AI-DevAssist: KI-gestützte sichere Softwareentwicklung

Schlussbericht zum BMBF-Verbundforschungsprojekt im Teilvorhaben

“Der Faktor Mensch in der KI-gestützten sicheren Softwareentwicklung”

von Klaus Tulbure, M.Sc., und Prof. Dr. Matthew Smith, Universität Bonn

Teil I: Kurzbericht

1 Motivation und Ausgangslage

Der Faktor Mensch spielt eine wesentliche Rolle bei der Entwicklung von zuverlässiger und sicherer Software. Da Menschen von Natur aus nicht unfehlbar sind, sind selbst erfahrene Entwickler damit überfordert, vollständig fehlerfreie Software zu erstellen. Da Softwarefehler aber Schwachstellen mit sich bringen können, gilt es, diese möglichst zu vermeiden. Aus diesem Grund wurden über die Zeit zahlreiche Schwachstellenanalysen entwickelt, die die Entwickler bei der Entwicklung unterstützen. Die traditionellen, meist statischen Schwachstellenanalysen sind in der Lage, die gängigsten Fehler verlässlich zu finden und die Funde den Entwicklern mitzuteilen. Sie stoßen allerdings ebenfalls systembedingt an ihre Grenzen, da sich mit ihnen nicht alle Schwachstellen aufdecken lassen. Aus diesem Grund bietet sich Coverage-Guided Fuzzing als neue Methode der letzten Jahre an, um eine größere Menge an Fehlern auffinden zu können.

2 Ziel des Projektes

Bei Fuzzing handelt es sich um eine Methode, die aufgrund ihrer Vorteile für sichere Softwareentwicklung viel Potenzial birgt, jedoch noch insbesondere in ihrer Zugänglichkeit und Bedienbarkeit verbessert werden muss. Ziel dieses Projektes ist es, mithilfe der Durchführung von Nutzerstudien geeignete Lösungsansätze zu entwickeln, die Entwicklern künftig den Einstieg in die Verwendung von Fuzzern erleichtert und somit langfristig die Softwaresicherheit verbessert.

Das Projekt läuft als Teilprojekt “Der Faktor Mensch in der KI-gestützten sicheren Softwareentwicklung” der Universität Bonn unter dem BMBF-Verbundforschungsprojekt “AI-DevAssist: KI-gestützte sichere Softwareentwicklung”. In dem Verbundforschungsprojekt sollen die verschiedenen Fachbereiche Fuzzing, statistische Analysen, KI und Usable Security miteinander kombiniert werden, um Prototypen zu entwickeln, die auf effiziente Arten und Weisen Schwachstellen in Java-Programmen aufdecken können.

Das in diesem Dokument beschriebene Teilprojekt befasst sich hauptsächlich mit dem Anteil der Verbesserung der Benutzerfreundlichkeit von Fuzzing. Um eine Ausgangslage dafür zu schaffen, inwiefern die Benutzerfreundlichkeit und die Bedienbarkeit verbessert werden können, sollten zunächst Nutzerstudien durchgeführt werden. Auf Basis der Studienergebnisse sollten geeignete Lösungsansätze entwickelt werden, die Softwareentwicklern den Zugang und den Umgang mit Fuzzern erleichtern können. Die Lösungsansätze sollten in Form von zusätzlichen statischen Analysen sowie in Form einer grafischen Benutzeroberfläche umgesetzt werden.

3 Anforderungserhebungen durch Nutzerstudien

Mithilfe von Nutzerstudien sollte ermittelt werden, auf welche Probleme die Nutzer*innen bei der Nutzung von Fuzzern stoßen, und welche Anforderung an das Fuzzing diese Probleme beheben könnten. In einer ersten Nutzerstudie wurde die Bedienbarkeit eines Fuzzers mit einer traditionellen, statischen Schwachstellenanalyse verglichen. In einer weiteren Nutzerstudie wurden Entwickler bei der Nutzung von Fuzzern längerfristig begleitet, um ihre Vorgehensweisen und Entscheidungsfindungsprozesse zu protokollieren.

Die Studien zeigten, dass Fuzzing auf eine Reihe technischer und konzeptioneller Limitierungen stößt. Neben den bereits zuvor allgemein bekannten Limitierungen konnten wir mit Nutzerstudien folgende weitere Erkenntnisse nachweisen:

- Die Stellen in der Software, die zum Testen mit dem Fuzzer gut geeignet sind (die sogenannten Angriffspunkte), müssen größtenteils manuell gefunden werden. Für diese müssen manuell Fuzz Targets geschrieben werden, d.h. kleine Codestücke, die definieren, wie der Fuzzer die Software mit den Eingaben ausführen soll. Beides setzt für die Nutzer*innen tiefgreifendes Wissen über die zu testende Software voraus, ohne die die Nutzer*innen nicht bewerten können, wie vielversprechend die Angriffspunkte sind und wie geeignete Fuzz Targets aufgebaut sein müssen.

- Das korrekte Aufsetzen von Fuzzern erweist sich in der Praxis als sehr komplex. Selbst erfahrene Entwickler scheitern daran, da Fuzzer viele Voraussetzungen an das System haben, viele Konfigurationsmöglichkeiten erlauben und nicht immer gut dokumentiert sind.
- Abgesehen von einigen Metriken und etwaiger Funde liefern Fuzzer kaum Informationen an die Nutzer*innen. Die Ausgaben in der Konsole sind sehr spärlich und die erzeugten Log-Dateien nicht ohne weiteres verständlich. Da Fuzzer über eine sehr lange Zeit laufen, wünschen sich Nutzer*innen, mehr Einblicke in den Fortschritt zu erhalten.

Mithilfe der Erkenntnisse aus den Studien wurden zwei Prototypen entwickelt. Der erste Prototyp ist eine Toolchain mit statischen Analysen, die das Finden von Angriffspunkten, das Generieren von Fuzz Targets und eine bessere Abdeckung des Programmcodes ermöglicht. Der zweite Prototyp ist eine grafische Benutzeroberfläche, die die Hürden bei der Bedienung von Fuzzern reduzieren kann.

4 Toolchain als Prototyp zur automatischen Generierung geeigneter Fuzz Targets

Die *Toolchain* kombiniert für das Lösen der oben genannten Probleme die Fähigkeiten von statischen Analysen und Fuzzing. Dabei liefern die statischen Analysen die Angriffspunkte und Fuzz Targets, die dann mit einem Fuzzer getestet werden können. Das statische Schwachstellenerkennungstool *SWAN* des Projektpartners Fraunhofer IEM wird verwendet, um mit Hilfe von Machine Learning die Stellen im Programm zu finden, die sicherheitsrelevant oder anfällig für Schwachstellen sind. In einem iterativen Prozess generieren die beiden statischen Analysen der Universität Bonn, *Coverage Predictor* und *Jazzerator*, gemeinsam die vielversprechendsten Fuzz Targets. Die generierten Fuzz Targets können dann mit dem Java-Fuzzer *Jazzer* des Projektpartners Code Intelligence ausgeführt werden.

Coverage Predictor: Vielversprechende Angriffspunkte finden, Fuzz Targets bewerten
Wir nehmen an, dass ein guter Angriffspunkt möglichst viele Teile und insbesondere sicherheitsrelevante oder fehleranfällige Teile der Software erreichen können muss. Jeder Teil der Software sollte dabei möglichst mindestens einmal von einem Angriffspunkt aus erreichbar sein. Gleichzeitig sollten die Angriffspunkte gut genug verteilt sein, um unnötige Überlappungen zu vermeiden. Wir haben den Coverage Predictor entwickelt, um solch geeignete Kombinationen der vielversprechendsten Angriffspunkte zu finden. Dabei betrachtet der Coverage Predictor sowohl Muster und Strukturen im Kontrollfluss, als auch die fehleranfälligen Stellen, die SWAN identifiziert. Zusätzlich zum Finden von Angriffspunkten kann der Coverage Predictor für bestehende Fuzz Targets abschätzen, wie gut sie die Software abdecken können.

Jazzerator: Fuzz Targets für die Angriffspunkte generieren
Fuzz Targets führen die Angriffspunkte nicht nur aus, sondern müssen auch die vom Fuzzer generierten Eingaben geeignet umwandeln und einen sinnvollen Kontext für den Angriff aufbauen. Dieser Kontext ist nötig, da Angriffspunkte meist mitten im Kontrollfluss des Programms liegen und somit Abhängigkeiten auf andere Programmteile haben. Da Fuzz Targets aber sehr schnell ausführbar sein sollen, müssen diese Abhängigkeiten geeignet reduziert werden. Wir wenden in Jazzerator verschiedene Methoden an, um die Umwandlung der Eingaben und den Aufbau solcher Kontexte automatisch zu generieren und daraus Fuzz Targets zu bauen. Jazzerator sammelt dazu Kontexte aus bestehenden Programmcodes, in denen die Angriffspunkte verwendet werden. Zusätzlich synthetisiert Jazzer anhand von Heuristiken weitere valide Kontexte. Die gesammelten und synthetisierten Kontexte werden dann rekombiniert, um effizientere und gezielte Kontexte aufzubauen.

Erste Auswertungen der Toolchain an Open-Source-Projekten haben die Fähigkeiten der Toolchain bestätigt. Eine ausführliche Auswertung mit einer größeren Anzahl an Projekten steht noch aus.

5 Grafische Benutzeroberfläche als Prototyp für eine nutzerfreundliche Bedienung von Fuzzern

Für Fuzzer gibt es bisher keine weit verbreiteten, benutzerfreundlichen grafischen Benutzeroberflächen. Stattdessen werden Fuzzer größtenteils in der Konsole ausgeführt, in der die Informationen nur sehr vereinfacht, knapp und nicht grafisch aufbereitet ausgegeben werden. Um die Benutzerfreundlichkeit von Fuzzern zu steigern, wurde deshalb *FuzzerView* als Prototyp entwickelt. FuzzerView bietet ein zentrales Dashboard, das Informationen über das zu testende Projekt anzeigt. Das Dashboard erlaubt es, die Tools SWAN, Coverage Predictor, Jazzerator und Jazzer interaktiv auszuführen. Es stellt außerdem alle notwendigen Konfigurationsmöglichkeiten und die Ausgaben der Tools übersichtlich dar. Zudem integriert FuzzerView notwendige Dokumentationen, Erklärungen, Hilfestellungen und Empfehlungen an fast allen Stellen im Dashboard, sodass die Nutzer*innen jederzeit direkten Zugriff auf sie haben. In Nutzerstudien konnte gezeigt werden, dass die Kombination aus grafischer Aufarbeitung und integrierter Dokumentation die Nutzer*innen wesentlich bei der Bedienung und beim Erlernen von Fuzzern unterstützt.

UNIVERSITÄT BONN

Behavioural Security Group

AI-DevAssist: KI-gestützte sichere Softwareentwicklung

Schlussbericht zum BMBF-Verbundforschungsprojekt im Teilvorhaben "Der Faktor Mensch in der KI-gestützten sicheren Softwareentwicklung"

von Klaus Tulbure, M.Sc., und Prof. Dr. Matthew Smith, Universität Bonn

Teil II: Eingehende Darstellung

1 Motivation

Der Faktor Mensch in der Entwicklung sicherer und zuverlässiger Software stellt eine große und bisher ungelöste Herausforderung dar. Die Anzahl der kritischen Schwachstellen steigt trotz aller Versuche, diese einzudämmen.[1] Eine Analyse von Microsoft Research hat zudem gezeigt, dass Entwickler heute größtenteils dieselben Fehlern machen, die bereits vor 20 Jahren gemacht wurden.[2] Es zeigt sich somit klar, dass der Mensch damit überfordert ist, sichere Software zu erstellen, und auch, dass aktuelle Entwicklungswerkzeuge nicht intelligent genug sind, Menschen – insbesondere Softwareentwickler – dabei hinreichend zu unterstützen.

Aus diesem Grund wurden über die Zeit zahlreiche Schwachstellenanalysen entwickelt, die die Entwickler bei der Entwicklung unterstützen. Die traditionellen, meist statischen Schwachstellenanalysen sind in der Lage, die gängigsten Fehler verlässlich zu finden und die Funde den Entwicklern verständlich mitzuteilen. Sie stoßen allerdings ebenfalls systembedingt an ihre Grenzen, da sich mit ihnen nicht alle Schwachstellen aufdecken lassen. Aus diesem Grund bietet sich Coverage-Guided Fuzzing als neue Methode der letzten Jahre an, um eine größere Menge an Fehlern auffinden zu können.

Ziel dieses Projektes ist es, KI-gestützte Methoden zur Schwachstellenerkennung und -verhinderung zu erforschen und Prototypen zu entwickeln, die es Entwicklern ermöglichen, leichter sichere Software zu entwickeln.

2 Ausgangslage zu Beginn des Projekts

Die meisten traditionellen Schwachstellenanalysen sind statisch, sie führen die zu testende Software also nicht aus, sondern identifizieren Fehler und fehleranfällige Stellen anhand von bekannten Mustern und Strukturen im Programmcode. Sie sind generell sehr gut dazu in der Lage, gängige Fehlertypen wie z.B. SQL-Injection oder API-Fehlbenutzungen zu erkennen. Dennoch stoßen rein statische Schwachstellenanalysen systembedingt an ihre Grenzen, sobald das Vorhandensein einer Schwachstelle von Faktoren abhängt, die ohne Ausführen der Software nicht zu bestimmen sind. Dies betrifft insbesondere Abhängigkeiten von Programmeingaben (beispielsweise durch Nutzerschnittstellen oder Eingabedateien) und von Konfigurationsparametern in der Ausführungsumgebung.[3] Statische Analysen müssen bei solchen Abhängigkeiten approximieren, was oft zu unnötigen und falschen Warnungen führt. Eine hohe Anzahl solcher Warnungen ist nachgewiesenermaßen für Entwickler und Tester hochproblematisch, da die falschen Warnungen die Entwickler nicht nur demotivieren, sondern auch von den korrekten und daher wichtigen Warnungen ablenken. Studien haben gezeigt, dass hohe False Positive Rates bei statischen Analysen Entwickler regelmäßig dazu bringen, diese wieder abzuschaffen.[4]

In den letzten Jahren gewinnt deswegen eine Form der dynamischen Analyse namens Fuzzing immer mehr an Bedeutung. Die erste Generation an Fuzzern setzte komplett auf einen randomisierten Ansatz. Die Struktur der zu fuzzenden Programme wurde nicht betrachtet und die Programme wurden daher mit vollkommen zufälligen Daten ausgeführt, um mit diesen Eingaben Programmabstürze herbeizuführen. Diese sehr ineffiziente und wenig intelligente Vorgehensweise ist nicht in der Lage,

[1] Einheitliche Kennung für bekannte Sicherheitslücken. Mehr unter cve.mitre.org.

[2] Matt Miller. Trends, Challenges, and Strategic Shifts in the Software Vulnerability Mitigation Landscape. In BlueHat IL, 2019.

[3] Linghui Luo, Eric Bodden, and Johannes Späth. A qualitative analysis of taint-analysis results. Technical report, Heinz Nixdorf Institute, Paderborn University, August 2018.

[4] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In Proceedings - International Conference on Software Engineering, pages 672–681, 2013.

komplexe Programme in endlicher Zeit zielführend und vollständig zu überprüfen. Das sogenannte Coverage-Guided Fuzzing ist eine signifikante Optimierung des Fuzzing Ansatzes und behebt diese Probleme.[5]

Sowohl in der akademischen Forschung als auch bei den führenden IT-Unternehmen wie Google[6] und Microsoft[7] werden mit diesen intelligenten Fuzzern große Fortschritte erzielt. Dies gilt insbesondere für systemnahe Sprachen wie C/C++ und Assembler. Das Ausführen eines Fuzzers benötigt die manuelle Erstellung sogenannter Fuzz-Targets, die als Einstiegspunkte für den Fuzzer dienen. Der große Nachteil dieses Verfahrens ist, dass das Tooling sehr komplex ist und die dazu notwendigen Fuzzing-Experten auf dem Personalmarkt sehr teuer und überhaupt selten verfügbar sind. Daher ist Fuzzing faktisch für die meisten Unternehmen nicht wirtschaftlich.

2.1 Fuzzing

Fuzzing ist eine teilautomatisierte Testmethode, um Software auf Fehler und somit auch auf Schwachstellen zu überprüfen. Ihr wesentlicher Vorteil liegt darin, dass mit einem Fuzzer die Tests langfristig im Hintergrund ausgeführt werden können und somit weniger Arbeitszeit der Nutzer*innen in Anspruch genommen wird. Den Nutzer*innen bleiben somit als Aufgabe das Aufsetzen und Konfigurieren des Fuzzers sowie das Auswerten der Ergebnisse. Die Laufzeit eines Fuzzers variiert sehr stark im Bereich von Sekunden bis zu Wochen, abhängig davon, ob überhaupt Fehler bestehen und wie schwer sie in der zu testenden Software zu finden sind. Idealerweise führt der Fuzzer dabei hunderte bis tausende Tests pro Sekunde durch.

Das Ziel eines Coverage-Guided Fuzzers ist, mit den Tests möglichst jede Codestelle in der zu testenden Software mindestens einmal auszuführen. Dazu analysiert der Fuzzer zunächst die zu testende Software und sammelt alle ihm verfügbaren Informationen. Mithilfe statischer Analysen und Heuristiken ermittelt er dabei die Struktur und die Datenflüsse innerhalb der zu testenden Software. Anschließend generiert er mittels eines evolutionären Algorithmus möglichst gezielt Eingaben, die potentiell neue Stellen in der Software erreichen können, und führt diese aus. Dies wiederholt der Fuzzer solange, bis eine Eingabe einen Fehler auslöst.

Dynamische Software-Tests werden auch als Angriffe auf die zu testende Software bezeichnet, da sie Fehler in der Software finden, indem sie die Software zum Absturz bringen. Beim Fuzzing werden daher die Stellen in der Software, die zum Testen mit dem Fuzzer gut geeignet sind, als Angriffspunkte bezeichnet. Die eigentlichen Angriffe auf die Software, die sogenannten Fuzz Targets, sind kleine Codestücke, die definieren, wie der Fuzzer die Angriffspunkte mit den Eingaben ausführen soll.

3 Ziel des Projekts

Das Projekt läuft als Teilprojekt “Der Faktor Mensch in der KI-gestützten sicheren Softwareentwicklung” der Universität Bonn unter dem BMBF-Verbundforschungsprojekt “AI-DevAssist: KI-gestützte sichere Softwareentwicklung”. In dem Verbundforschungsprojekt sollen die verschiedenen Fachbereiche Fuzzing, statistische Analysen, KI und Usable Security miteinander kombiniert werden, um Prototypen zu entwickeln, die auf effiziente Arten und Weisen Schwachstellen in Java-Programmen aufdecken können.

Bei Fuzzing handelt es sich um eine Methode, die aufgrund ihrer Vorteile für sichere Softwareentwicklung viel Potenzial birgt, jedoch noch insbesondere in der Zugänglichkeit und Bedienbarkeit verbessert werden muss. Das in diesem Dokument beschriebene Teilprojekt befasst sich in erster Linie mit dem Anteil der Verbesserung der Benutzerfreundlichkeit von Fuzzing. Um eine Ausgangslage dafür zu schaffen, inwiefern die Benutzerfreundlichkeit und die Bedienbarkeit verbessert werden können, sollten zunächst Nutzerstudien durchgeführt werden. Auf Basis der Studienergebnisse sollten geeignete Lösungsansätze entwickelt werden, die Softwareentwicklern den Zugang und den Umgang mit Fuzzern erleichtern können. Die Lösungsansätze sollten in Form von zusätzlichen statischen Analysen sowie in Form einer grafischen Benutzeroberfläche umgesetzt werden.

4 Ergebnisse des Teilprojektes

Nachfolgend werden die Ergebnisse des Teilprojektes anhand der Arbeitsschritte zur Entwicklung der Prototypen veranschaulicht. Diese Struktur entspricht nicht mehr den ursprünglich definierten Arbeitspaketen vom Beginn des Projektes,

---

[5] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. Automated whitebox fuzz testing. In NDSS, volume 8, pages 151–166, 2008.
Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In OSDI, volume 8, pages 209–224, 2008.

[6] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. Taming compiler fuzzers. In ACM SIGPLAN Notices, volume 48, pages 197–208. ACM, 2013.

[7] Patrice Godefroid, Michael Y Levin, and David Molnar. Sage: whitebox fuzzing for security testing. Communications of the ACM, 55(3): pages 40–44, 2012.

da sich über den Projektverlauf hinweg ergeben hat, dass sich die ursprünglich definierten Arbeitspakete teils überschneiden, mehr Berührungspunkte zueinander besitzen oder teils einen größeren Umfang haben als antizipiert.

4.1 Anforderungserhebungen durch Nutzerstudien (AP 3, AP 6.1)

Das Ziel der Nutzerstudien war es, ein umfangreiches Bild über die menschliche Anwendung von Schwachstellenanalysen zu erhalten. Diese sollten aufgearbeitet werden, um etwaige Lösungen in den Demonstrator einzubauen. Wichtige Kriterien waren dabei: allgemeine Probleme; Unterschiede bei der Nutzung verschiedener Schwachstellenanalysen; Denkweisen und Arbeitsschritte bei der Nutzung von Fuzzern; Unterschiede zwischen und innerhalb von Nutzergruppen, insbesondere zwischen erfahrenen und unerfahrenen Nutzer*innen; sowie Wünsche der Nutzer*innen an den Fuzzer.

In der ersten Studie wurde die Benutzerfreundlichkeit der statischen Schwachstellenanalyse Clang Static Analyzer mit dem Fuzzer libFuzzer verglichen. Sie sollte aufklären, wieso statische Analysen allgemein genutzt wurden, während Fuzzing von vielen gemieden wurde. Die Studie konnte bestätigen, dass die Nutzerfreundlichkeit von Fuzzern deutlich schlechter war als die von statischen Analysen. Denn während die meisten Studienteilnehmer*innen den Clang Static Analyzer ohne nennenswerte Probleme ausführen konnten, sind fast alle Studienteilnehmer*innen bei der Nutzung von libFuzzer gescheitert. Sie hatten insbesondere Probleme, geeignete Angriffspunkte auszuwählen, Fuzz Targets zu schreiben und zu kompilieren, sowie die Ausgaben und Ergebnisse des Fuzzers zu verstehen. Aus der Studie wurde geschlussfolgert, dass eine ausführlichere Dokumentation, ein Easy-Mode als Einstiegshilfe, Live-Informationen über die Coverage während des Fuzzens, sowie eine ausführlichere Ausgabe der Ergebnisse zur Lösung der beobachteten Probleme beitragen können.

In einer weiteren Studie wurden Entwickler bei der Einrichtung und der langfristigen Nutzung von Jazzer begleitet. Die Studie hat Herausforderungen, Lösungsansätze und Fortschritte der Studienteilnehmer*innen protokolliert, um die notwendigen Vorgehensweisen und Entscheidungsfindungsprozesse bei der Bedienung von Jazzer aufzuzeigen. In dieser Studie konnten zunächst die gleichen Probleme und Wünsche wie in der ersten Studie festgestellt werden. Zudem hat sich gezeigt, dass die steile Lernkurve von Fuzzing dazu führt, dass vom ursprünglichen Arbeitsplan abgewichen wird. Zum einen werden wesentliche Schritte übersprungen, zum anderen werden komplexe Herausforderungen angegangen noch bevor das notwendige Wissen vollständig angeeignet wurde. Außerdem scheitern die Nutzer*innen an vielen Fehlversuchen, wobei in einigen Fällen nebensächliche Probleme zum Scheitern führen. Die zweite Studie zeigte zudem, dass die zwischenzeitlich verbesserte Dokumentation von Jazzer zu einer deutlich besseren Verständlichkeit geführt hat.

4.2 Coverage Predictor: Geeignete Angriffspunkte finden, Fuzz Targets bewerten (AP 1, AP 5.2)

Das Ziel von AP1 war für das Verbundprojekt AI-DevAssist eine Benchmark zur Bewertung von Fuzzern zu entwickeln. In diesem Rahmen wurde entschieden, den Fokus auf die Bewertung von Fuzz Targets zu richten. Dies hatte folgende Gründe:

- Fuzz Targets laufen für gewöhnlich sehr lange, wobei teilweise erst nach längerer Zeit ersichtlich wird, wie gut das Fuzz Target eigentlich ist.
- Für größere Projekte sind mehrere Fuzz Targets erforderlich, um alle Teile der Software abdecken zu können. Gleichzeitig möchte man allerdings Redundanzen zwischen den Fuzz Targets verhindern, beispielsweise wenn zwei Fuzz Targets nur dieselben Teile der Software abdecken.
- Mit der Coverage alleine lässt sich nicht ermitteln, wie gut ein Fuzz Target ist. Beispielsweise kann ein Fuzz Target zwar eine hohe Coverage im Kontrollfluss erreichen, den großen abgedeckten Bereich jedoch mit einer unzureichenden Anzahl an Eingaben testen und somit Fehler übersehen. Andersherum kann ein Fuzz Target eine niedrige Coverage im Kontrollfluss haben, den kleinen abgedeckten Bereich jedoch mit einer hohen Anzahl an Eingaben ausgiebig testen.
- Es gibt verschiedene Arten, wie man Code Coverage umsetzen kann, unter anderem auf Source-Code-Basis oder auf Binärcode-Basis, sowie in verschiedenen Detailstufen. Dabei stellte sich die Frage, ob mittels der verschiedenen Coverage-Arten unterschiedliche Schlüsse über die Qualität eines Fuzz Targets gezogen werden können.

Ausgehend davon wurde in diesem Arbeitspaket der *Coverage Predictor* entwickelt, der den Programmcode statisch analysiert und die erreichbare Coverage schätzt. Die Abschätzung hat eine untere Schranke ("wird sehr wahrscheinlich erreicht") und eine obere Schranke ("kann prinzipiell erreicht werden"). Dabei bietet der Coverage Predictor Lösungsansätze für die oben genannten Probleme:

- Die statischen Analysen laufen im Minutenbereich, sind also sehr schnell im Vergleich zum Fuzzen.
- Für Projekte mit mehreren Fuzz Targets wird ermittelt, inwiefern sich die abgedeckten Bereiche überlappen.
- Es kann im einzelnen betrachtet werden, ob ein Fuzz Target alleine zwar eine geringe Coverage erzielt, im Zusammenhang mit allen Fuzz Targets des Projekts aber wichtige Lücken schließt.

- Weiterhin werden sechs Coverage-Arten abgeschätzt: Instruction Coverage, Basic Block Coverage, Basic Block Edge Coverage, Method Coverage, Source Code Line Coverage und Security Relevant Method Coverage. Die ersten fünf sind allgemein bekannt, während die Security Relevant Method Coverage in diesem Arbeitspaket neu erfunden wurde. Diese Coverage-Art kann zum Beispiel mittels SWAN bestimmt werden.

Generell hat sich gezeigt, dass der Coverage Predictor zwar für eine Benchmark geeignet ist, das volle Potential aber erst in Kombination mit der Fuzz-Target-Generierung aus AP 5.2 entfaltet. Aus diesem Grund kann die finale Version des Coverage Predictors auch alle potenziellen Angriffspunkte des Programms bewerten. Somit kann der Coverage Predictor abschätzen, wie hoch die Coverage sein wird, wenn ein Angriffspunkt ausgewählt wird. Dadurch kann eine Reihenfolge der vielversprechendsten Angriffspunkte bestimmt werden. Jedem Angriffspunkt wird zudem ein Coverage Potential zugewiesen, das abschätzt, wie sich die gesamte Coverage Prediction durch das Abdecken des Angriffspunktes voraussichtlich verändern wird. Das Coverage Potential ist somit eine Art Differenz von Coverage Predictions. Die Kombination mit den anderen Komponenten wird in den Abschnitten 4.4 und 4.5 weiter beschrieben.

4.3 Jazzerator: Fuzz Targets für die Angriffspunkte generieren (AP 5.2)

Ziel war es, den manuellen Aufwand für das Schreiben von Java Fuzz Targets zu automatisieren. Dies wurde in der Vergangenheit bereits für andere Programmiersprachen realisiert, weshalb bereits vorhandene Konzepte teilweise für Java adaptiert werden konnten.

Fuzz Targets führen die Angriffspunkte nicht nur aus, sondern müssen auch die vom Fuzzer generierten Eingaben geeignet umwandeln und einen sinnvollen Kontext für den Angriff aufbauen. Dieser Kontext ist nötig, da Angriffspunkte meist mitten im Kontrollfluss des Programms liegen und somit Abhängigkeiten auf andere Programmteile haben. Da Fuzz Targets aber sehr schnell ausführbar sein sollen, müssen diese Abhängigkeiten geeignet reduziert werden. Jazzerator wendet mehrere Methoden an, um die Umwandlung der Eingaben und den Aufbau solcher Kontexte automatisch zu generieren und daraus Fuzz Targets zu bauen. Dazu sammelt Jazzerator zunächst Programmsequenzen aus bestehender Software, in der die Angriffspunkte verwendet werden, und synthetisiert zusätzlich anhand der vorhandenen Typ-Informationen weitere Programmsequenzen. Die gesammelten und synthetisierten Programmsequenzen werden dann iterativ kombiniert, um valide und effiziente Kontexte aufzubauen. Die Kontexte werden dann zu Fuzz Targets vervollständigt. Durch das Kombinieren entstehen somit für jeden Angriffspunkt mehrere Fuzz Targets, die sich in Inhalt und Komplexität unterscheiden.

4.4 Grafische Benutzeroberfläche als Prototyp für eine nutzerfreundliche Bedienung von Fuzzern (AP 5.1, AP 6.2, AP 6.3, AP 6.4, AP 7)

Das Ziel von diesem Arbeitspaket war es, für das Verbundprojekt eine zugängliche Benutzeroberfläche zu schaffen. Bisher besitzen die Softwareanalysen eher einen Black-Box-Charakter, da diese größtenteils in einer Konsole ausgeführt werden, wo die Informationen nur sehr vereinfacht, knapp und nicht grafisch aufbereitet ausgegeben werden. Durch das fehlende Feedback und Hilfestellungen haben die Nutzer*innen nur eine eingeschränkte Übersicht über die Prozesse, können sich deswegen nur schwer orientieren.

Somit wurde in diesem Arbeitspaket die Benutzeroberfläche *FuzzerView* entwickelt, die ein zentrales Dashboard für die im Verbundprojekt entwickelten Tools bildet. Unterstützt werden dabei die Tools Jazzer, SWAN, der Coverage Predictor, Jazzerator und der Java Compiler. FuzzerView handhabt die Einrichtung des zu testenden Projekts, das Anzeigen der Projektstruktur, das Ausführen der Tools, das Anzeigen der Ergebnisse sowie die Dokumentation der einzelnen Schritte.

Für die Entwicklung der Benutzeroberfläche sind die Erkenntnisse aus den Nutzerstudien aus Abschnitt 4.1 mit eingeflossen. FuzzerView kombiniert die Informationen der Tools mit passenden Interaktionsmöglichkeiten und dazugehöriger Dokumentation für die Nutzer*innen. Dafür werden die Informationen auf verschiedene Arten und Weisen aufbereitet und dargestellt, um die Zusammenhänge zwischen den Daten aufzuzeigen. Insgesamt gibt es dafür eine Vielzahl an Ansichten, die der Nutzer, je nachdem welche Aktionen er ausführen möchte, entsprechend auswählen kann.

Im Folgenden werden beispielhaft vier dieser Ansichten beschrieben:

- Beispiel 1 (siehe Abbildung 1): Eine tabellarische Übersicht zeigt die Sicherheitsrelevanzen und die Coverage Predictions bzw. Potentials zu den Libraries des Projekts, zu den Fuzz Targets und zu den Angriffspunkten an, und unterstützt das Sortieren und Filtern dieser.
- Beispiel 2 (siehe Abbildung 2): Die Inhalte der tabellarischen Übersicht werden auch in einem Diagramm, der *SoftwareTreeView,* visuell aufgearbeitet. Die SoftwareTreeView wurde in diesem Arbeitspaket konzipiert, um klassifizierende Informationen wie die Coverage Informationen und die Sicherheitsrelevanzen für eine gesamte

Software in einer kompakten Ansicht anzuzeigen. Die Struktur der SoftwareTreeView gliedert sich dabei hierarchisch nach den Paketen, Klassen und Methoden der Software. Die Klassifizierungen werden dabei durch Farben und Symbole markiert.

- Beispiel 3 (siehe Abbildung 3): Es gibt eine Ansicht, die für eine Methode als Angriffspunkt ihre Sicherheitsrelevanz, ihr Coverage Potential sowie geeignete Fuzz Targets anzeigt. Gleichzeitig gibt es einen Button zum direkten Ausführen von Jazzerator für die Methode als Angriffspunkt.
- Beispiel 4 (siehe Abbildung 4): Der Ausgaben von Jazzer an die Konsole werden grafisch aufgearbeitet. Die Ansicht zeigt den aktuellen Zustand von Jazzer sowie den Fortschritt der erreichten Coverage an.

FuzzerView unterstützt zudem eine interaktive Einrichtung des Projekts, eine Übersicht der im hintergrund laufenden Ausführungen der Tools, weitere Live-Ansichten mit den Fortschritten der Tools, eine optionale Schnellanleitung, Tooltips Informationen für alle Informationen und an allen Interaktionspunkten, detaillierte Erklärungen, ein Lexikon mit den relevanten Begriffen, sowie einen Editor zum Bearbeiten der Fuzz Targets.

Abschließend wurde zur Evaluierung von FuzzerView eine Nutzerstudie durchgeführt. Dabei ist FuzzerView gegen ChatGPT 3.5 angetreten, um den manuellen Teil der Arbeitsabläufe zu bewerten. Den Nutzer*innen wurde deshalb FuzzerView ohne Jazzerator bereitgestellt. Die Aufgabe für die Studienteilnehmer*innen war die Erstellung und die erfolgreiche Ausführung von Fuzz Targets. Dabei konnten folgende Resultate festgestellt werden:

- FuzzerView bietet Nutzer*innen auch bei größeren Projekten einen Vorteil in der Bedienbarkeit.
- Die Coverage Prediction bietet eine hilfreiche EInsicht in die Qualität der Fuzz Targets.
- Beim Aussuchen geeigneter Angriffspunkte sind die Informationen von SWAN und vom Coverage Predictor hilfreich.
- Das manuelle Schreiben von Fuzz Targets inklusive der Einarbeitung dauert im Schnitt 34 MInuten.
- Es gibt noch Verbesserungspotenzial für FuzzerView, indem bestimmte Arbeitsabläufe vereinfacht bzw. für die Nutzer*innen hervorgehoben werden.

4.5 Toolchain als Prototyp zur automatischen Generierung von Fuzz Targets (AP 5.2, AP 7)

Bei der Auswertung von FuzzerView sind einige Arbeitsabläufe immer wiederholt vorgekommen. Insbesondere hat sich dabei gezeigt, dass sich der Coverage Predictor und Jazzerator passend ergänzen, dass Jazzerator ursprünglich ineffizient genutzt wurde, und dass SWAN mit der Security Relevant Method Coverage ebenfalls helfen kann. Daher ist in diesem Arbeitspaket die *Toolchain* entstanden, die diese drei Komponenten in einem automatisierten Ablauf bündelt und als Ergebnis geeignete Fuzz Targets für Jazzer liefert. Die Toolchain erfüllt zum Teil die gleichen Aufgaben wie FuzzerView, automatisiert aber viele manuelle Schritte der Nutzer*innen.

Ursprünglich hat Jazzerator Fuzz Targets für alle potenziellen Angriffspunkte auf einmal generiert. Dies hat sich als ineffizient und zu langwierig herausgestellt, da bei einigen Projekten bis zu mehrere hunderttausend Fuzz Target erzeugt wurden. Zudem wurden unnötigerweise selbst für offensichtlich ungeeignete Angriffspunkte Fuzz Targets generiert. Für die getesteten Projekte waren eher eine ein- bis zweistellige Anzahl an Fuzz Targets angemessen. Aus diesem Grund sollten die Fuzz Targets nicht mehr für alle Angriffspunkte auf einmal generiert werden, sondern iterativ für die jeweiligen vielversprechendsten Angriffspunkte. Dafür wurde der Coverage Predictor, wie in Abschnitt 4.2 beschrieben, erweitert, um die vielversprechendsten Angriffspunkte entsprechend zu benennen.

Auf Basis der Erkenntnisse aus Abschnitt 4.5 wurden die zwei Coverage Arten Security Relevant Method Coverage (SRM Coverage) und Basic Block Edge Coverage (Edge Coverage) miteinander kombiniert. Diese beiden Arten haben sich als am geeignetsten herausgestellt und wurden somit in der Toolchain eingesetzt. Dabei zielt die Toolchain zunächst auf eine Maximierung der oberen Schranke ab. Sobald diese ausreichend hoch ist, wird auch die Maximierung der unteren Schranke angestrebt.

Die Toolchain hat somit den folgenden Ablauf erhalten. Zunächst werden mit SWAN die sicherheitsrelevanten Methoden bestimmt. Diese werden an den Coverage Predictor übergeben, um damit die SRM Coverage bestimmen zu können. Der Coverage Predictor ermittelt nun anhand der SRM Coverage und der Edge Coverage den vielversprechendsten Angriffspunkt, für den dann mit Jazzerator Fuzz Targets generiert werden. Durch das Kompilieren der Fuzz Targets kann festgestellt werden, ob sie läufig sind. Der Coverage Predictor wird nun erneut benutzt, diesmal um von den lauffähigen Fuzz Targets das vielversprechendste auszuwählen. Dieses Fuzz Target wird dem Ergebnis hinzugefügt. Die Auswahl- und Generierungsschritte werden nun wiederholt, wobei der Coverage Predictor bei der Auswahl des vielversprechendsten Angriffspunkt das bisherige Ergebnis berücksichtigt. Die Generierung endet, sobald eine eingestellte Coverage oder eine eingestellte Anzahl an Fuzz Targets erreicht wird, beziehungsweise sobald keine neuen Fuzz Targets mehr generiert werden können. Das Ergebnis kann nun von Nutzer*innen eingesehen werden, optional angepasst werden und dann in Jazzer ausgeführt werden.

4.6 Erkennung von Race Conditions in Java (AP 5.3)

Multithreading ist eine bevorzugte Technik in Software, da sie erlaubt, Programmteile gleichzeitig auszuführen. Allerdings benötigt sie meist eine manuelle Synchronisierung. Dieser Schritt ist sehr komplex und ist somit sehr anfällig für neun Schwachstellenarten, u.a. Race Conditions. Auch gängige Fuzzer stießen bei Multithreading-Projekten an ihre Grenzen, da die Analyse der parallel laufenden Programmteile sehr komplex und nicht deterministisch ist. Ziel war deshalb die Erweiterung Jacozer für Jazzer, die diese Schwachstellenarten erkennen sollte.

In Form einer Benchmark wurde zu allen neun Schwachstellenarten jeweils ein minimales repräsentatives Projekt entwickelt. Jedes dieser Projekte enthält eine Schwachstelle der jeweiligen Art, so dass die Schwachstelle mit einer hohen Wahrscheinlichkeit vom Fuzzer erreicht werden kann. Die Benchmark sollte genutzt werden, um Jazzer mit und ohne Jacozer zu vergleichen. Bei der Ausführung ohne Jacozer hat Jazzer die fehlerhaften Codestellen zwar abgedeckt, die Fehler aber wie zu erwarten nicht erkannt.

Die Entwicklung von Jacozer musste allerdings aufgrund von technischen Hindernissen eingestellt werden. Es hat sich erst nach einiger Zeit herausgestellt, dass Jazzer und die notwendige Library JEP draft nicht miteinander kompatibel sind. Beide starten eine eigene Java Virtual Machine (JVM), die unabhängig voneinander laufen. Eine Zusammenführung der beiden JVMs war nicht möglich. Auch war es ohne unverhältnismäßig hohem Aufwand nicht möglich, die Instrumentierung von JEP draft in Jazzer direkt zu integrieren. Die Ergebnisse von Jazzer in der Benchmark zeigen allerdings, dass die Lösung dieses Problems in der Zukunft fortgesetzt werden sollte.

4.7 Schnittstellen zwischen Komponenten des Verbundprojekts (AP 2, AP 4, AP 5.1)

Innerhalb des Verbundprojektes AI-DevAssist fließen die Informationen vieler einzelnen Komponenten zusammen. Die Struktur des Verbundprojektes und das Zusammenspiel der Komponenten wurde deshalb anschaulich festgehalten und regelmäßig aktualisiert. Somit konnten alle Projektpartner jederzeit einen Überblick über die Struktur behalten.

5 Ergänzende Informationen

5.1 Die wichtigste Positionen des zahlenmäßigen Nachweises

Die finanziellen Mittel wurden für die Personalkosten der für das Projekt eingestellten Wissenschaftler eingesetzt.

5.2 Die Notwendigkeit und Angemessenheit der geleisteten Projektarbeiten

Alle oben beschriebenen Komponenten haben dazu beigetragen, das Ziel dieses Teilprojekts, die Benutzerfreundlichkeit von Fuzzing in Kombination mit statischen Analysen und Künstlicher Intelligenz, zu erreichen. Als Ergebnis entstanden Prototypen, die die einhergehende Problemstellung in einem angemessenen sowie erfolgreichen Maße lösten. Auch die Zusammenarbeit mit den Projektpartnern und ihren Komponenten haben maßgeblich zum Erfolg des Projektes beigetragen.

5.3 Der voraussichtliche Nutzen und Planungen für die nähere Zukunft

Der Coverage Predictor, Jazzerator, FuzzerView und die Toolchain sollen als Open-Source-Projekt frei verfügbar gemacht werden. Sie sollen dadurch jedem frei zugänglich werden und sich für die in Abschnitt 4 beschriebenen Zwecke einsetzen lassen. Dies erlaubt zudem eine Fortsetzung der Projekte nicht nur durch die am Projekt beteiligten, sondern auch durch Dritte.

5.4 Fortschritt von Dritten auf dem Gebiet des Vorhabens

Das umfangreiche Forschungsfeld zum Fuzzing hat während der Durchführung des Vorhabens zahlreiche neue Fortschritte gemacht. Es sind jedoch keine neuen Ergebnisse von dritter Seite bekannt, die thematisch im direkten Zusammenhang oder in direkter Konkurrenz zu den oben beschriebenen Komponenten stehen.

5.5 Die erfolgten oder geplanten Veröffentlichungen

Es stehen noch wissenschaftliche Publikationen für die Toolchain und für die FuzzerView aus.

Anhang

COVERAGE PREDICTION FOR LIBRARIES

Name	Methods			Basic block edges			Basic blocks			Instructions			Lines		
	lower	upper	total	lower	upper	total	lower	upper	total	lower	upper	total	lower	upper	total
gson-2.9.1.jar	70	359	803	1277	5176	8452	729	2719	4895	2587	8960	17114	635	2060	3858
Combined	70	359	803	1277	5176	8452	729	2719	4895	2587	8960	17114	635	2060	3858

COVERAGE PREDICTION FOR FUZZ TARGETS

Name	Methods			Basic block edges			Basic blocks			Instructions			Lines		
	lower	upper	total	lower	upper	total	lower	upper	total	lower	upper	total	lower	upper	total
FuzzParse	31	351	803	860	5099	8452	459	2690	4895	1521	8892	17114	373	2033	3858
FuzzReader	54	343	803	816	4915	8452	544	2544	4895	2324	8471	17114	563	1950	3858
FuzzStreamParser	28	352	803	762	4956	8452	428	2631	4895	1503	8834	17114	369	2022	3858
Combined	70	359	803	1277	5176	8452	729	2719	4895	2587	8960	17114	635	2060	3858

METHODS with SECURITY-RELEVANCE and COVERAGE POTENTIAL

Name	Security-Relevance			Coverage Potential									
Search	source	sanit.	# CWEs	Methods		Basic block edges		Basic blocks		Instructions		Lines	
	sink	auth.		lower	upper	lower	upper	lower	upper	lower	upper	lower	upper
DateTypeAdapter.read(JsonReader)				13	13	2374	2374	693	693	1164	1164	242	242
DefaultDateTypeAdapter.read(JsonReader)				14	14	2374	2374	693	693	1169	1169	243	243
DateTypeAdapter.deserializeToDate(JsonReader)				10	10	2259	2259	630	630	1048	1048	220	220
DefaultDateTypeAdapter.deserializeToDate(JsonRead...				10	10	2259	2259	630	630	1048	1048	220	220
Gson.fromJson(JsonReader,Type)				2	3	331	662	111	221	89	173	24	47
UnsafeAllocator.create()				1	2	266	532	137	274	107	214	22	44
TypeAdapters$EnumTypeAdapter(Class)				3	4	245	490	123	246	111	222	20	40
ReflectiveTypeAdapterFactory.create(Gson,TypeToken)				63	63	476	476	389	389	1645	1645	374	374
ReflectiveTypeAdapterFactory.getBoundFields(Gson,T...				61	61	426	426	353	353	1458	1458	335	335
Gson.fromJson(String,Class)				8	9	397	397	154	155	212	222	56	58

Abbildung 1: Tabellarische Übersichten zu den Sicherheitsrelevanzen und Coverage Predictions bzw. Potentials pro Projektdatei (oben), Fuzz Target (mitte) und Methode (unten).

Die Balken zeigen die folgenden Anteile farblich an:

- grün (= untere Schranke): wird sehr wahrscheinlich gecovert
- blau (= obere Schranke - untere Schranken): kann prinzipiell gecovert werden
- rot (= alle - obere Schranke): wird wahrscheinlich nicht gecovert

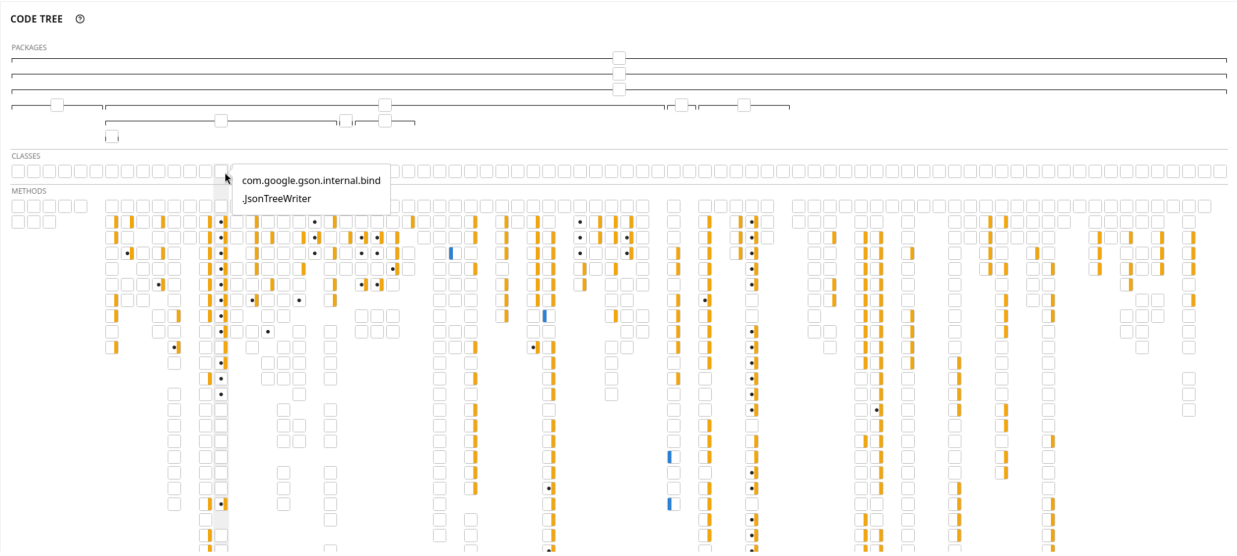


Abbildung 2: Ausschnitt des Code Trees mit Paketen, Klassen und Methoden, hier nach den Sicherheitsrelevanzen farblich klassifiziert

SELECTED METHOD

Package	com.google.gson.internal.bind
Class	ReflectiveTypeAdapterFactory
Name	create
Parameters	com.google.gson.Gson com.google.gson.reflect.TypeToken

Security Relevance

Categories	-
CWEs	-

Coverage Potential

Type	lower	upper	total
Methods	115	115	412
Basic block edges	696	696	1750
Basic blocks	588	588	1409
Instructions	2614	2614	5606
Source code lines	573	573	1236

Coverage Prediction

will likely be covered by	-
could be covered by	Gson_toJson_Target3453 JsonParser_parseReader_
suitable entry points to fuzz this method	

Generate Fuzz Target

Open in Intellij IDEA

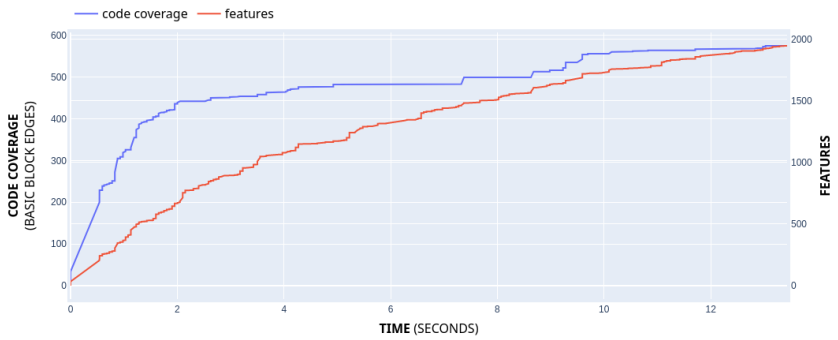
Abbildung 3: Rechte Informationsleiste mit Details und Interaktionsmöglichkeiten für die ausgewählte Methode

00:13	197 389	575	1 947	468	17 KB	14 696
elapsed time	iterations	code coverage	features	corpus entries	corpus size	executions per second

CONFIGURATION

Fuzz target: FuzzParse.java

CODE AND FEATURE COVERAGE OVER TIME



CRASH INFO

no crash found

Abbildung 4: Detailansicht eines einzelnen Jazzer-Prozesses während der Ausführung