelte Nutzung der sich

langsam ändernden Werte bedeutet. Diese doppelte Nutzung wurde eliminiert. Auch ist darauf hinzuweisen, dass die hochwertigen Bits der Prozessorzyklen und die Jiffies voneinander abhängen. Der Patch stellt aber sicher, dass immer noch alle verfügbaren Zeit-Informationen in den Zeitstempel einfließen.

- `add_timer_randomness`: Die unbedingte Verwendung von `get_cycles` wird ersetzt durch das Lesen eines `int`-Wertes vom Hardware-RNG, wenn dieser vorhanden ist. Ansonsten wird `get_cycles` verwendet. Da die Variable `cycles` (32 Bit) kleiner ist, als `get_cycles` (64 Bit), werden die 32 niederwertigsten Bits des Zeitstempels von `get_cycles()` verwendet. Folgendes Programm zeigt, dass GCC die niederwertigsten Bits verwendet, da der Output "55667788" ist:

```
#include <stdio.h>

unsigned long long b()
{
    return(0x1122334455667788);
}

int main()
{
    unsigned a = 0;

    a = b();

    printf("%x\n", a);
}

Quellcode 37: Behandlung des Casting von long long nach unsigned int
```

Demzufolge werden die sich am schnellsten ändernden Bits verwendet.

Damit hat diese Änderung keinen Einfluss auf den LRNG, vorausgesetzt der Hardware-RNG wird nicht verwendet.

- `init_std_data`: Bei der Initialisierung des LRNG während des Startens des Kerns werden die Entropie-Pools mit Daten aus dem Hardware-RNG durchmischt. Dabei werden diese Daten mit der Funktion entsprechend Abschnitt 2.5.2 in die Pools eingefügt.

Diese Änderung hat keinen Einfluss auf den LRNG, vorausgesetzt der Hardware-RNG wird nicht verwendet.

- `proc_do_uid`: Diese Funktion ist eine Hilfsfunktion für andere Teile des Kerns und wird nicht im LRNG verwendet.

Demzufolge hat diese Änderung keinen Einfluss auf den LRNG.

B.1.1.2 include/linux/random.h

Das Diff zeigt keine Änderungen und damit keine Auswirkungen.

B.1.1.3 arch/x86/include/asm/archrandom.h

Das Diff zeigt keine Änderungen und damit keine Auswirkungen.

B.1.2 Änderungen in Aufrufen der Entropiesammelfunktionen

B.1.2.1 add_input_randomness

Keine Änderungen in Aufruf in Funktion `input_event` und damit keine Auswirkungen.

B.1.2.2 add_interrupt_randomness

Keine Änderungen in Aufruf in Funktionen `ab3100_irq_handler`³⁵ und `handle_irq_event_percpu` und damit keine Auswirkungen.

B.1.2.3 add_disk_randomness

Keine Änderungen in Aufruf in Funktionen `blk_update_bidi_request` und damit keine Auswirkungen.

B.1.3 Definition und Verwendung von neuen Schnittstellen

Es sind keine neuen Funktionen in `random.c` mit den Symbolen `EXPORT_SYMBOL*()` versehen. Damit gibt es keine neuen Schnittstellen.

B.2 Linux Kern 3.6

Vorherige Kernversion: [Linux-Kern 3.5](#)

Derzeitige Kernversion: [Linux-Kern 3.6](#)

Die folgenden Abschnitte enthalten die Prüfung auf Änderungen.

B.2.1 Änderung von in Kapitel 1 genannten Dateien

B.2.1.1 drivers/char/random.c

Diff: siehe `kernelupdates/3.6/random.c.diff`

Folgende Änderungen sind im Diff ersichtlich:

- Änderung der Datenstruktur `struct entropy_store`:
 - `input_rotate`: Das Verschieben der Variable und die Änderung von `int` zu `unsigned int` hat keine Auswirkung auf den LRNG. Dennoch ist eine kleine Unsauberkeit im Code ersichtlich: die Funktion `_mix_pool_bytes` weist den Inhalt von `input_rotate` einer `signed int` Variable zu. Da diese Variable aber ausschließlich als Bitmaske verwendet wird, ist ein Wrap in den negativen Bereich als harmlos anzusehen.
 - `entropy_total`: Diese Variable ist neu und enthält einen Zähler der die gesamte Entropie, die für einen Entropie-Pool erzeugt wurde, bis zum Überschreiten der Schranke von 128 (Bit) aufsummiert. Dieser Zähler wird dazu verwendet, die `initialized` Variable auf 1 (`true`) zu setzen, wenn der Zähler 128 (Bit) überschreitet.
 - `Initialized`: Bei der Neufassung der Funktion `add_interrupt_randomness` wird diese Variable verwendet, um den Entropie-Pool auszuwählen, welcher mit Entropie von Interrupt-Ereignissen gespeist wird. Ist diese Variable für den `nonblocking_pool` auf 1 (`true`) gesetzt, wird der `input_pool` als Ziel verwendet. Ansonsten wird der `nonblocking_pool` verwendet. Siehe unten für weitere Ausführungen zu `add_interrupt_randomness`.
- Definition von `twist_table` als `static const`: diese Variable wird nun an zwei Stellen des LRNGs verwendet und wird demnach als globale Variable einmal definiert.

Diese Änderung hat keine Auswirkung auf den LRNG.
- Generell: der LRNG Code hat mehrere Änderungen, welche `FTRACE`-Hooks zur Verfügung stellen. `FTRACE` ist ein Mechanismus, um den Linux-Kern zur Laufzeit zu beobachten, ohne den Aufwand eines Debuggers mitzuschleppen. Diese Änderungen sind an Funktionsaufrufen zu erkennen, deren Namen „`trace_`“ als Prefix beinhalten.

Diese Änderung hat keine Auswirkung auf die Ausführungslogik des LRNG.

35 Obwohl diese in einen Treiber eingebundene Funktion vorher nicht beschrieben wurde, ruft sie direkt die Entropiesammelfunktion auf.

- Funktion `_mix_pool_bytes`: Die Funktion `mix_pool_bytes_extract` wurde zu `_mix_pool_bytes` umbenannt. Der Code nutzt Memory Barriers um die Variable `input_rotate` und `add_ptr` zu schützen. Dies verändert die Ausführungsgeschwindigkeit und die Synchronisation bei einer parallelen Ausführung des LRNG auf mehreren CPUs.
Diese Änderung hat keine Auswirkung auf die Ausführungslogik des LRNG.
- Funktion `__mix_pool_bytes`: Diese Funktion ist neu und unterstützt die FTRACE-Implementierung.
Diese Änderung hat keine Auswirkung auf die Ausführungslogik des LRNG.
- Funktion `mix_pool_bytes`: Diese Funktion ist neu und unterstützt die FTRACE-Implementierung.
Diese Änderung hat keine Auswirkung auf die Ausführungslogik des LRNG.
- Datenstruktur `fast_pool`, Funktion `fast_mix`, Entfernen von `irq_timer_state` und dessen Hilfsfunktionen, Neufassung von `add_interrupt_randomness`, Entfernen von `rand_initialize_irq`: Vollständig neues Design wie Interrupts verwendet werden. Die Änderungen sind relativ klein, haben aber eine große Auswirkung und können nicht in diesem Rahmen vollständig diskutiert werden.
Diese Änderung hat eine massive Änderung des LRNG zur Folge.
- Funktion `credit_entropy_bits`: der `nonblocking_pool` wird beim Starten des Kerns direkt mit Hardware-Ereignissen gefüttert. Erst nachdem dieser Pool genügend Entropie bekommen hat, schaltet der LRNG auf den `input_pool` um. Der Hintergrund ist die rechtzeitige Versorgung von `nonblocking_pool` mit Entropie beim Starten, da dieser Pool als Quelle für viele Zufallszahlen während des Startens verwendet wird.
Diese Änderung hat eine kleine Auswirkung auf die Funktionsweise des LRNG, da die ersten 128 Hardware-Ereignisse direkt in den `nonblocking_pool` eingemischt werden. D.h. der `input_pool` wird erst ab dem 129sten Ereignis durchmischt. Beim Startvorgang sind aber 128 Hardware-Ereignisse innerhalb weniger Millisekunden durch die hohe Festplattenaktivität erreicht. Damit hat diese Änderung praktisch keine Auswirkungen auf die Ausführungslogik des LRNG zur Laufzeit des Systems.
- Funktion `add_device_randomness`: Vollständig neue Quelle von Entropie, welche nur zur Initialisierung der Entropie-Pools verwendet wird.
Diese Änderung hat eine kleine Auswirkung auf die Funktionsweise des LRNG.
- Funktion `extract_buf`: der Output von Hardware RNGs (z.Z. Intel RDRAND-Instruktion) wird immer mittels XOR mit der LRNG-Zufallszahl verknüpft, falls solche Hardware vorhanden ist.
Diese Änderung hat eine kleine Auswirkung auf die Funktionsweise des LRNG. Durch die Nutzung der XOR Verknüpfung kann damit niemals eine Verschlechterung der Daten aus `/dev/random` oder `/dev/urandom` erfolgen. Falls der Hardware RNG Entropie liefert, wird die Qualität der Zufallsdaten aus `/dev/random` nicht nennenswert besser, da diese Analyse zeigt, dass bereits ein Bit aus `/dev/random` fast ein Bit an Entropie liefert. Hingegen wird die Qualität von `/dev/urandom` massiv verbessert vor allem wenn zu wenig Entropie aus Hardware-Ereignissen im Verhältnis zu der Menge an zu generierenden Zufallszahlen gesammelt wird. Wenn kein Hardware-Zufallszahlengenerator vorhanden ist, hat diese Änderung keine Auswirkung auf den LRNG.
- Funktion `init_std_data`: Nutzung des Hardware RNG während der Initialisierung, um die Entropie-Pools vor der Nutzung durch den LRNG mit Zufallszahlen vorzubereiten.
Diese Änderung hat eine kleine Auswirkung auf die Funktionsweise des LRNG zur Startzeit (der Entropie-Pool ist mit zufälligen Werten gefüllt. Die Änderung hat aber keine Auswirkung zur Laufzeit.

B.2.1.2 include/linux/random.h

Diff: siehe kernelupdates/3.6/random.h.diff

Folgende Änderungen sind in dem Diff ersichtlich:

- Aktualisierungen der Funktionsdefinitionen der Funktionen, welche in random.c implementiert sind. Diese Änderung hat keinen Einfluss auf den LRNG.

B.2.1.3 arch/x86/include/asm/archrandom.h

Das Diff zeigt keine Änderungen und damit keine Auswirkungen.

B.2.2 Änderungen in Aufrufen der Entropiesammelfunktionen

B.2.2.1 add_input_randomness

Keine Änderungen in Aufruf in Funktion `input_event` und damit keine Auswirkungen.

B.2.2.2 add_interrupt_randomness

Vollständiges Re-Design der Nutzung von Interrupts als Rauschquelle. Die entsprechenden Abschnitte in Kapitel 2 wurden angepasst.

B.2.2.3 add_disk_randomness

Keine Änderungen in Aufruf in Funktionen `blk_update_bidi_request` und damit keine Auswirkungen.

B.2.3 Definition und Verwendung von neuen Schnittstellen

Folgende neuen Funktionen in random.c mit den Symbolen `EXPORT_SYMBOL*()` versehen:

- `add_device_randomness`: Neue Entropie-Sammelfunktion
- `get_random_bytes_arch`: Falls ein Hardware RNG vorhanden ist, wird der für die Zufallszahl verwendet. Ansonsten gleiche Logik wie `get_random_bytes`.

B.3 Linux-Kern 3.7

Vorherige Kernversion: [Linux-Kern 3.6](#)

Derzeitige Kernversion: [Linux-Kern 3.7](#)

Die folgenden Abschnitte enthalten die Prüfung auf Änderungen.

B.3.1 Änderung von in Kapitel 1 genannten Dateien

B.3.1.1 drivers/char/random.c

Diff: keine Änderungen.

B.3.1.2 include/linux/random.h

Diff: siehe [kernelupdates/3.7/random.h.diff](#)

Folgende Änderungen sind im Diff ersichtlich:

- Entfernung der IOCTL-Definitionen, welche in die Datei `include/uapi/linux/random.h` verschoben wurden (keine inhaltlichen Änderungen)

Keine Auswirkung auf die Ausführungslogik des LRNG.

B.3.1.3 include/uapi/linux/random.h

Diff: siehe [kernelupdates/3.7/uapi_random.h.diff](#)

Folgende Änderungen sind im Diff ersichtlich:

- Neue Datei, welche die IOCTL-Definitionen enthält.

Keine Auswirkung auf die Ausführungslogik des LRNG.

B.3.1.4 arch/x86/include/asm/archrandom.h

Das Diff zeigt keine Änderungen und damit keine Auswirkungen.

B.3.2 Änderungen in Aufrufen der Entropiesammelfunktionen

B.3.2.1 add_input_randomness

Keine Änderungen in Aufruf in Funktion `input_event` und damit keine Auswirkungen.

B.3.2.2 add_interrupt_randomness

Keine Änderungen in Aufruf in Funktion `handle_irq_event_percpu` und damit keine Auswirkungen.

B.3.2.3 add_disk_randomness

Keine Änderungen in Aufruf in Funktionen `blk_update_bidi_request` und damit keine Auswirkungen.

B.3.3 Definition und Verwendung von neuen Schnittstellen

Keine Änderungen.

B.4 Linux-Kern 3.8

Vorherige Kernversion: [Linux-Kern 3.7](#)

Derzeitige Kernversion: [Linux-Kern 3.8](#)

Die folgenden Abschnitte enthalten die Prüfung auf Änderungen.

B.4.1 Änderung von in Kapitel 1 genannten Dateien

B.4.1.1 drivers/char/random.c

Diff: siehe [kernelupdates/3.8/random.c.diff](#).

Folgende Änderungen sind im Diff ersichtlich:

- Debug logging leicht verändert.
Diese Änderung hat keine Auswirkung auf die Ausführungslogik des LRNG.
- Hinzufügen von `last_data_init` und das „Priming“ des FIPS 140-2 Continuous Test entsprechend der FIPS 140-2-Spezifikation, Kap 4.9.2.
Diese Änderung hat keine Auswirkung auf die Ausführungslogik des LRNG mit der Ausnahme, dass die erste generierte Zufallszahl nicht an den Aufrufer übergeben wird, sondern eine neue Zufallszahl erzeugt wird.
- Neusortierung der Fehlerbehandlung in `random_read`.
Diese Änderung hat keine Auswirkung auf die Ausführungslogik des LRNG.

B.4.1.2 include/linux/random.h

Diff: siehe [kernelupdates/3.8/random.h.diff](#)

Folgende Änderungen sind im Diff ersichtlich:

- Leichte Re-definition von Funktionen, welche in `lib/random.c` implementiert sind. Diese Funktionen sollen in Zukunft eliminiert werden.
Keine Auswirkung auf die Ausführungslogik des LRNG.

- Entfernung der IOCTL-Definitionen, welche in die Datei `include/uapi/linux/random.h` verschoben wurden (keine inhaltlichen Änderungen)
Keine Auswirkung auf die Ausführungslogik des LRNG.

B.4.1.3 include/uapi/linux/random.h

Das Diff zeigt keine Änderungen und damit keine Auswirkungen.

B.4.1.4 arch/x86/include/asm/archrandom.h

Das Diff zeigt keine Änderungen und damit keine Auswirkungen.

B.4.2 Änderungen in Aufrufen der Entropiesammelfunktionen

B.4.2.1 add_input_randomness

Keine Änderungen des Aufrufs in Funktion `input_event` und damit keine Auswirkungen.

B.4.2.2 add_interrupt_randomness

Keine Änderungen in Aufruf in Funktion `handle_irq_event_percpu` und damit keine Auswirkungen.

B.4.2.3 add_disk_randomness

Keine Änderungen in Aufruf in Funktionen `blk_update_bidi_request` und damit keine Auswirkungen.

B.4.3 Definition und Verwendung von neuen Schnittstellen

Keine Änderungen.

B.5 Linux-Kern 3.9

Vorherige Kernversion: [Linux-Kern 3.8](#)

Derzeitige Kernversion: [Linux-Kern 3.9](#)

Die folgenden Abschnitte enthalten die Prüfung auf Änderungen.

B.5.1 Änderung von in Kapitel 1 genannten Dateien

B.5.1.1 drivers/char/random.c

Diff: siehe `kernelupdates/3.9/random.c.diff`.

Folgende Änderungen sind im Diff ersichtlich:

- Initialisierung der Spinlocks der Entropie-Pools verändert. Anstelle eines Pointers verarbeitet die Spinlock-Initialisierungsfunktion nun den direkten Wert.
Diese Änderung hat keine Auswirkung auf die Ausführungslogik des LRNG.
- Aufwecken der Prozesse mit einem Poll auf `/dev/random` oder `/dev/urandom` ist ans Ende der Funktion verschoben worden, um den schützenden Spinlock vor dem aufwecken zu lösen. Der Entropie-Pool wird beim aufwecken der Prozesse nicht mehr gelesen oder verändert. Demzufolge ist ein Spinlock um diese Funktion nicht notwendig und erlaubt andere Entropie-Pool Nutzer schneller auf den Entropie-Pool zuzugreifen.

Diese Änderung hat keine Auswirkung auf die Ausführungslogik des LRNG.

B.5.1.2 include/linux/random.h

Diff: siehe `kernelupdates/3.9/random.h.diff`

Folgende Änderungen sind im Diff ersichtlich:

- Hinzufügen der Hilfsfunktion `next_pseudo_random32`, welche nicht als offizielle Schnittstelle des LRNG definiert ist. Es ist zu beachten, dass zusätzlich zu dem LRNG weitere Funktionen in `random.c` und `random.h` implementiert sind. Siehe Kapitel 2.6.2 für weitere Informationen.

Keine Auswirkung auf die Ausführungslogik des LRNG.

B.5.1.3 include/uapi/linux/random.h

Das Diff zeigt keine Änderungen und damit keine Auswirkungen.

B.5.1.4 arch/x86/include/asm/archrandom.h

Das Diff zeigt keine Änderungen und damit keine Auswirkungen.

B.5.2 Änderungen in Aufrufen der Entropiesammelfunktionen

B.5.2.1 add_input_randomness

Keine Änderungen des Aufrufs in Funktion `input_pass_values` und damit keine Auswirkungen.

B.5.2.2 add_interrupt_randomness

Keine Änderungen in Aufruf in Funktion `handle_irq_event_percpu` und damit keine Auswirkungen.

B.5.2.3 add_disk_randomness

Keine Änderungen in Aufruf in Funktionen `blk_update_bidi_request` und damit keine Auswirkungen.

B.5.2.4 add_device_randomness

Keine Änderungen der Nutzung der Funktion in Initialisierungsfunktionen von Gerätetreibern.

B.5.3 Definition und Verwendung von neuen Schnittstellen

Keine Änderungen und Neuerungen von Schnittstellen zu den Entropie-Pools.

B.6 Linux-Kern 3.10

Vorherige Kernversion: [Linux-Kern 3.9](#)

Derzeitige Kernversion: [Linux-Kern 3.10](#)

Die folgenden Abschnitte enthalten die Prüfung auf Änderungen.

B.6.1 Änderung von in Kapitel 1 genannten Dateien

B.6.1.1 drivers/char/random.c

Diff: siehe [kernelupdates/3.10/random.c.diff](#).

Folgende Änderungen sind im Diff ersichtlich:

- Die Variable für den Entropieschätzer wird mittels einer atomaren Operation verändert. Die bisherige Implementierung hatte eine Race-Condition, bei der ein Update des Entropieschätzers verloren gehen konnte.

Diese Änderung hat keine Auswirkung auf die Ausführungslogik des LRNG.

- Die Prüfung ob der FIPS 140-2 Test initialisiert wurde, ist an den Anfang der Extraktionsfunktion geschoben worden. Dieser Test wird nun nicht mehr pro RNG-Runde durchgeführt, sondern nur noch einmal pro Aufruf.

Diese Änderung hat keine Auswirkung auf die Ausführungslogik des LRNG.

B.6.1.2 include/linux/random.h

Diff: siehe kernelupdates/3.10/random.h.diff

Folgende Änderungen sind im Diff ersichtlich:

- Entfernen von Hilfsfunktionen, welche bereits als ausgemustert markiert worden. Diese Funktionen werden nicht vom LRNG verwendet.

Keine Auswirkung auf die Ausführungslogik des LRNG.

B.6.1.3 include/uapi/linux/random.h

Das Diff zeigt keine Änderungen und damit keine Auswirkungen.

B.6.1.4 arch/x86/include/asm/archrandom.h

Das Diff zeigt keine Änderungen und damit keine Auswirkungen.

B.6.2 Änderungen in Aufrufen der Entropiesammelfunktionen

B.6.2.1 add_input_randomness

Keine Änderungen des Aufrufs in Funktion `input_pass_values` und damit keine Auswirkungen.

B.6.2.2 add_interrupt_randomness

Keine Änderungen in Aufruf in Funktion `handle_irq_event_percpu` und damit keine Auswirkungen.

B.6.2.3 add_disk_randomness

Keine Änderungen in Aufruf in Funktionen `blk_update_bidi_request` und damit keine Auswirkungen.

B.6.2.4 add_device_randomness

Keine Änderungen der Nutzung der Funktion in Initialisierungsfunktionen von Gerätetreibern.

B.6.3 Definition und Verwendung von neuen Schnittstellen

Keine Änderungen und Neuerungen von Schnittstellen zu den Entropie-Pools.

B.7 Linux-Kern 3.11

Vorherige Kernversion: [Linux-Kern 3.10](#)

Derzeitige Kernversion: [Linux-Kern 3.11](#)

Die folgenden Abschnitte enthalten die Prüfung auf Änderungen.

B.7.1 Änderung von in Kapitel 1 genannten Dateien

B.7.1.1 drivers/char/random.c

Diff: siehe kernelupdates/3.11/random.c.diff.

Folgende Änderungen sind im Diff ersichtlich:

- Ein typedef für eine Struktur außerhalb der LRNG-Verarbeitung wurde entfernt und durch die direkte Typdefinition ersetzt.

Diese Änderung hat keine Auswirkung auf die Ausführungslogik des LRNG.

B.7.1.2 include/linux/random.h

Das Diff zeigt keine Änderungen und damit keine Auswirkungen.

B.7.1.3 include/uapi/linux/random.h

Das Diff zeigt keine Änderungen und damit keine Auswirkungen.

B.7.1.4 arch/x86/include/asm/archrandom.h

Das Diff zeigt keine Änderungen und damit keine Auswirkungen.

B.7.2 Änderungen in Aufrufen der Entropiesammelfunktionen

B.7.2.1 add_input_randomness

Keine Änderungen des Aufrufs in Funktion `input_pass_values` und damit keine Auswirkungen.

B.7.2.2 add_interrupt_randomness

Keine Änderungen in Aufruf in Funktion `handle_irq_event_percpu` und damit keine Auswirkungen.

B.7.2.3 add_disk_randomness

Keine Änderungen in Aufruf in Funktionen `blk_update_bidi_request` und damit keine Auswirkungen.

B.7.2.4 add_device_randomness

Keine Änderungen der Nutzung der Funktion in Initialisierungsfunktionen von Gerätetreibern.

B.7.3 Definition und Verwendung von neuen Schnittstellen

Keine Änderungen und Neuerungen von Schnittstellen zu den Entropie-Pools.

B.8 Linux-Kern 3.12

Vorherige Kernversion: [Linux-Kern 3.11](#)

Derzeitige Kernversion: [Linux-Kern 3.12](#)

Die folgenden Abschnitte enthalten die Prüfung auf Änderungen.

B.8.1 Änderung von in Kapitel 1 genannten Dateien

B.8.1.1 drivers/char/random.c

Diff: siehe [kernelupdates/3.12/random.c.diff](#).

Folgende Änderungen sind im Diff ersichtlich:

- Header Datei `linux/irq.h` wird nun immer inkludiert und nicht nur, wenn die Kompilierzeit-Kernel-Option `CONFIG_GENERIC_HARDIRQS` gesetzt ist. Diese header Datei stellt den Link zu den Linux-Kernel Interrupt-Funktionen her, welcher für die `add_interrupt_randomness` Funktion benötigt wird. Da keine funktionelle Änderung von `add_interrupt_randomness` und deren Servicefunktionen vorliegt, ist die Änderung als „Aufräumarbeit“ einzustufen.

Diese Änderung hat keine Auswirkung auf die Ausführungslogik des LRNG.

- In der gesamten Implementierung des LRNGs wird der Funktionsaufruf von `get_cycles` (diese Funktion liefert den hochauflösenden Zeitstempel) durch `random_get_entropy` ersetzt. Das Ziel ist, dass die Maintainer der verschiedenen Architekturen im Linux-Kern darauf hingewiesen werden, dass diese Implementierung wichtig für den LRNG ist – damit soll so schnell wie möglich alle Architekturen mit einer entsprechenden Implementierung eines Zeitgebers versehen werden (einige Architekturen haben keine Implementierung von `get_cycles` / `random_get_entropy`!).

Diese Änderung hat keine Auswirkung auf die Ausführungslogik des LRNG. Dennoch wurde die Designbeschreibung angepasst.

- Umdeklarierung einer statischen Funktion in eine nicht-statische Funktion, damit diese Funktion zur Boot-Zeit des Kerns ausgeführt werden kann. Diese Funktion füllt eine Variable mit Zufallszahlen, um die API Funktion `get_random_int`, welche nicht als Teil des LRNG angesehen wird, zu unterstützen.

Diese Änderung hat keine Auswirkung auf die Ausführungslogik des LRNG.

B.8.1.2 include/linux/random.h

Diff: siehe `kernelupdates/3.12/random.h.diff`.

Folgende Änderungen sind im Diff ersichtlich:

- Export der oben genannten, umdeklarierten Funktion.

Diese Änderung hat keine Auswirkung auf die Ausführungslogik des LRNG.

B.8.1.3 include/uapi/linux/random.h

Das Diff zeigt keine Änderungen und damit keine Auswirkungen.

B.8.1.4 arch/x86/include/asm/archrandom.h

Das Diff zeigt keine Änderungen und damit keine Auswirkungen.

B.8.2 Änderungen in Aufrufen der Entropiesammelfunktionen

B.8.2.1 add_input_randomness

Keine Änderungen des Aufrufs in Funktion `input_pass_values` und damit keine Auswirkungen.

B.8.2.2 add_interrupt_randomness

Keine Änderungen in Aufruf in Funktion `handle_irq_event_percpu` und damit keine Auswirkungen.

B.8.2.3 add_disk_randomness

Keine Änderungen in Aufruf in Funktionen `blk_update_bidi_request` und damit keine Auswirkungen.

B.8.2.4 add_device_randomness

Keine Änderungen der Nutzung der Funktion in Initialisierungsfunktionen von Gerätetreibern.

B.8.3 Definition und Verwendung von neuen Schnittstellen

Keine Änderungen und Neuerungen von Schnittstellen zu den Entropie-Pools.

Appendix C Mapping zum Dokument von T-Systems

Zur einfacheren Vergleichbarkeit des vorliegenden Dokuments zum vorherigen Dokument von T-Systems (siehe [LLT07]), stellt folgenden Tabelle den Inhalt der Kapitel grob gegenüber.

T-Systems Kapitel	Vorliegendes Dokument Kapitel
1	1
2	Entfallen/ersetzt durch andere Informationen
3	2
-	3
-	4
4	5,6
-	7
5	8

Aus der Tabelle wird ersichtlich, dass in dieser Untersuchung zusätzlich eine kurze Zusammenfassung von bekannten Analysen des LRNG (siehe Kapitel 3) eingefügt wurde, sowie eine Überprüfung des LRNG auf Abdeckung der NTG.1-Anforderungen (siehe Kapitel 4).

Des Weiteren werden die von T-Systems durchgeführten Tests aufgelistet und mit dem vorliegende Dokument verglichen.

T-Systems Tests	Tests im vorliegenden Dokument
Test in Abschnitt 4.1	Ersetzt durch Tests in Abschnitt 5.2.6, 5.2.7, 5.2.8 , die die Verteilung der Entropie-Pools analysieren.
Test in Abschnitt 4.2	Abschnitt 6.1.2
Test in Abschnitt 4.3.1	Abschnitt 6.2.3
Tests in Abschnitt 4.3.2	Abschnitt 6.2.4 und Abschnitt 6.2.6 Folgende Tests sind nicht vorhanden: <ul style="list-style-type: none"> • Durchführung der Fouriertransformation von Beobachtungswerten • Analyse der Verteilung der Bits bei den Differenzen der Prozessorzyklen • Tests, welche die T-Systems Abbildungen 15ff durchführen – es ist unklar, auf welche Komponenten sich die Tests beziehen.
Tests in Abschnitt 4.3.4	Abschnitt 6.2.7

Wir bemerken an dieser Stelle, dass hier auf die Verwendung der Fouriertransformation zur Identifizierung von Regelmäßigkeiten innerhalb der gemessenen Werte verzichtet wurde, da die Ergebnisse von T-Systems entweder ohne Befund waren oder Signale aufgedeckt haben, für die keine Erklärung gefunden werden konnte.

Appendix D Abkürzungen und Glossar

IOCTL	Input / Output Control – Systemaufruf bei Unix
IOMMU	Input / Output Memory Management Unit
MSI	Message Signaled Interrupt
Jiffies	„Im Bereich der Computertechnik steht ein Jiffy für die Periodendauer des Timer-Interrupts und stellt damit eine betriebssystem- und hardwareabhängige Maßeinheit für die Anzahl der Zeitschlitze (engl. timeslices), die ein Prozess benötigt, dar.“ ³⁶ In Linux wird dieser Wert ist normalerweise alle 4 ms (== 250 Hz) erhöht. Zur Kompilierzeit wird dieser Wert auf 250 Hz als Standardwert für x86 gesetzt, kann aber mittels des Kern-Konfigurations-Parameters CONFIG_HZ verändert werden.
Bit-Summe	Anzahl der gesetzten Bits

36 Quelle: <http://de.wikipedia.org/wiki/Jiffy>

Appendix E Literaturverzeichnis

- [AIS2031] A proposal for: Functionality classes for random number generators, Version 2.0, 18 September 2011
- [GPR06] Zvi Gutterman, Benny Pinkas, Tzachy Reinmann: "Analysis of the Linux Random Number Generator", IACR Cryptology ePrint Archive, Article No.86, 2006 (<http://eprint.iacr.org/2006/086>).
- [KS11] Wolfgang Killmann, Werner Schindler: "A proposal for: Funtionality classes for random number generators", 2011.
- [LLT07] Kerstin Lemke-Rust, Tobias Lohmann, Wolfgang Thumser: "Dokumentation und Analyse des LINUX Zufallszahlengenerators", 2007
- [LPM10a] Linux Programmer's Manual: "random, urandom - kernel random number source devices", 2010.
- [LPM10b] Linux Programmer's Manual: "poll, ppoll - wait for some event on a file descriptor", 2010.
- [LRSV12] Patrick Lacharme, Andrea Röck, Vincent Strubel, Marion Videau: "The Linux Pseudorandom Number Generator Revisited", IACR Cryptology ePrint Archive, Article No. 251, 2012 (<http://eprint.iacr.org/2012/251>)
- [Mü12] Stephan Müller: "Proper Use of the Linux Kernel Random Number Generator", 2012.
- [P12] Benjamin Pousse: "Short communication: An interpretation of the Linux entropy estimator", IACR Cryptology ePrint Archive, Article No. 487, 2012 (<http://eprint.iacr.org/2012/487>)
- [T06] Theodore Ts'o: "Re: /dev/random on Linux", <http://lkml.org/lkml/2006/5/16/300>, 2006.