Automatisches Erkennen von fehlerhaften Konfigurationseinstellungen

Stefan Sterz - 01426147[a]

a TU Wien, Wien, Österreich

Zusammenfassung

Context Fehlerhafte Konfigurationen sind für einen großen Teil der Fehler in heutigen Software-Systemen verantwortlich. Sie führen nicht nur zu Performance-Einbrüchen und Systemausfällen, sondern stellen auch ein großes Sicherheitsrisiko dar.

Inquiry Eine schnelle und zuverlässige Erkennung von fehlerhaft konfigurierten Programmen und den entsprechenden fehlerhaften Konfigurationstoken, ist daher notwendig. Viele unterschiedliche Ansätze auf dem Gebiet der automatischen Erkennung von Fehlkonfigurationen existieren bereits und bieten jeweils andere Vor- und Nachteile.

Approach Diese Arbeit betrachtet daher zwei unterschiedliche Ansätze auf dem Gebiet der automatischen Erkennung von Fehlkonfigurationen und vergleicht diese. Anschließend wird noch ein Ausblick auf weitere Ansätze auf diesem Gebiet gegeben.

Knowledge In dieser Arbeit beschäftigen wir uns mit zwei Ansätzen im Bereich der automatischen Erkennung von Fehlkonfigurationen. Einerseits werden wir eine Methode betrachten, die eine Informationsflussanalyse verwendet um falsch gesetzte Konfigurationsoptionen zu erkennen. Andererseits werden wir uns einen Ansatz, welcher Methoden des maschinellen Lernens verwendet, näher ansehen.

Grounding Die zwei genannten Ansätze werden anschließend miteinander verglichen, um ihre Vor- und Nachteile gegenüber zu stellen. Es wird gezeigt, dass es schwer ist eine allgemein gültige Lösung für dieses Problem zu finden.

Importance Konfigurationsfehler stellen immer noch eines der größten Probleme im aktiven Betrieb von Software-Systemen dar. Eine manuelle Korrektur dieser ist allerdings sehr zeitaufwändig. Automatisierte Ansätze können viel Zeit sparen, für einen stabileren Betrieb der Systeme sorgen und auch zu Performance-Verbesserungen führen.

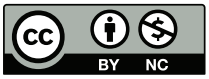
ACM CCS 2012

- **Software and its engineering → System administration;**
- *Computing methodologies → Neural networks;*

The Art, Science, and Engineering of Programming

Perspective The Empirical Science of Programming

Area of Submission Testing and debugging



Submitted to *The Art, Science, and Engineering of Programming*.

1 Einleitung

Heutige Software-Systeme bieten einen hohen Grad an Flexibilität, indem sie viele Konfigurationsoptionen anbieten. Obwohl eine solche Flexibilität im Großen und Ganzen wünschenswert ist, um Systeme vielen unterschiedlichen Umgebungen anzupassen, können dadurch auch Probleme entstehen. Falsche Konfigurationseinstellungen können nicht nur zu Performance-Rückgängen [11], sondern auch zu schweren Sicherheitsrisiken führen. So sieht das Open Web Application Security Project solche Fehler als den sechst häufigsten Grund für Sicherheitsrisiken an [7].

Für Anwenderinnen und Anwender besitzt das Erkennen und Korrigieren von Konfigurationsfehlern einen hohen Stellenwert. Allerdings ist das Finden entsprechender Einstellungen meist mit viel Aufwand verbunden. Die Dokumentation eines Systems und das manuelle Durchsuchen von Online-Foren sind meist die einzigen verfügbaren Informationsquellen, um sie bei dieser Aufgabe zu unterstützen.

Die automatische Erkennung von Konfigurationsoptionen (auch oft Konfigurationstoken genannt), die nicht korrekt gesetzt wurden, kann wesentlich schneller durchgeführt werden als traditionelle Methoden, um Konfigurationen zu korrigieren. Diese Arbeit wird sich daher zuerst mit einem Ansatz von Attariyan und Flinn [1] befassen (siehe Unterabschnitt 2.1) und anschließend werden wir einen aktuelleren Ansatz von Kalibhat et al. [3] betrachten, bei dem maschinelles Lernen verwendet wird (siehe Unterabschnitt 2.2). Im Anschluss werden die zwei Ansätze kurz miteinander vergleichen (siehe Abschnitt 3) und abschließend werden wir noch einen kurzen Ausblick über die Literatur im Allgemeinen in diesem Bereich nehmen (siehe Abschnitt 4). In Abschnitt 5 werden wir die Ergebnisse zum Abschluss noch einmal zusammenfassen.

2 Erkennung von Fehlkonfigurationen

Es gibt einige unterschiedliche Möglichkeiten, um falsche Konfigurationseinstellungen zu erkennen und nicht alle sind im gleichen Kontext anwendbar (siehe Abschnitt 4). In diesem Abschnitt werden wir uns nun mit zwei solchen Systemen beschäftigen, die sich unterschiedlichen Mechanismen, um Fehlkonfigurationen zu erkennen, bedienen.

2.1 Informationsflussanalyse: ConfAid

Ein Ansatz, der von Attariyan und Flinn [1] untersucht wurde, ist die Möglichkeit der Informationsflussanalyse. Dabei wird versucht, unerwünschtes Verhalten eines Software-Systems mit einzelnen Konfigurationsoptionen in Verbindung zu setzen, indem der Fluss von Informationen aus Konfigurationen beim Ausführen des Programms nachverfolgt wird. Ein Ziel des von ihnen erstellten Programms *ConfAid* ist es dabei, nicht nur auf Programme anwendbar zu sein, für welche Quellcode für eine Analyse verfügbar ist, sondern es soll auch direkt Binärdateien untersuchen können.

ConfAid setzt dabei die Arbeit eines früheren Projekts, *AutoBash* [9], fort. *AutoBash* ermöglichte es, einzelne Konfigurationsdateien korrekt als die Ursache von Fehlern

Auflistung 1 Ein Beispiel zur Erklärung von Kausalitätsverfolgung [1]

```
1 if (c == 0) { /* c set to 0 in config file */
2   x = a; /* taken path */
3 } else {
4   y = b; /* alternate path */
5 }
6 z = d;
7 if (z) assert();
```

zu identifizieren, aber es war nicht möglich, einzelne Konfigurationsoptionen als inkorrekt aufzuzeigen. Der Grund dafür lag darin, dass *AutoBash* nur auf Prozessebene arbeitete. Da eine solche Datei aber hunderte Optionen beinhalten kann, war ein weiteres Ziel bei der Entwicklung von *ConfAid* die Verwendung eines höheren Grades an Granularität, um auch einzelne Einstellungsoptionen identifizieren zu können. [1]

Um diese Ziele zu erreichen, instrumentalisiert *ConfAid* eine zu untersuchende Anwendung, indem es eine zusätzliche Ausführungslogik vor und nach den meisten x86-Instruktionen hinzufügt. Danach betrachtet es Systemaufrufe, welche Konfigurationsdateien einlesen und markiert alle Register, Flags und Speicheradressen, welche von diesen Aufrufen verändert werden, als von einer bestimmten Konfigurationsoption abhängig. Dieser Vorgang wird mithilfe des sogenannten “taint tracking” [6] umgesetzt. [1]

Dabei wird eine Variable x mit einer Menge T_x in Verbindung gebracht oder auch markiert, wobei T_x alle Konfigurationstoken sind, von denen der Wert von x potentiell abhängig ist. Eine solche Markierung kann über den Datenfluss[1] und über den Kontrollfluss[1] weitergereicht werden. Beim Ausführen einer Instruktion werden alle Markierungsmengen der betroffenen Register, Speicheradressen und Flags miteinander vereinigt. Bei einer Operation $x = y + z$ gilt also $T_x = T_y \cup T_z$. [1]

Für die Erkennung von Abhängigkeiten über den Kontrollfluss kommt es bei einem naiven Ansatz allerdings zu Problemen. Falls *ConfAid* bei jeder Verzweigung im Kontrollfluss die Markierungen der Variable, welche über den gewählten Zweig entscheidet, hinzufügen würde, würde die Markierungsmenge unbegrenzt anwachsen. In Auflistung 1 würde die Markierungsmenge bei der Zusicherung (Zeile 7) T_c beinhalten, das sollte aber nicht der Fall sein, denn die Variable c hat keine Auswirkung auf diesen Teil des Programms und es würde sich gleich verhalten, unabhängig vom Wert von c. Ein Mechanismus, der gewisse Markierungen wieder entfernt, ist also notwendig. [1]

In Auflistung 1 sollte die Markierungsmenge bei der Zusicherung nur T_d enthalten, denn nur der Wert von z bzw. d entscheidet, ob es zu der Zusicherung kommt. Es gibt hier noch weitere Fälle, die für eine Erkennung über den Kontrollfluss wichtig sind. So betrachtet *ConfAid* auch alternative Zweige, um zu erkennen, ob ein alternativer Zweig fehlerhaftes Verhalten vermeiden würde. *ConfAid* sieht fehlerhaftes Verhalten darin, dass das Programm abbricht oder eine fehlerhafte Ausgabe produziert. Ein alternativer

[1] Während der Kontrollfluss bestimmt in welcher Reihenfolge Befehle ausgeführt werden, bezeichnet der Datenfluss welche Daten an welcher Stelle im Programm vorhanden sind.

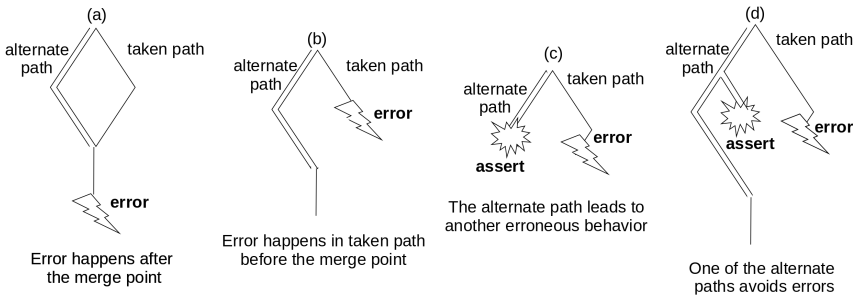


Abbildung 1 Beispiele zur Erklärung der Pfadanalyse durch *ConfAid* [1]

Pfad ist für *ConfAid* dann korrekt, wenn das Programm korrekt beendet oder wenn eine Stelle im Programm erreicht wird, die auch ein fehlerfreies Ausführen erreichen würde. Um eine solche Erkennung zu ermöglichen, wird zuerst mithilfe von IDA Pro [2], welches das Dekompilieren von Programmen erlaubt, eine statische Analyse der Binärdatei durchgeführt. Danach führt *ConfAid* zuerst den alternativen Zweig aus und setzt nach dessen Analyse den Zustand des Programms zurück. Anschließend wird der gewählte Pfad analysiert. [1]

Abbildung 1 zeigt einige Beispiele für die Evaluierung von alternativen Pfaden auf. Im Beispiel (a) tritt der Fehler erst auf, nachdem beide Pfade wieder aufeinander stoßen. Somit geht *ConfAid* davon aus, dass der Kontrollfluss hier keine Auswirkung auf das Fehlverhalten hat, denn es wäre auch aufgetreten, falls der alternative Pfad gewählt worden wäre. Punkte, an denen mehrere Pfade wieder aufeinander treffen, werden ebenfalls durch eine statische Analyse vor dem Ausführen des Programms ermittelt. Im Fall (b) würde der alternative Pfad den Fehler vermeiden. Es wird daher angenommen, dass Token, die Einfluss auf die Entscheidungsvariablen dieser Verzweigung haben, mögliche Gründe für das Fehlverhalten sind. In Beispiel (c) werden Abhängigkeiten, die über den Kontrollfluss entstanden sind, vernachlässigt, denn es kommt hier auch im alternativen Pfad zu einem Fehlverhalten. Fall (d) demonstriert das Verhalten, falls einer von mehreren alternativen Pfaden das Fehlverhalten vermeidet. Hier sind Konfigurationstoken, die die Entscheidung über den gewählten Zweig beeinflussen, ebenfalls in der Markierungsmenge enthalten. Verzweigungen, bei denen Konfigurationstoken keinen Einfluss über die Entscheidung für einen Pfad haben, werden von *ConfAid* ignoriert, denn eine Änderung der Konfiguration hat in solchen Fällen keine Auswirkung über den Programmablauf. [1]

Um die Kontrollflussanalyse zu beschleunigen, benutzt *ConfAid* mehrere Heuristiken. Bei der Betrachtung von alternativen Pfaden werden, dank der “bounded horizon heuristic”, nur eine bestimmte Anzahl von Instruktionen ausgeführt und danach abgebrochen. So soll verhindert werden, dass eine Analyse besonders lange dauert, weil es viele Verzweigungen gibt. Bei einer erneuten Verzweigung in einem alternativen Pfad wird wie folgt vorgegangen: Im Standardfall werden 80 Instruktionen pro alternativem Pfad erlaubt. Hat ein alternativer Pfad also schon 50 Instruktionen ausgeführt und spaltet sich dann wieder, so stehen nun für jeden dieser Unterzweige 30 Instruktionen zur Verfügung. Weiters geht *ConfAid* davon aus, dass nur einige wenige (im Standardfall ein einzelnes) Konfigurationstoken für das Fehlverhalten

verantwortlich sind, um die Kontrollflussanalyse zu erleichtern. Beide Heuristiken können dazu führen, dass gewisse Konfigurationstoken vernachlässigt werden. [1]

Eine Besonderheit des "taint trackings" von *ConfAid* ist, dass nicht nur ein binärer Wert für eine Markierung gespeichert wird, sondern eine Gewichtung, wie stark *ConfAid* vermutet, dass ein Token für das fehlerhafte Verhalten verantwortlich ist. Hierfür werden zwei Heuristiken eingesetzt: Erstens werden Abhängigkeiten, die über den Datenfluss erfasst werden, als wichtiger betrachtet als jene, die über den Kontrollfluss erkannt werden. Denn manche Implementierungen führten hier zu einer hohen Akkumulation von Markierungen. Zum Beispiel würde das Traversieren einer Liste von Konfigurationen, um ein einzelnes Token zu finden, alle anderen vorher traversierten Token hinzufügen. Zweitens werden Abhängigkeiten, die über den Kontrollfluss erkannt werden, als wichtiger eingeschätzt, wenn sie später und damit näher am fehlerhaften Verhalten erfasst werden. [1]

2.2 Maschinelles Lernen

Kalibhat et al. [3] richteten ihre Arbeit darauf aus, ein Hadoop Cluster zu konfigurieren. In einem Hadoop Cluster delegieren Eltern- Nodes gewisse Aufgaben an sie umgebende Arbeiter-Nodes. Kalibhat et al. gingen nun davon aus, dass korrekt konfigurierte Peer-Nodes, die dieselbe Anwendung ausführen, alle eine ähnliche Performance besitzen und daher auch zu korrekten Ergebnissen führen sollten. Nun wurden mehrere maschinelle Lernmodelle gewählt, um zu erkennen, welches von Ihnen Fehlkonfiguration am besten erkennen kann und anschließend wurden ihre Ergebnisse verglichen.

Initial wurde erkannt, dass die Faktoren CPU-Auslastung, Verwendung des dynamischen Speichers und die Anzahl der Threads zwischen korrekt und inkorrekt konfigurierten Systemen signifikante Abweichungen zeigten. Andere verfügbare Faktoren, wie die Anzahl der Java Klassen, hingegen nicht. Des Weiteren wurde erkannt, dass diese Messung der Werte von Faktoren, wie CPU-Auslastung, über eine zeitliche Abfolge ein sich wiederholendes Muster zeigen, falls eine Node richtig konfiguriert worden ist. Ein Konfigurationsfehler drückt sich durch einen initialen Anstieg und ein darauffolgendes Erliegen der Werte aus, weil ein Fehler eingetreten ist. [3]

Da nicht nur erkannt werden sollte, ob eine Node überhaupt ein Konfigurationsproblem hat, sondern auch welche Konfigurationsgruppe und genauer welches Konfigurationstoken für dieses verantwortlich ist, wurden Klassifikationsalgorithmen wie "Support Vector Machine" und "Logistische Regression" gewählt. Des Weiteren wurden "deep- learning"-Modelle wie "Neuronale Netzwerke" trainiert, um nichtlineare Abhängigkeiten in den Daten zu erkennen. Außerdem wurde, um zeitlich abhängige Muster besser zu erkennen, ein "Long Short-Term Memory Recurrent Neural Network" gewählt. [3]

Zum Sammeln der Test- und Trainingsdaten wurde eine automatische Datensammlung entwickelt, die die oben genannten Systemdaten der einzelnen Nodes im korrekt und im inkorrekt konfigurierten Zustand sammelte. Die acht gewählten Fehlkonfigurationen wurden anhand der häufigsten Probleme von Hadoops JIRA Plattform und anderen Forenbeiträgen ausgesucht. Die Anwendungen, die auf dem Hadoop

Cluster ausgeführt wurden, waren die “Big Data Bench”, “HiBench” und “Hadoop Examples” Benchmarks, welche unterschiedliche Aspekte des Clusters testen sollten. Jede Ausführung dauert 50-60 Sekunden und Daten wurden in einem Intervall von einer Sekunde erhoben. Diese Daten wurden anschließend zufällig in einen Trainings- und Testing- Datensatz im Verhältnis 80:20 aufgeteilt. [3]

Anschließend wurden die oben genannten Modelle trainiert, indem alle Daten in einen großen Datensatz kombiniert wurden. Es stellte sich jedoch heraus, dass dieser Ansatz mit steigender Anzahl von Nodes nicht mehr skalierbar war und daher wurden die Modelle für jede Node einzeln trainiert. Weiters wurde festgestellt, dass ein Weglassen des Faktors der Verwendung des dynamischen Speichers zu einer hohen Steigerung an Genauigkeit der Modelle führte und er wurde daher verworfen. [3]

3 Vergleich

Der von Attariyan und Flinn vorgestellte Ansatz erkannte 18 von 18 Konfigurationsproblemen, die von Nutzerinnen und Nutzern von verschiedenen Software-Paketen gemeldet wurden, korrekt. Die entsprechenden Konfigurationstoken wurden jeweils an erster oder zweiter Stelle als für die Probleme verantwortlich aufgelistet. Außerdem wurde *ConfAid* mit zufällig generierten falschen Konfigurationen getestet, wobei ebenfalls 55 von 60 Fehler korrekt erkannt wurden. [1]

Allerdings hat dieser Ansatz auch einige Nachteile. So werden nur Fehlkonfigurationen erkannt, welche auf fehlerhafte Ausgaben, Zusicherungen oder Abstürze zurückführbar sind. Außerdem bedient sich *ConfAid* momentan der Annahme, dass es nur eine Fehlkonfiguration gibt und kann so Zusammenhänge zwischen mehreren Konfigurationstoken nicht gut erkennen. [1]

Auch Kalibhat et al. berichten von sehr guten Ergebnissen. So konnten das “Support Vector Machine”- und das “Recurrent Neural Network”-Modell in 95% der Fälle erkennen, ob sich ein System in einem inkorrekt konfigurierten Zustand befand. Die Art der betroffenen Konfigurationsoptionen konnte in 85% der Fälle korrekt identifiziert werden. Das exakte Konfigurationstoken, hingegen, konnte nur in etwa 60% der Fälle erkannt werden. [3]

Ein Problem mit maschinellem Lernen im Allgemeinen und daher auch mit dem Ansatz von Kalibhat et al. [3], welches unter anderem von Wang et al. [11] aufgezeigt wird, ist, dass diese keine formalen Garantien geben können. So wurde auch von Maggio et al. [5] gezeigt, dass andere Ansätze bessere Ergebnisse erzielen, wenn es um das Erfüllen bestimmter Einschränkungen im Zusammenhang mit dem Verwenden von bestimmten Ressourcen geht. Auch müssten neue Modelle für andere Programme erst erlernt werden, bevor dieser Ansatz auch für diese Programme funktioniert.

4 Verwandte Literatur

Lillack et al. [4] entwickelten *LOTRACK*, ein System welches mit ähnlichen Methoden wie *ConfAid* [1] für Entwickler von Android-Anwendungen aufzeigt, welche Teile

des Codes von welchem Konfigurationstoken beeinflusst werden. Sie fokussieren sich dabei auf Optionen, die beim Start einer Anwendung eingelesen werden.

MisconfDoctor [12] ist ein Werkzeug, welches zuerst eine Datenbank von Fehlkonfigurationen aufbaut, indem es diese absichtlich einfügt und den von einer Anwendung erzeugten Log betrachtet. Anschließend werden potentiell inkorrekt konfigurierte Anwendung mit den in der Datenbank gespeicherten Daten verglichen und eine Wahrscheinlichkeit errechnet, dass es sich dabei um denselben Fehler handelt.

Xu et al. [14] untersuchten Konfigurationsfehler, die sich erst verhältnismäßig spät im Programmablauf zu erkennen geben und entwickelten daraufhin *PCheck*. Es untersucht den Quellcode einer Anwendung und generiert daraufhin Code, welcher eine spätere Ausführung emuliert und so schon früher Fehler erkennen soll.

SmartConf [11] ist ein Framework, welches auf der Kontrolltheorie basiert. Es setzt Konfigurationswerte automatisch und passt diese auch dynamisch an, um bestimmte Performance-Ziele zu erreichen. Das Ziel hier ist also nicht nur die Vermeidung von Fehlkonfigurationen, sondern auch Performance-Optimierung.

Whitaker et al. [13] erstellten *Chronus*, welches eine virtuelle Maschine benutzt, um die Vorgänge in dem System zu rekonstruieren und so festzustellen, wann es zu einer Fehlkonfiguration kam. Eine Anwenderin oder ein Anwender muss dazu eine Sonde bereitstellen, welche es *Chronus* ermöglicht zwischen einem fehlerfreien und einem fehlerhaften Zustand zu unterscheiden.

Wang et al. [10] verwendeten statistische Methoden, um aus einer Gruppe von Maschinen darauf schließen zu können, wie ein korrekt konfiguriertes System aussieht, um so ein inkorrektes zu erkennen. Außerdem versucht ihr Programm, *PeerPressure*, daraufhin auch die zugrunde liegende Konfigurationsoption zu erkennen.

Rabkin und Katz [8] verwendeten eine statische Datenflussanalyse, um von jedem Programmpunkt aus Konfigurationstoken zu erkennen, die an dieser Stelle zu Fehlern führen könnten. Weiters können andere Systemdaten in die Analyse mit einfließen.

5 Konklusion

In dieser Arbeit haben wir zwei unterschiedliche Ansätze zum Erkennen von Fehlkonfigurationen näher betrachtet. Der erste bediente sich einer Informationsflussanalyse, bei der ein Programm daraufhin analysiert wird, wie sich einzelne Konfigurationstoken auf den Programmablauf auswirken. So wird versucht, ein Konfigurationstoken mit Fehlverhalten des Programms in Verbindung zu bringen. Der zweite Ansatz basierte auf maschinellem Lernen, verwendete mehrere Lernalgorithmen und verglich die unterschiedlichen Modelle zum Finden von Konfigurationsfehlern.

Beide Ansätze hatten ihre Vor- und Nachteile und wie eine Literaturrecherche zeigte, gibt es noch viele weitere durchaus erfolgreiche Ansätze auf diesem Gebiet. Daraus lässt sich schließen, dass es nicht die eine Lösung gibt, welche für alle Systeme immer die richtigen Konfigurationsoptionen findet, sondern dass sich Fehlkonfigurationen sehr unterschiedlich ausdrücken können und ihre Erkennung daher auch von Fall zu Fall unterschiedlich sein kann.

Literaturverzeichnis

[1] Mona Attariyan und Jason Flinn. "Automating Configuration Troubleshooting with Dynamic Information Flow Analysis". In: *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*. OSDI'10. Berkeley, CA, USA: USENIX Association, 2010, Seiten 237–250. URL: http://dl.acm.org/citation.cfm?id=1924943.1924960 (besucht am 2018-10-12).

[2] *IDA: About*. URL: https://www.hex-rays.com/products/ida/index.shtml (besucht am 2018-11-18).

[3] N. M. Kalibhat, S. Varshini, C. Kollengode, D. Sitaram und S. Kalambur. "Software Troubleshooting Using Machine Learning". In: *2017 IEEE 24th International Conference on High Performance Computing Workshops (HiPCW)*. Dez. 2018, Seiten 3–10. ISBN: 978-1-5386-1439-6. DOI: 10.1109/HiPCW.2017.00010. URL: doi.ieeecomputersociety.org/10.1109/HiPCW.2017.00010 (besucht am 2018-10-24).

[4] M. Lillack, C. Kästner und E. Bodden. "Tracking Load-time Configuration Options". In: *IEEE Transactions on Software Engineering* (2018), Seiten 1–1. ISSN: 0098-5589. DOI: 10.1109/TSE.2017.2756048.

[5] Martina Maggio, Henry Hoffmann, Alessandro V. Papadopoulos, Jacopo Panerati, Marco D. Santambrogio, Anant Agarwal und Alberto Leva. "Comparison of Decision-Making Strategies for Self-Optimization in Autonomic Computing Systems". In: *ACM Trans. Auton. Adapt. Syst.* 7.4 (Dez. 2012), 36:1–36:32. ISSN: 1556-4665. DOI: 10.1145/2382570.2382572. URL: http://doi.acm.org/10.1145/2382570.2382572 (besucht am 2018-10-31).

[6] James Newsome und Dawn Xiaodong Song. "Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software". In: *Proceedings of the 12th Network and Distributed Systems Security Symposium*. 2005. URL: http://valgrind.org/docs/newsome2005.pdf (besucht am 2018-10-16).

[7] Open Web Application Security Project. *OWASP Top 10 - 2017*. USA: OWASP, 2017. URL: https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.pdf.

[8] Ariel Rabkin und Randy Katz. "Precomputing Possible Configuration Error Diagnoses". In: *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. ASE '11. Washington, DC, USA: IEEE Computer Society, 2011, Seiten 193–202. ISBN: 978-1-4577-1638-6. DOI: 10.1109/ASE.2011.6100053. URL: http://dx.doi.org/10.1109/ASE.2011.6100053 (besucht am 2018-10-31).

[9] Ya-Yunn Su, Mona Attariyan und Jason Flinn. "AutoBash: Improving Configuration Management with Operating System Causality Analysis". In: *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*. SOSP '07. New York, NY, USA: ACM, 2007, Seiten 237–250. ISBN: 978-1-59593-591-5. DOI: 10.1145/1294261.1294284. URL: http://doi.acm.org/10.1145/1294261.1294284 (besucht am 2018-10-16).

[10] Helen J. Wang, John C. Platt, Yu Chen, Ruyun Zhang und Yi-Min Wang. "Automatic Misconfiguration Troubleshooting with PeerPressure". In: *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*. OSDI'04. Berkeley, CA, USA: USENIX Association, 2004, Seiten 17–17. URL: http://dl.acm.org/citation.cfm?id=1251254.1251271 (besucht am 2018-10-31).

[11] Shu Wang, Chi Li, Henry Hoffmann, Shan Lu, William Sentosa und Achmad Imam Kistijantoro. "Understanding and Auto-Adjusting Performance-Sensitive Configurations". In: *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '18. New York, NY, USA: ACM, 2018, Seiten 154–168. ISBN: 978-1-4503-4911-6. DOI: 10.1145/3173162.3173206. URL: http://doi.acm.org/10.1145/3173162.3173206 (besucht am 2018-10-14).

[12] T. Wang, X. Liu, S. Li, X. Liao, W. Li und Q. Liao. "MisconfDoctor: Diagnosing Misconfiguration via Log-Based Configuration Testing". In: *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. 2018 IEEE International Conference on Software Quality, Reliability and Security (QRS). Juli 2018, Seiten 1–12. DOI: 10.1109/QRS.2018.00014.

[13] Andrew Whitaker, Richard S. Cox und Steven D. Gribble. "Configuration Debugging As Search: Finding the Needle in the Haystack". In: *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*. OSDI'04. Berkeley, CA, USA: USENIX Association, 2004, Seiten 6–6. URL: http://dl.acm.org/citation.cfm?id=1251254.1251260 (besucht am 2018-10-31).

[14] Tianyin Xu, Xinxin Jin, Peng Huang, Yuanyuan Zhou, Shan Lu, Long Jin und Shankar Pasupathy. "Early Detection of Configuration Errors to Reduce Failure Damage". In: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. OSDI'16. Berkeley, CA, USA: USENIX Association, 2016, Seiten 619–634. ISBN: 978-1-931971-33-1. URL: http://dl.acm.org/citation.cfm?id=3026877.3026925 (besucht am 2018-10-16).