



Delft University of Technology

Document Version

Final published version

Citation (APA)

Cao, C. S. (2026). *Modeling Behavior Patterns in Microservice Applications: Practical Applications Using Finite State Machines*. [Dissertation (TU Delft), Delft University of Technology]. Delft University of Technology.
<https://doi.org/10.4233/uuid:60b7d580-e9cb-4874-b67d-e9987f028f49>

Important note

To cite this publication, please use the final published version (if applicable).
Please check the document version above.

Copyright

In case the licence states "Dutch Copyright Act (Article 25fa)", this publication was made available Green Open Access via the TU Delft Institutional Repository pursuant to Dutch Copyright Act (Article 25fa, the Taverne amendment). This provision does not affect copyright ownership.
Unless copyright is transferred by contract or statute, it remains with the copyright holder.

Sharing and reuse

Other than for strictly personal use, it is not permitted to download, forward or distribute the text or part of it, without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license such as Creative Commons.

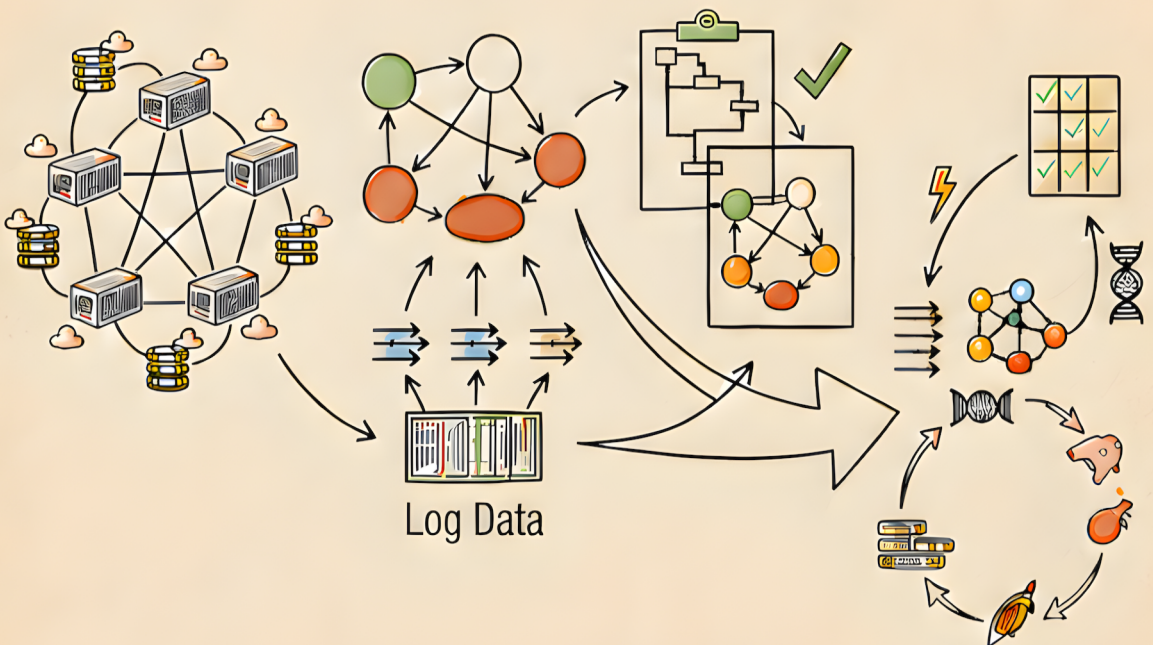
Takedown policy

Please contact us and provide details if you believe this document breaches copyrights.
We will remove access to the work immediately and investigate your claim.

This work is downloaded from Delft University of Technology.

MODELING BEHAVIOR PATTERNS IN MICROSERVICE APPLICATIONS

PRACTICAL APPLICATIONS USING FINITE STATE MACHINES



CLINTON CAO
DELFT UNIVERSITY OF TECHNOLOGY

Modeling Behavior Patterns in Microservice Applications

Practical Applications Using Finite State Machines

Modeling Behavior Patterns in Microservice Applications

Practical Applications Using Finite State Machines

Dissertation

for the purpose of obtaining the degree of doctor
at Delft University of Technology
by the authority of the Rector Magnificus,
Prof.dr.ir. H. Bijl,
chair of the Board for Doctorates,
to be defended publicly on
Friday, 19 June 2026 at 10:00

by

Clinton Shen CAO

This dissertation has been approved by the promotor.

Composition of the doctoral committee:

Rector Magnificus,	chairperson
Dr. ir. S. E. Verwer,	Delft University of Technology, promotor
Dr. A. Panichella,	Delft University of Technology, promotor

Independent Members:

Prof. dr. P. A. N. Bosman,	Delft University of Technology
Dr. X. Devroey,	University of Namur, Belgium
Dr. G. Gay,	Chalmers University of Technology, Sweden
	University of Gothenburg, Sweden
Dr. ir. M. J. G. Olsthoorn,	Delft University of Technology
Dr. E. Poll,	Radboud University, the Netherlands
Prof. dr. M. T. J. Spaan,	Delft University of Technology, reserve member

SIKS Dissertation Series No. 2026-35.

The research reported in this thesis has been carried out under the auspices of SIKS, the Dutch Research School for Information and Knowledge Systems, and under the European Project AssureMOSS (The Assurance and certification in secure Multi-party Open Software and Services).



Keywords: Microservice Applications, Behavior Modeling, Cyber Security, Software Testing, Software Validation, Run-time Analysis

Printed by: Proefschriftspecialist

Cover: Clinton Cao (with help from ChatGPT and Gemini)

Copyright © 2026 by Clinton Cao
ISBN 978-94-6518-368-8

An electronic version of this dissertation is available at <http://repository.tudelft.nl/>.

To my beloved parents.

Contents

Summary	xi
Samenvatting	xiii
Acknowledgments	xv
1 Introduction	1
1.1 Background & Context	3
1.1.1 Microservices	3
1.1.2 Deterministic Finite State Machines	3
1.1.3 AssureMOSS Project	6
1.2 Research Goals & Questions	6
1.3 Contributions	7
1.4 Contribution to Open Science	10
1.5 Research Outline	10
1.5.1 Part I: Monitoring Deployment for Anomalies	10
1.5.2 Part II: Validating Deployment Against Implementation	11
1.5.3 Part III: Automated Testing with Model Inference	11
I Monitoring Deployment for Anomalies	15
2 ENCODE: Encoding NetFlows for Network Anomaly Detection	17
2.1 Introduction	18
2.1.1 Our Contribution.	19
2.1.2 Outline of Chapter	19
2.2 Threat Model.	20
2.2.1 Goals of the Adversary.	20
2.2.2 Capabilities of the Adversary.	20
2.3 Related Works	20
2.4 Intuition Behind ENCODE	21
2.5 Encoding Algorithm	22
2.5.1 Computing the Context	22
2.5.2 Creating the Encoding	22
2.5.3 An Illustrative Example	22
2.5.4 Implementation Details	23
2.6 Evaluation of ENCODE.	27
2.6.1 Selection of ML Models	27
2.6.2 Comparison Between Encoding Methods.	28
2.6.3 Dataset Selection.	28
2.6.4 Feature Selection.	28

2.6.5	Encoding Train & Test Data	30
2.6.6	Creating Sequences For Sequential Models	31
2.7	Results	31
2.7.1	Overall Results	32
2.7.2	The Effect of Cluster Number On Model Performance	35
2.8	Use-case and robustness	41
2.8.1	Experimental Results	41
2.9	Conclusion & Future Work	43
3	Monitoring MSAs Deployed on Kubernetes For Anomalies	47
3.1	Introduction	48
3.2	Related Work	49
3.2.1	Kubernetes Security Management	49
3.2.2	General Approaches	50
3.2.3	Our Contribution	51
3.3	Threat Modelling in Kubernetes	52
3.3.1	Microservices Architectures	52
3.3.2	Kubernetes Threat Matrices	52
3.4	Methodology	53
3.4.1	State-Machine Learning	53
3.4.2	Feature Selection & NetFlow data Encoding (Step 1)	55
3.4.3	Anomaly Detection using PDFA (Step 2)	56
3.5	Cluster Environment Setup	56
3.5.1	Kubernetes Cluster Setup	57
3.5.2	Description of the Attack Scenarios	57
3.5.3	Data Collection	60
3.6	Experiment Results	62
3.6.1	Performance Results	62
3.6.2	Likelihood Probability Plots	62
3.7	Conclusion & Future Work	63
4	SEQUENT: State Frequency Estimation For Network Anomaly Detection	67
4.1	Introduction	68
4.2	Preliminaries	69
4.2.1	State Machine	69
4.2.2	State Machines Considered in SEQUENT	70
4.3	Assumptions of the Adversary	71
4.4	Related Works	71
4.5	SEQUENT	73
4.5.1	Intuition Behind SEQUENT	73
4.5.2	SEQUENT's Anomaly Detection Pipeline	73
4.6	Experiment Setup	77
4.6.1	Dataset Selection	77
4.6.2	Creating Train/Test Splits	78
4.6.3	Methods for Comparisons	78
4.6.4	Feature Selection & FlexFringe Setup	79

4.7	Evaluation Results	80
4.7.1	Effectiveness of SEQUENT	80
4.7.2	Runtime Analysis of SEQUENT	81
4.7.3	Example Analysis Using Root Causes	82
4.7.4	SEQUENT's Robustness	85
4.8	Conclusion & Future Work	89
4.9	Acknowledgments	89
II	Validating Deployment Against Implementation	95
5	Conformance Analysis Tool For Microservice Applications	97
5.1	Introduction	98
5.2	Running Example	98
5.3	CATMA	99
5.4	Tool Evaluation	103
5.5	Related work	105
5.6	Conclusion & Future Work	105
	Appendix A: Interview Transcript of CATMA's Pilot Study	109
III	Automated Testing with Model Inference	115
6	Automated Test-Case Generation for REST APIs Using Model Inference Search Heuristic	117
6.1	Introduction	118
6.2	Background & Related Work	119
6.2.1	Automated Test Case Generation using EAs	119
6.2.2	Automated Testing of REST APIs	120
6.2.3	State Machine & Testing REST APIs	120
6.3	Approach	122
6.3.1	Transforming Log Events to Traces	122
6.3.2	Continuous Learning of State Machine	124
6.3.3	Defining Fitness using State Machine	124
6.3.4	Evolving and Selecting Test Cases	125
6.4	Empirical Study	125
6.4.1	Research Questions	125
6.4.2	Selection of Applications	126
6.4.3	Version & Parameter Settings for EvoMaster	127
6.4.4	Execution Procedure of Each Algorithm	128
6.4.5	Comparison Metrics	128
6.5	Evaluation Results	128
6.5.1	RQ1: MISH's Performance In Generating Test Cases	129
6.5.2	RQ2: Where does MISH exhibit good/poor performance?	130
6.5.3	RQ3: MISH's Coverage Over Time	133
6.6	Threat to Validity	138
6.7	Conclusion & Future Work	138

Appendix B: Preliminary Results of MISHMOSA	145
6.8 Preliminary Analysis of MISHMOSA's Performance	145
7 Conclusion	151
7.1 Research Questions Revisited	151
7.2 Limitations & Potential Future Works	158
Curriculum Vitæ	163
List of Publications	165
Titles in the SIKS Dissertation Series Since 2016	167

Summary

Over the past decade, the microservice architectural style has gained immense popularity among software companies. Many adopt this architectural style to develop their web services due to its improved maintainability, reliability, and agility. These benefits enable companies to quickly develop and deploy their web services iteratively.

Despite its advantages, microservices are notoriously known to be difficult to debug when software failures occur. Developers often struggle to identify the underlying cause of a failure due to the distributed nature of microservices. In fact, certain software failures only emerge when complex interconnected interactions are triggered between microservices. Therefore, it is becoming increasingly important to develop monitoring tools to streamline the process of determining the underlying cause of problems detected within microservices.

The AssureMOSS project, part of the EU Horizon 2020 Research Program, focuses on delivering various tools or approaches to enhance security at various phases of the microservice development life-cycle. This dissertation, conducted as part of the AssureMOSS project, primarily investigates runtime monitoring of microservices to detect anomalies such as cyber-attacks and discrepancies between observed behaviors and the intended behaviors specified in the source code.

A key project requirement is using lightweight interpretable machine learning models to aid the understanding of detected software failures. State machines, also known as Finite State Automata, have proven to be practical models for modeling behaviors of various software systems. Furthermore, these models are considered to be inherently interpretable as one can visualize the behaviors exhibited by a system. For this reason, state machines are a suitable choice for monitoring microservice applications. This dissertation explores how state machines can be applied to address challenges across various phases of the microservices development life-cycle.

Part I explores how state machines can be utilized to monitor the runtime of microservices for anomalies. In particular, the studies conducted in Part I focus on detecting network anomalies, such as network attacks, within microservice applications. Part I first introduces ENCODE in Chapter 2, an encoding algorithm designed to preprocess NetFlow records, making them suitable for learning state machine models. Chapter 3 demonstrates how state machines can be learned from NetFlow data collected from microservices and evaluate their effectiveness in detecting various network attacks targeting microservices deployed in a Kubernetes cluster. Finally, Chapter 4 introduces SEQUENT, a novel anomaly detection approach that dynamically uses state visit frequencies to compute anomaly scores for behaviors observed at test time. SEQUENT addresses several limitations within the approach employed in Chapter 3, offering practical applications such as ranking and clustering alerts to understand detected anomalies better. Results suggest that SEQUENT achieved considerably better performance than the approach presented in Chapter 3.

Part II of this dissertation explores how state machines can detect discrepancies (non-conformances) between the observed runtime behaviors of microservices and the intended behaviors specified in the source code. Chapter 5 introduces CATMA, a lightweight tool for conducting conformance analysis on microservices applications. CATMA uses state machines to capture observed behaviors and compares them against insights derived from Dataflow Diagrams, representing the intended behaviors specified in the source code. This comparison reveals any non-conformances, and CATMA provides human-readable explanations for these non-conformances to aid developers with identifying potential starting points to debug their microservices applications. CATMA's pilot study shows promising performance results and provides valuable insights to developers. Additionally, CATMA made a small impact in the open-source software community by identifying and fixing a non-conformance detected in an open-source project.

Finally, Part III of this dissertation explores how state machines can guide an Evolutionary Algorithm in the automated generation of test cases for the REST APIs of microservices. Chapter 6 introduces MISH, a search heuristic that continuously learns a state machine model from log statements from microservices. These models are then used to generate more effective system-level test cases. MISH extracts various insights from the state machine to compute the fitness of test cases, which the Evolutionary Algorithm uses to generate better test cases. MISH demonstrates promising results in using state machines to guide an evolutionary algorithm toward generating more effective test cases, even outperforming a state-of-the-art search algorithm, MOSA, in specific scenarios. Additionally, a preliminary study suggests that integrating MISH as a part of MOSA could enhance the quality of system-level tests, creating an interesting future research direction.

This dissertation demonstrates the effectiveness of state machines in modeling the behavioral patterns of microservices across various practical applications, including monitoring, validation, and testing. Consequently, multiple toolchains and algorithms were developed to evaluate the models' efficacy and to facilitate the use of state machines in representing the behavioral patterns of software systems.

Samenvatting

In de afgelopen tien jaar heeft microservices enorm aan populariteit gewonnen onder softwarebedrijven. Velen kiezen voor deze architectuurstijl om hun webservices te ontwikkelen vanwege de verbeterde onderhoudbaarheid, betrouwbaarheid en flexibiliteit. Dankzij deze voordelen kunnen bedrijven hun webservices snel en iteratief ontwikkelen en implementeren.

Ondanks de voordelen staan microservices erom bekend dat ze moeilijk te debuggen zijn als er softwarefouten optreden. Ontwikkelaars hebben vaak moeite om de onderliggende oorzaak van een fout te achterhalen vanwege de gedistribueerde aard van microservices. Sterker nog, het is zo dat bepaalde softwarefouten pas aan het licht komen wanneer complexe, onderling verbonden interacties tussen microservices worden geactiveerd. Daarom wordt het steeds belangrijker om monitoringtools te ontwikkelen die het proces van het bepalen van de onderliggende oorzaak van problemen binnen microservices, kunnen stroomlijnen.

Het AssureMOSS-project, dat deel uitmaakt van het onderzoeksprogramma Horizon 2020 van de Europese Unie, richt zich op het leveren van verschillende tools of methoden om de beveiliging te verbeteren in verschillende fasen van de ontwikkelingscyclus van microservices. Dit proefschrift, die is uitgevoerd als onderdeel van het AssureMOSS-project, onderzoekt voornamelijk runtime-monitoring van microservices om afwijkingen te detecteren, zoals cyberaanvallen en inconsistenties tussen geobserveerd gedrag en het bedoelde gedrag dat in de code is gespecificeerd.

Een belangrijke eis van het project is het gebruik van lichtgewicht interpreteerbare modellen om inzicht te krijgen in gedetecteerde softwarefouten. Toestandsdiagrammen, ook wel bekend als eindigetoestandautomaten, zijn bewezen praktische modellen te zijn voor het modelleren van gedrag van verschillende softwaresystemen. Bovendien worden deze modellen beschouwd als interpreteerbaar omdat men het gedrag van een systeem kan visualiseren. Om deze reden lijken toestandsdiagrammen een geschikte keuze voor het monitoren van microservices. Dit proefschrift onderzoekt hoe toestandsdiagrammen kunnen worden toegepast om uitdagingen in verschillende fasen van de ontwikkelingscyclus van microservices aan te pakken.

Deel I onderzoekt hoe toestandsdiagrammen kunnen worden gebruikt om de runtime van microservices te controleren op anomalieën. De studies in deel I richten met name zich op de detectie van netwerkanomalieën, zoals netwerkaanvallen, binnen microservices. Deel I introduceert eerst ENCODE in hoofdstuk 2, een algoritme ontworpen om NetFlows te verwerken, zodanig dat ze geschikt zijn voor het leren van toestandsdiagrammen. Hoofdstuk 3 laat zien hoe toestandsdiagrammen geleerd kunnen worden van NetFlows die zijn verzameld van microservices. De modellen worden geevalueerd op effectiviteit in het detecteren van verschillende netwerkaanvallen gericht op microservices in een Kubernetes-cluster. Ten slotte introduceert hoofdstuk 4 SEQUENT, een nieuwe methode voor anomaliedetectie die gebruik maakt van de frequenties van de toestanden om dyna-

misch scores uit te rekenen voor gedrag dat tijdens de test wordt waargenomen. SEQUENT verhelpt verschillende beperkingen van de aanpak in hoofdstuk 3 en biedt praktische toepassingen zoals het rangschikken en clusteren van waarschuwingen om gedetecteerde anomalieën beter te begrijpen. Resultaten suggereren dat SEQUENT aanzienlijk betere prestaties levert in vergelijking met de aanpak in hoofdstuk 3.

Deel II van dit proefschrift onderzoekt hoe toestandsdiagrammen kunnen worden gebruikt om inconsistenties (ook wel non-conformances genoemd in dit proefschrift) te detecteren tussen het geobserveerde runtime gedrag van microservices en het bedoelde gedrag in de code. Hoofdstuk 5 introduceert CATMA, een tool voor het uitvoeren van conformiteitsanalyse op microservices. CATMA gebruikt toestandsdiagrammen om waargenomen gedragingen vast te leggen en vergelijkt deze met inzichten uit Dataflow Diagrammen, die het bedoelde gedrag in de code weergeven. Deze vergelijking onthult eventuele non-conformances en CATMA geeft leesbare verklaringen voor deze non-conformances om ontwikkelaars te helpen bij het identificeren van mogelijke startpunten voor het debuggen van hun microservices. De studie van CATMA laat veelbelovende resultaten zien op het gebied van prestaties en het bieden van bruikbare inzichten aan ontwikkelaars. Daarnaast heeft CATMA een kleine impact gemaakt in de open-source software gemeenschap door het identificeren en leveren van een oplossing voor een non-conformance die in een open-source project werd gedetecteerd.

Tot slot onderzoekt deel III van dit proefschrift hoe toestandsdiagrammen een evolutionair algoritme kunnen helpen bij het geautomatiseerd genereren van testen voor de REST API's van microservices. Hoofdstuk 6 introduceert MISH, een zoekalgoritme dat continu toestandsdiagrammen leert uit logboekoverzichten van microservices. Deze modellen worden vervolgens gebruikt om betere testen op systeemniveau te genereren. MISH haalt verschillende inzichten uit de toestandsdiagram om de fitness van de testen te berekenen, die door het evolutionaire algoritme worden gebruikt om betere testen te genereren. MISH laat veelbelovende resultaten zien in het gebruik van toestandsdiagrammen om een evolutionair algoritme te helpen met het genereren van effectievere testen, waarbij het zelfs een state-of-the-art zoekalgoritme, MOSA, in bepaalde scenario's overtreft. Daarnaast suggereert een voorlopige studie dat het integreren van MISH als onderdeel van MOSA de kwaliteit van systeemniveau testen zou kunnen verbeteren, die een interessante toekomstige onderzoeksrichting creëert.

Dit proefschrift toont de effectiviteit aan van toestandsmachines bij het modelleren van gedragspatronen van microservices in diverse praktische toepassingen, waaronder monitoring, validatie en testen. Als gevolg hiervan zijn meerdere toolchains en algoritmen ontwikkeld om de doeltreffendheid van de modellen te evalueren en het gebruik van toestandsmachines bij het representeren van gedragspatronen van softwaresystemen te vergemakkelijken.

Acknowledgments

While I thought this would be the easiest part of my dissertation to write, it ended up being one of the most difficult. Not because I wanted it to sound formal and match the rest of the dissertation, and certainly not because I did not know what to write. Quite the opposite. There is so much that I would like to say, and so many people I would like to thank, that I do not know where to begin. In fact, I am afraid that the book you are holding would become far too thick and heavy if I were to include everything that has been on my mind while writing these acknowledgments. My PhD journey was not only about writing papers and learning from the research conducted by others. It was also a journey filled with personal challenges and lessons. It taught me how to manage multiple projects at the same time, how to balance deadlines with spending time with friends, family, and myself, how to deal with frustration when things did not go according to plan, and perhaps most importantly, how to say no. The last one remains a challenge to this day. Although I have learned to say no more often than before, I still do not do it as often as I probably should. However, as one of my former managers once told me: *“You are who you are; trying to change yourself completely will only cause you to lose yourself”*. Helping others, sometimes at the expense of my own time, is part of who I am. While I continue to work on setting better boundaries, I have also learned to accept and appreciate this part of myself.

When I read in a colleague’s acknowledgments that his PhD journey had been a rollercoaster ride, I found myself reflecting on my own experience. Looking back, I could not agree more. It has indeed been a rollercoaster ride. From the excitement of having your first paper accepted to the disappointment of seeing another paper rejected six times. From believing that you are conducting impactful research to seriously questioning whether you should continue pursuing the PhD at all. Throughout these years, I experienced emotional highs and lows that often arrived when I least expected them. The fluctuation between excitement, frustration, confidence, and doubt felt remarkably similar to the twists, turns, and sudden drops of a rollercoaster ride. Similar to the beginning of a rollercoaster ride, you start by thinking that it is going to be a very long journey (at least that is how it feels to me, as I am not a big fan of rollercoasters). However, once you are on the ride, you realize that it is over much sooner than you expected. Four years may sound like a long time, but when you are fully focused on your research, those years pass surprisingly quickly. Although I might be willing to stand in line for another rollercoaster ride, I cannot say the same for a PhD journey. One such journey is more than enough for me.

But I could not have made it through this journey without the support of many people. First of all, I would like to thank my promoters, **Sicco Verwer** and **Annibale Panichella**.

Sicco: Thank you very much for your supervision over the past four years (and also for the two and a half years during my Master’s). Thank you for always asking insightful questions about my work, questions that often revealed the blind spots in my research. These questions were always very helpful, not only for improving my work but also for teaching me to ask myself the same critical questions. Thank you for encouraging me to

keep reading during the first two years of my PhD, something I do not naturally enjoy. Thank you for the time you took to debug code with me and for the many brainstorming sessions on how to refine my work. Thank you for being there during the lowest point of my PhD and for encouraging me not to give up. I have learned a great deal from you over the past four years, and I hope you will continue to challenge your students with your insightful questions.

Annibale: Thank you for supervising me over the same period as Sicco. From the moment I needed to find another promotor, you were the first person who came to mind. Thank you for always pushing me to achieve the highest standard in my work and for encouraging me to submit to ICSE and other top-tier conferences. Your passion (and workaholic nature) has always been very inspiring to me. Thank you also for being there during one of my lowest periods, and for sharing your own experiences and struggles from your PhD. This helped me get through that time and gave me motivation to continue. Thank you for showing me the more social and enjoyable side of research at ICSE, and for all the laughs from our silly (sometimes very childish) jokes at conferences and at the university. This would not have been possible without your energetic and humorous personality. Keep that passion burning, and continue inspiring the next generation of students.

There are many colleagues at the university to whom I would like to extend my gratitude. First of all, I would like to thank the PhD students from the Cybersecurity research group. Thank you Florine, Huimin, Jing, Marina, Oguzhan, and Stefano for the many enjoyable conversations at the coffee machines on a wide range of topics. Thank you Jelle for the discussions on random topics and for exchanging experiences, even after your PhD. It was always a pleasure to talk with you, and I have missed those conversations. Of course, I would also like to thank Sandra from the Cybersecurity group. Thank you for your patience in answering all my questions during my time in the group, and for the always enjoyable coffee chats. I would also like to thank a few master students: Bram, Charlie, and Wessel. Bram and Wessel, thank you for the fun chats and the strange card-sliding game we invented to play when we wanted to take a break. Thank you Charlie for giving me the opportunity to supervise your Master's thesis. It was a pleasure to discuss your work and help you explore different directions in your research.

Second of all, I would like to thank the colleagues from the Algorithmic research group. Thank you all for the warm welcome to the research group. Thank you Anna (Latour), Eggy, Imko, Issa, Jhon, Johannes, Kim, Koos, Maaïke, Noah, and Sterre for the many enjoyable coffee chats and the most random lunch-break discussions. I am still not entirely sure what counts as "soup". Thank you Harry and Xueying for coming together to make dumplings for the potluck. It was a fun experience, and I am still slightly surprised that we managed to finish in time (it definitely felt like we were running behind while making the dumplings). A special thanks goes to the colleagues from the "Pedro" office (E4.20): Jeff, Maarten, Reuben, Tillman, Elif, and Johannes. Thank you for the silly memes we share in our Mattermost channel, and for all the stories and experiences you have shared over time. What happens in the "Pedro" office stays in the "Pedro" office.

Next, I would like to thank Mitchell. Thank you for helping out with AISTR, and for exchanging ideas and experiences around our work. Thank you for being part of my graduation committee (please have mercy on me with your questions). I also want to thank you for one of the funniest experiences I had at ICSE. It was the funniest dinner I

had at ICSE, and I could not stop laughing that night at the restaurant.

I would like to thank everyone from the Cyber Analytics Lab. Thank you Azqa, Chris, Daniël, (other) Daniël, Robert, Simon, and Tom. Thank you all for the fun times we shared, from online game nights during COVID, to playing board games at Sicco's place, and trips to Efteling and Walibi. Azqa and Daniël, thank you for the many laughs at various BNAIC conferences, after-work drinks, and other hangouts. Doing silly things together helped me greatly throughout my PhD. Thank you both as well for sharing your PhD experiences with me; it made me feel less alone in this journey. A special thanks to Daniël for listening to my complaints on various topics, talking about gym workouts, and being my rubber duck whenever I got stuck with my code. I will greatly miss the breaks and our chats. Robert, thank you for the conversations we had at the start of our PhD, and for exchanging ideas and thoughts on a wide range of topics. It was always an interesting discussion. Thank you Simon for inviting me to your wedding and for the interesting discussions on what we could do within the field of state machine learning.

I would like to thank all my colleagues from the Data & AI Hub. Thank you all for the very warm welcome during my onboarding. As I have said many times already, I felt like I landed in a very good team on the very first day, and I still feel that we are making a lot of impact within the company. Thanks to Adrian, Jacky, Jesse, Kris, Luke, Mathew, Max, Merijn, Stefan, and Vincent for not only making my work fun and impactful but also for all the laughs we shared during team drinks and events. Let us continue to work hard and play hard. Of course, I cannot forget to mention Olivier. Although you have a new role now, you are still part of the team in my heart. Thank you for always helping me when I encounter the weirdest issues on my machine or in our application—you always seem to know how to fix them. And of course, thank you also for the laughs during team drinks and events. I would also like to thank Alexa, Alper, Amir, and Ernst for the fun conversations and laughs during lunch and team events. You all allowed me to step out of my daily role of writing code, checks, or flows, and talk about life in general instead. Thank you Chris for sacrificing your spot when I arrive later in the office, although it seems you do not offer that quite as often anymore now that you have become a manager (just kidding, of course). Thank you also for the regular check-ins and for keeping me sharp when I was working on my first flow. Finally, I would like to give a special thanks to Remco and Muriel. Thank you both for always asking how I am doing personally. I tend to quickly focus on work, and you both remind me that it is not always about work; personal well-being is just as important.

I would like to thank all my friends. Thank you all for the many fun times and all the things we have done together. This is going to be a long list, so please bear with me while I (attempt to) mention everyone. First of all, thank you Michelle for suggesting that I join the Data & AI Hub. Without this suggestion, I would not have ended up in my current team. Thank you also for the many chats about travel plans; you inspired me to add new places to my bucket list. Next, I would like to thank Willem and Simba. Thank you for the motivation in the gym and for the karaoke sessions.

I would like to thank the following people, as they played a very important role in helping me stay mentally sane during my PhD journey. Thank you all for the trips, hangouts, dinners, and game nights. These really helped me take my mind off the PhD (something much needed once in a while). I am also grateful for your patience in listening to my com-

plaints and frustrations over the past four years. Thank you Alex, Amy, Jackie, Johnson, Kevin, Kayan, Theresa, and Yongjie.

I would also like to thank everyone from the “RiceFam” . Thank you all for the many fun things we have organized together. What started as a small tennis group has grown into a large dinner group. The dinners are getting fancier each time, and I am slightly worried about my turn; I am not sure whether I can live up to the current standards. Thank you Amy, Boning, Bingxi, Jacky, Janet, Jenny, Johnson, Eline, Michael, Michel, Shirley, Wei Wha, Xinbo, and Yenly.

I would like to thank Mandy. Although we have both been busy with our own lives and have not spoken very often, we have still managed to maintain the close friendship we have had since Colegio Arubano. Thank you very much for inviting me to your wedding in Aruba. I still find it funny how you went from (very confidently) saying “*I do not want to get married*” at Colegio Arubano to saying “*Yes, I do*” a few years later. Nevertheless, the wedding was beautiful, and I am very happy that I was able to attend. Thank you also for being there for me during one of the lowest points of my PhD journey, and for listening to me vent all my frustrations. I hope we can catch up more often in the future.

I would like to give a very special thank you to Junhan. Who would have thought that I would end up in the same research group as you at the final stage of my PhD journey. After that one single graduate school course, I thought we would have parted ways and never met again. Somehow, we met again at an ELLIS poster session and later at Algorithmics. Thank you very much for listening to my stories and complaints about rejected papers. Thank you for giving me badminton training just because I wanted to join an amateur competition. Thank you for being there during some of the lowest points of my PhD. Thank you for pushing me (sometimes) out of my comfort zone to try new things or to look at situations from a different perspective. This has (very slowly) helped me develop further. Thank you also for convincing me to go on different trips. Each of these places had its own unique character and left a special memory with me. But I cannot thank you enough for your support during the final part of my PhD journey. I think this was the most challenging period in terms of planning: my defense, finalizing my thesis, arranging my parents’ visit, and doing my work at NN. During this time, my temper was not always at its best. You heard me complain about many things, you saw me getting irritated at times, and you also heard me saying “*no*” more often than usual. Yet you stayed calm and supportive throughout. I know this period was not easy for you either, and for that I am deeply grateful. Thank you, and I look forward to creating many more memories together.

Last but not least, I would like to thank my brother Hao and my parents. Thank you, Hao, for always sharing your experiences from your PhD. Your experiences, stories, and achievements have always motivated me to do better in my own research, although I still believe your work will have a greater impact on society than mine. I am rooting for your research and hope to see it get adopted in clinics in the future. Thank you for always spending time with me during special holidays, birthdays, or simply during casual hangouts to talk about life. I wish you all the best with the final part of your PhD journey, and I am excited to see what you will achieve next.

As for my parents, thank you very much for always checking in on how I am doing and for reminding me to take care of myself. Although you never pushed me to achieve anything at the university (not even to pursue a PhD), I still wanted to make you proud

by completing my PhD. Thank you both for your unconditional support, and I am very happy that you will be able to attend my defense. I hope you will enjoy the trip that Hao and I have planned for you, and I also hope to take you to visit more places in Europe in the future. I plan to visit you more often as well, as I have not been back to Aruba very often during my PhD journey. Thank you both, and see you soon.

多谢, *Dank, Danki, Gracias, Merci, Thanks.*

Clinton

Delft, May 2026

1

Introduction

Microservices have gained immense popularity in the past decade within the software development community and the industry. Large corporations such as Google, Meta, and Microsoft leverage this (software) architectural style to provide their web services to their users. Google Drive is a widely used example, serving over two billion users as of December 2023 [10]. This service enables its users to easily store and share files with friends, family, or colleagues. However, like any software system, microservices also suffer from software failures. These can result from bugs introduced by developers during the development process or from cyber attacks launched by malicious actors [11]. With the rapid development of microservices, it has become increasingly important to detect potential software failures as early as possible within every phase of the microservice's software development life-cycle (SDLC).

Not only is it crucial to detect potential failures as early as possible, but providing an explanation or root cause for each failure is equally important. These explanations or root causes reduce the time and effort a developer or analyst spends identifying the underlying cause for a failure occurring in the system. Over the past few years, various machine learning (ML) techniques have been introduced to address software failure detection challenges. In particular, neural networks (NN) have been very successful in tackling challenges such as automated vulnerability detection [22] or anomaly detection [14]. However, NNs are considered to be black-box models, making it difficult for humans to comprehend how they arrived at their conclusions [19]. Additionally, multiple NN-based approaches demand additional hardware requirements (e.g., a GPU) to train a model, making them less practical for users with financial constraints.

On the other hand, numerous studies have shown that finite state machines (FSMs) are effective at modeling behavioral patterns of software systems [9, 12, 13, 17, 18, 20, 27]. Moreover, this type of model is considered inherently interpretable [19] (i.e., it can be visualized and used as an explanation). It does not require additional hardware requirements for training, making it a more lightweight and practical option for deployment.

The applications of FSMs for detecting software faults in various phases of the SDLC of microservices remain relatively unexplored. Since FSMs are learned from discrete behavioral events, it is not trivial what kind of data should be used to create discrete behavioral events for microservices. Creating too many unique events risks a combinatorial explosion of states within a state machine, resulting in large models that require considerably longer training times. Conversely, overgeneralizing behavioral events of microservices can cause the model to miss essential patterns necessary for detecting software faults.

While the literature includes various studies investigating the effectiveness of FSMs in detecting software faults, their focus was not on microservices. This results in a lack of a clear baseline for researchers applying FSMs to microservices. It is not trivial which type of data is suitable for learning FSMs, how to process the data for learning FSMs, and what kind of information should be utilized from FSMs to derive insights for detecting and finding the root cause of software faults.

This dissertation aims to answer these questions by developing various algorithms and tools that make FSM suitable for modeling behavioral patterns of microservices. The studies presented in this dissertation provide various toolchains and algorithms developed to facilitate using FSMs to model microservices' behaviors. The studies employ FSMs in various practical applications for detecting software faults within microservices and target

various phases within the SDLC of microservices. Furthermore, the results of the studies serve as baselines for practitioners or researchers seeking to conduct similar studies.

1.1 Background & Context

A description is given for each of the two key terminologies used in this dissertation: Microservices and Finite State Machines. Then, an overview of the AssureMOSS project is presented to establish the context that motivated the studies carried out within this dissertation.

1.1.1 Microservices

In a traditional monolithic software system, all components are developed, maintained, and deployed as a unified application. Microservice architecture, by contrast, is an architectural pattern in which a software system's core functionalities are divided into smaller, independent components known as services. These services communicate with each other using lightweight network protocols such as Hypertext Transfer Protocol (HTTP). Unlike the monolithic systems, each service can be developed, maintained, and deployed independently, resulting in increased software application modularity, scalability, and resilience. An example comparison between a monolithic system and a microservice application is presented in Figure 1.1.

Despite its many advantages, microservices are notoriously difficult to debug due to their distributed nature, where calls between services can occur asynchronously. Several studies have highlighted that debuggers can spend days trying to pinpoint the root cause of faults within a microservice application [16, 21, 25, 26]. Consequently, there is a clear need for a set of tools and approaches that enable faster detection and diagnosis of faults within microservices. These tools or approaches should target various phases of the SDLC of a microservice application.

1.1.2 Deterministic Finite State Machines

A state machine, formally known as a finite state machine or finite state automaton, is a mathematical model for modeling discrete behavioral patterns of a (software) system S . State machines come in two flavors, namely deterministic and non-deterministic. This dissertation exclusively focuses on deterministic state machines. For the sake of simplicity, the term "state machine" is used to refer to a "deterministic state machine" throughout this dissertation. A state machine is formally defined as a quintuple $(\Sigma, Q, q_0, \delta, F)$ [23], where:

- Σ is a finite alphabet containing all possible (discrete) events that can occur within S .
- Q is a finite set of states.
- $q_0 \in Q$ is a unique start state.
- $\delta : Q \times \Sigma \rightarrow Q$ is the transition function, denoting which state the state machine will transition to based on the current state that it is in and the next input symbol $a \in \Sigma$.

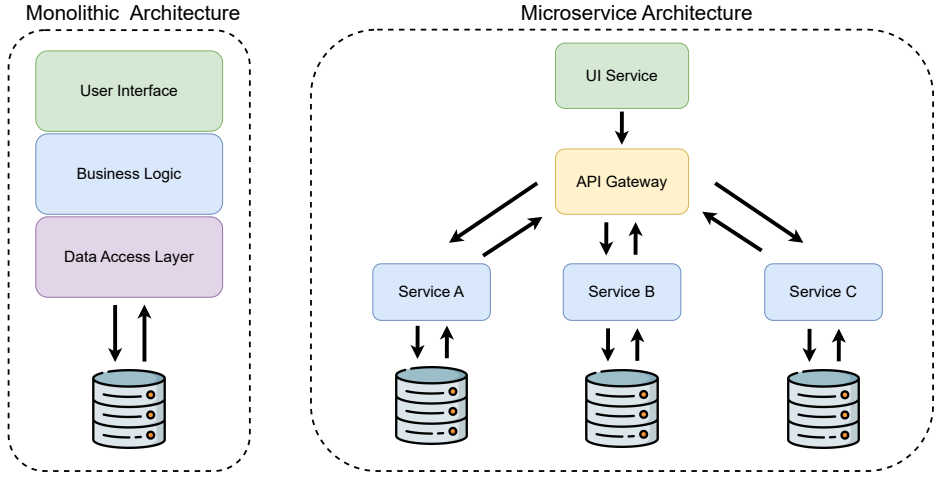


Figure 1.1: Example comparison between a monolithic software system and a microservice application.

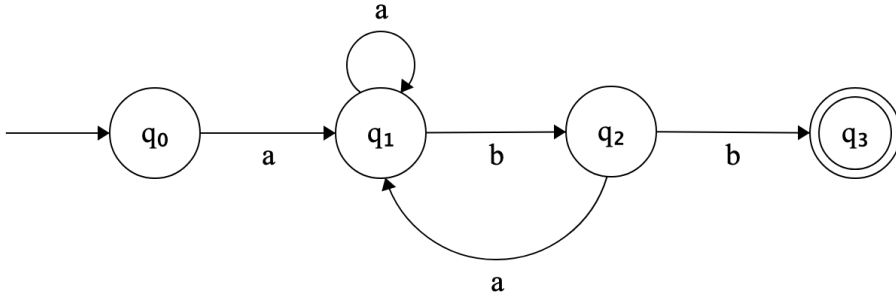


Figure 1.2: An illustrative example of a state machine. At the end of trace $t = aabab$, S will end in state q_2 .

- $F \subseteq Q$ is the set of final states.

Given an arbitrary trace t containing an arbitrary number of discrete events from Σ , the state machine can be used to follow the path that t traverses within the state machine. Using this path, one can determine both the sequence of states S traverses and the final state S reaches at the end of t . This sequence of state transitions provides insight into the sequential behavior exhibited by S for t . Figure 1.2 presents an illustrative example of a state machine. Given the arbitrary trace $t = aabab$, S will end up in state q_2 at the end of t . Since q_2 is a rejecting state, t is not a valid string and the state machine will reject it.

Learning State Machines from Data

A state machine can be learned from data collected from an arbitrary system S . In particular, state machines can be learned from traces (i.e., sequences of discrete events) collected from S . There are two primary approaches for learning a state machine from traces collected from S : *active* and *passive*-learning. The L^* -algorithm, proposed by Angluin [7], is a

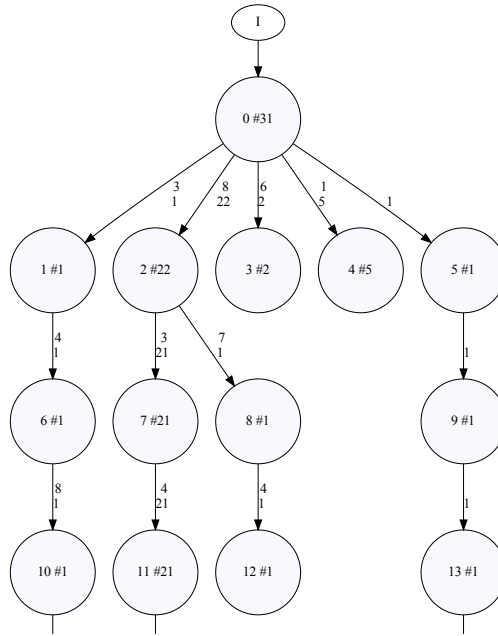


Figure 1.3: An illustrative example of a Prefix Tree Acceptor (PTA) extracted from discretized traces collected from a printing service (ProxyPrint) that implements a microservice architecture.

well-established algorithm used to learn state machines using the first approach. The algorithm continuously queries S for outputs by supplying various input traces. The resulting input-output pairs are used to construct the state machine. The algorithm iteratively tries to find a counterexample to verify the correctness of the constructed model. If a counterexample is found, the algorithm updates the model and the process repeats until no further counterexamples are discovered.

Contrary to the *active* approach, the *passive* approach does not interact with S during the learning process. Instead, it learns a model solely from the output data of S . This data can be obtained by passively observing S 's outputs during runtime or using previously recorded outputs stored in a database. Since this approach does not actively interact with S to collect output data, it is referred to as passive learning. The passive approach commonly uses state-merging algorithms to learn a state machine, as these were demonstrated to be most effective in learning a state machine using this approach [15]. The algorithm begins by first constructing a tree-like state machine known as a Prefix Tree Acceptor (PTA). Figure 1.3 depicts an illustrative example of a PTA. The PTA is constructed from (discretized) output traces collected from microservices (ProxyPrint) during testing. The algorithm iteratively compares pairs of states to identify those that can be merged. States are merged if their similarity score exceeds a predefined threshold. The algorithm repeats this process until no further merges are possible. The resulting output is an approximation of the smallest state machine needed to represent the provided input data.

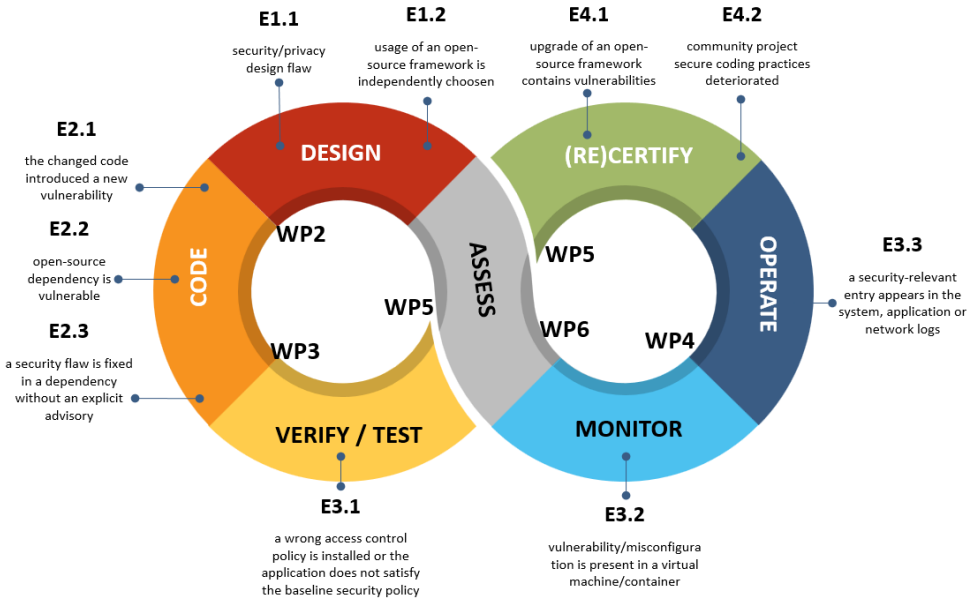


Figure 1.4: The SDLC that the AssureMOSS project considers. The image is from the public report (D6.1) provided by the project [1].

1.1.3 AssureMOSS Project

The AssureMOSS project is part of the European Union Horizon 2020 research and innovation program. Its mission is to design a coherent set of lightweight, automated techniques that help companies and developers conduct various risk assessments on their microservices in response to their increasingly fast-paced development environment [2]. The project covers potential risks associated with different phases of the SDLC of microservices. These risk assessments enable companies and developers to mitigate potential risks tied to software failures within their microservices. Figure 1.4 illustrates the SDLC the AssureMOSS project considers. Since the AssureMOSS project aims to provide lightweight techniques for assessments of microservices, state machine learning is a promising solution for conducting assessments for various phases of the SDLC of microservices. A substantial part of the studies carried out within this dissertation was a part of the AssureMOSS project. Within the AssureMOSS project, the studies focus on the “Assess”, “Design”, “Monitor”, and “Operate” phases of the SDLC.

1.2 Research Goals & Questions

While several studies demonstrated that state machines are effective in modeling behavioral software systems, their applications within the domain of microservices remain relatively unexplored. This dissertation investigates how to utilize state machines to tackle various challenges within the different SDLC phases of microservices. Additionally, the research in this dissertation also explores how to utilize key features of the state machine to generate interpretations/explanations for the model’s output. These overarching objec-

tives lead to the following research questions (RQ):

- **RQ1:** What outputs of microservices enable the learning of accurate state machines?
- **RQ2:** How can outputs of microservices be transformed into discrete events suitable for learning state machines?
- **RQ3:** How can state machines be applied to model behavioral patterns for anomaly detection in microservices?
- **RQ4:** How can state machines be utilized to generate root causes or explanations for faults or anomalies detected in microservices?
- **RQ5:** How can state machines be utilized to detect non-conformances between code and deployment of microservices?
- **RQ6:** How can state machines be utilized to generate test cases for microservices automatically?

1.3 Contributions

This dissertation not only provides insights and knowledge on the application of state machines on microservices but also introduces several software artifacts that can serve as starting points or baselines for future researchers or practitioners. The studies conducted for this dissertation have resulted in various algorithms and software tools, all made publicly available for use. Specifically, answering **RQ1** and **RQ2** led to the development of ENCODE, an algorithm designed to preprocess outputs of microservices and transform them into discrete events suitable for learning state machine models. ENCODE forms a foundational component for the approaches presented in Chapters 3 and 4, where contextual event discretization is required for effective state machine learning and anomaly detection. Evaluation across multiple studies demonstrated that ENCODE aids in learning more accurate models. Figure 1.5 demonstrates that a clear separation can be made between benign (blue vertical lines) and malicious traces (red vertical lines) by utilizing ENCODE to discretize log traces.

RQ3 and **RQ4** resulted in the development of SEQUENT, an anomaly detection package that leverages state machines for anomaly detection and generates root causes as explanations for the analyst. Additionally, **RQ3** led to the creation of an anomaly detection dataset tailored for microservices.

Answering **RQ5** resulted in the development of CATMA, a lightweight software tool that automatically detects discrepancies between the logic specified in the source code and the behavior observed at runtime. CATMA produces a high-level visualization (see Figure 1.6), showing all discrepancies detected between behaviors specified in the source code and observed at runtime. CATMA also generates human-readable interpretations aimed at helping developers understand and address the detected discrepancies. CATMA detected a discrepancy in a real-world open-source project. A fix was issued with the help of CATMA's output and was accepted by the author of the project.

Finally, **RQ6** led to the development of MISH, a novel search algorithm that guides an evolutionary algorithm (EA) in automatically generating test cases for REST APIs of

microservices. Evaluation of MISH demonstrated promising results, even outperforming a state-of-the-art search algorithm in specific scenarios. Figure 1.7 demonstrates MISH's superior performance in generating effective test cases for a large microservice application (OCVN).

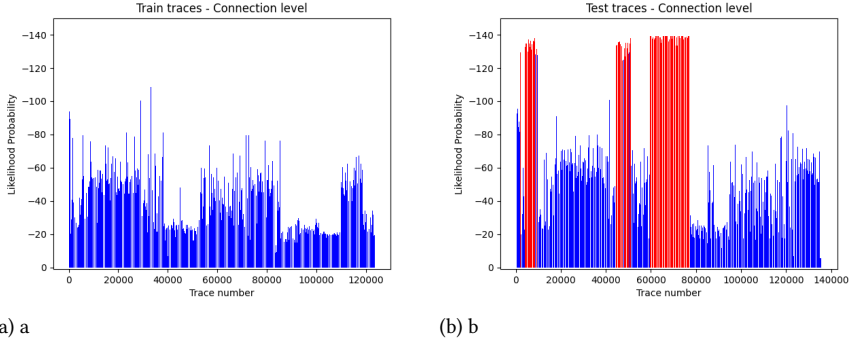


Figure 1.5: Likelihood probability plots generated from discretized log traces of microservices. Traces were discretized using the ENCODE algorithm. The higher the peak, the lower the likelihood probability of a given trace. Subfigures *a* and *b* present the probabilities of the train and test traces sorted by the connections.

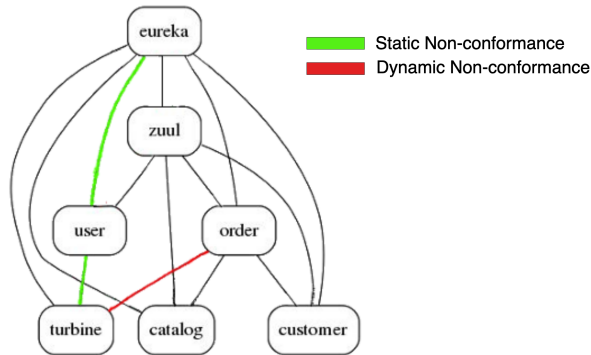


Figure 1.6: CATMA's high-level visualization showing the detected discrepancies between behaviors specified in the source code and observed at runtime.

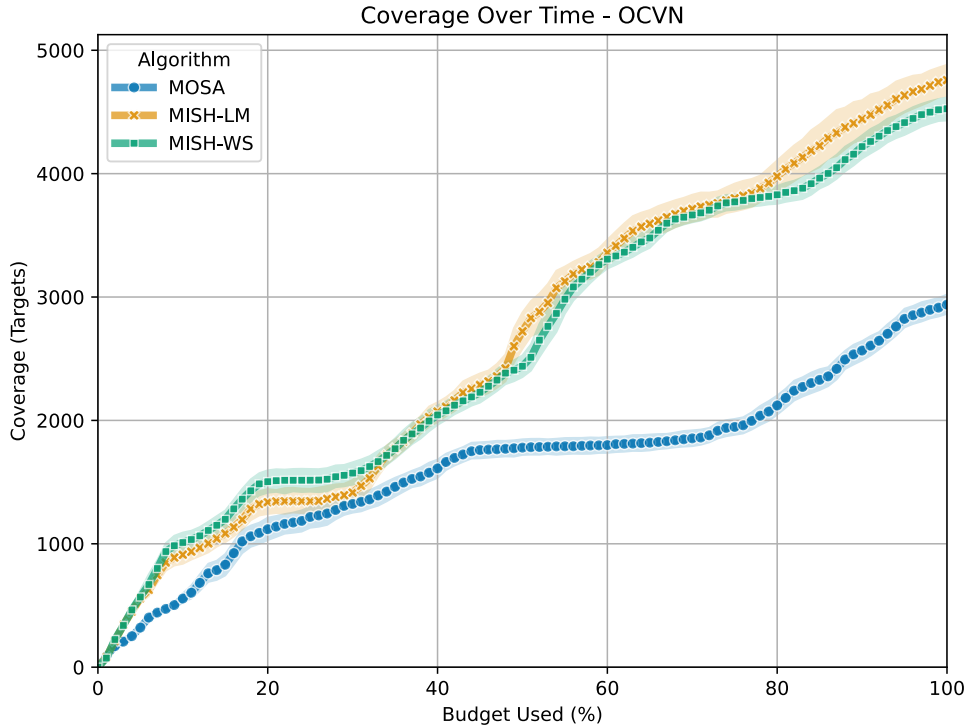


Figure 1.7: MISH outperforms the state-of-the-art algorithm, MOSA, in generating effective test cases.

1.4 Contribution to Open Science

Open science concerns the practice where the code, data, and tools used in the research studies are made available to the public, enhancing the reproducibility and transparency of the studies [24]. Every effort has been made to adhere to open science best practices for studies conducted in this dissertation. Moreover, since the European Union funded the AssureMOSS project, all studies underwent an extensive review by EU representatives to ensure they aligned with open science principles wherever possible. Researchers from the project must transparently describe their research process (where applicable) within their corresponding publicly accessible project reports. Table 1.1 summarizes the public repositories containing either code, tools, or data used to address the research questions in this dissertation. The related research questions are also listed in the table.

Name	Type	Related RQ	Link / DOI
ENCODE	Algorithm	<i>RQ1</i> and <i>RQ2</i>	[3]
AssureMOSS Kubernetes Run-time Monitoring Dataset	Dataset	<i>RQ3</i>	[8]
SEQUENT	Framework	<i>RQ3</i> and <i>RQ4</i>	[6]
CATMA	Tool	<i>RQ5</i>	[4]
MISH	Algorithm	<i>RQ6</i>	[5]

Table 1.1: Public links to code, data, or tools used to address the research questions in this dissertation.

1.5 Research Outline

This dissertation is organized into three main parts, each addressing research questions related to a specific phase of the SDLC of microservices. The following subsections outline the content of each part and the corresponding chapters.

1.5.1 Part I: Monitoring Deployment for Anomalies

The first part of this dissertation focuses on the “Assess”, “Monitor”, and “Operate” phases of the SDLC of microservices. Specifically, the studies conducted in this part investigate how state machines can be employed to monitor the runtime of microservices for network anomalies. The aim for this **Part I** is to address research questions *RQ1-RQ4*. **Chapter 2** addresses research questions *RQ1* and *RQ2* by introducing ENCODE, an encoding algorithm designed to transform NetFlow records collected from microservices into discrete events for learning a state machine. ENCODE serves as a foundational component for the studies presented in subsequent chapters, as contextual event discretization is essential for learning meaningful state machine representations and performing anomaly detection effectively. **Chapter 3** tackles research question *RQ3* by learning a state machine from NetFlow records collected from microservices deployed on a Kubernetes cluster and subsequently using the model for network anomaly detection within the Kubernetes cluster environment. Finally, **Chapter 4** is built on top of the study done in **Chapter 3**. This chapter aims to answer research question *RQ4* by proposing SEQUENT, a novel anomaly detection framework that utilizes state visit frequencies to determine an anomaly score for an arbitrary trace seen at test time. The study also investigates how state frequencies can be used to generate root causes to explain the anomalies detected by SEQUENT.

1.5.2 Part II: Validating Deployment Against Implementation

Part II of the dissertation focuses on the “Design” phase of the SDLC of microservices. This part consists of a single chapter, **Chapter 5**, which addresses research question **RQ5** by introducing CATMA. Specifically, CATMA investigates how state machines can be utilized to detect “non-conformances” (or discrepancies) between the behavior specified in the logic of the code and the behavior observed at the microservice’s runtime. Additionally, CATMA also contributes to research question **RQ4** by generating human-readable explanations that highlight the potential root causes of the detected “non-conformances”.

1.5.3 Part III: Automated Testing with Model Inference

The final part of this dissertation was conducted outside the university’s contribution to the AssureMOSS project and focuses on the “Code” phase of the SDLC of microservices. This part consists of a single chapter addressing the final research question, **RQ6**. The study presented in **Chapter 6** investigates how state machines can automatically generate test cases for REST APIs of microservices. In this approach, a state machine is learned from log statements outputted by the microservices, and the resulting model is used to define the fitness of test cases produced by an EA.

The final chapter, **Chapter 7**, summarizes the findings of this dissertation. It revisits each research question and outlines potential future works for each corresponding research question.

References

- [1] D6.1. Integrated Toolflow Evaluation and Demonstration Plan. <https://assuremoss.eu/en/resources/Deliverables/D6.1-Integrated-Toolflow-Evaluation-and-Demonstration-Plan>. [Accessed 6-2-2025].
- [2] AssureMOSS. <https://assuremoss.eu/en/>, 2020. [Accessed 6-2-2025].
- [3] ENCODE: Encoding NetFlows for Network Anomaly Detection. <https://github.com/tudelft-cda-lab/ENCODE>, 2022. [Accessed -2-2025].
- [4] Conformance Analysis Tool for Microservice Applications. <https://cyber-analytics.nl/catma.github.io/>, 2024. [Accessed 11-2-2025].
- [5] EvoMaster: A Tool For Automatically Generating System-Level Test Cases (Forked Repository). <https://github.com/ClintonCao/EvoMaster>, 2024. [Accessed 11-2-2025].
- [6] SEQUEENT: State Frequency Estimation for Anomaly Detection. <https://github.com/tudelft-cda-lab/SEQUENT>, 2024. [Accessed 11-2-2025].
- [7] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987.
- [8] Clinton Cao and Agathe Blaise. AssureMOSS Kubernetes Run-time Monitoring Dataset. https://data.4tu.nl/articles/dataset/AssureMOSS_Kubernetes_Run-time_Monitoring_Dataset/20463687/1, 2022.
- [9] Joeri de Ruiter and Erik Poll. Protocol state fuzzing of TLS implementations. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 193–206, Washington, D.C., August 2015. USENIX Association.
- [10] Fabio Duarte. Google workspace user stats (2024). <https://explodingtopics.com/blog/google-workspace-stats>, 2023. [Accessed 04-02-2025].
- [11] Jan Eloff and Madeleine Bihina Bella. *Software Failures: An Overview*, pages 7–24. Springer International Publishing, Cham, 2018.
- [12] Paul Fiterau-Brostean, Bengt Jonsson, Robert Merget, Joeri de Ruiter, Konstantinos Sagonas, and Juraj Somorovsky. Analysis of DTLS implementations using protocol state fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2523–2540. USENIX Association, August 2020.
- [13] Gudmund Grov, Marc Sabate, Wei Chen, and David Aspinall. Towards intelligible robust anomaly detection by learning interpretable behavioural models. *NISK* 7, 32:1–16, 2019.
- [14] Max Landauer, Sebastian Onder, Florian Skopik, and Markus Wurzenberger. Deep learning for anomaly detection in log data: A survey. *Machine Learning with Applications*, 12:100470, 2023.

- [15] Kevin J. Lang, Barak A. Pearlmutter, and Rodney A. Price. Results of the abbadingo one dfa learning competition and a new evidence-driven state merging algorithm. In Vasant Honavar and Giora Slutzki, editors, *Grammatical Inference*, pages 1–12, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.
- [16] Valentina Lenarduzzi and Annibale Panichella. Serverless testing: Tool vendors’ and experts’ points of view. *IEEE Software*, 38(1):54–60, 2021.
- [17] Qin Lin, Sridha Adepu, Sicco Verwer, and Aditya Mathur. Tabor: A graphical model-based approach for anomaly detection in industrial control systems. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, ASIACCS ’18, page 525–536, New York, NY, USA, 2018. Association for Computing Machinery.
- [18] Petr Matoušek, Vojtěch Havlena, and Lukáš Holík. Efficient modelling of ics communication for anomaly detection using probabilistic automata. In *2021 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, pages 81–89. IEEE, 2021.
- [19] Azqa Nadeem, Daniël Vos, Clinton Cao, Luca Pajola, Simon Dieck, Robert Baumgartner, and Sicco Verwer. Sok: Explainable machine learning for computer security applications. In *2023 IEEE 8th European Symposium on Security and Privacy*, pages 221–240. IEEE, 2023.
- [20] Gaetano Pellegrino, Qin Lin, Christian Hammerschmidt, and Sicco Verwer. Learning behavioral fingerprints from Netflows using Timed Automata. In *Proceedings of the IM 2017 - 2017 IFIP/IEEE International Symposium on Integrated Network and Service Management*, pages 308–316. Institute of Electrical and Electronics Engineers Inc., jul 2017.
- [21] Dudekula Mohammad Rafi, Katam Reddy Kiran Moses, Kai Petersen, and Mika V. Mäntylä. Benefits and limitations of automated software testing: Systematic literature review and practitioner survey. In *2012 7th International Workshop on Automation of Software Test (AST)*, pages 36–42, 2012.
- [22] Nima Shiri Harzevili, Alvine Boaye Belle, Junjie Wang, Song Wang, Zhen Ming (Jack) Jiang, and Nachiappan Nagappan. A systematic literature review on automated software vulnerability detection using machine learning. *ACM Comput. Surv.*, 57(3), November 2024.
- [23] Michael Sipser. *Introduction to the Theory of Computation*. Course Technology, Boston, MA, third edition, 2013.
- [24] UNESCO. Understanding open science. 2022.
- [25] Muhammad Waseem, Peng Liang, Gastón Márquez, and Amleto Di Salle. Testing microservices architecture-based applications: A systematic mapping study. In *2020 27th Asia-Pacific Software Engineering Conference (APSEC)*, pages 119–128, 2020.

- [26] Muhammad Waseem, Peng Liang, Mojtaba Shahin, Amleto Di Salle, and Gastón Márquez. Design, monitoring, and testing of microservices systems: The practitioners' perspective. *Journal of Systems and Software*, 182:111061, 2021.
- [27] Dongliang Zhao, Jiaxing Guo, Chunxiang Gu, Yonghui Zheng, and Xieli Zhang. Agl-fuzz: Automata-guided fuzzing for detecting logic errors in security protocol implementations. *Computers & Security*, page 103979, 2024.

Part I

Monitoring Deployment for Anomalies

2

ENCODE: Encoding NetFlows for Network Anomaly Detection

NetFlow data is a popular network log format utilized by many network analysts and researchers. Compared to deep packet inspection, NetFlow is easier to collect, easier to process, and less intrusive to privacy. Numerous studies have used machine learning (ML) to detect network attacks using NetFlow data. Typically, the initial step in these ML pipelines is to preprocess the data before passing it to the ML algorithm. Several approaches exist to preprocess NetFlow data; however, these often apply existing methods to the data, not considering the specific properties of network data. We argue that for data originating from software systems, such as NetFlow or software logs, similarities in frequency and contexts of feature values are more important than similarities in the value itself.

In this work, we propose an encoding algorithm (ENCODE) that directly takes the frequency and context of the feature values into account when the data is being processed. Our algorithm clusters similar types of network behavior based on the computed frequencies and context of the feature values, aiding the process of detecting anomalies within the network. We empirically evaluate the effectiveness of our encoding on three public NetFlow datasets. We encode each dataset using our algorithm and train several ML models for network anomaly detection using the encoded data. Our evaluation reveals that ML models benefit from using our encoding for network anomaly detection.

2.1 Introduction

NetFlow is a well-known format introduced by Cisco [13] in 1996, enabling network administrators to analyze the traffic occurring in the network. Each flow contains network statistics for a connection between two hosts. These statistics enable an administrator to infer whether any strange behaviors are occurring within the network. Using NetFlow data over deep-packet inspection for network analysis comes with multiple advantages; NetFlow is easier to collect, easier to process, and less intrusive to privacy, since it is not possible to inspect every packet in detail. However, NetFlow data comes with limitations. Because NetFlow only reports the summarized statistics of a communication between two hosts, crucial details may be lost, making it more challenging to infer malicious network.

Several NetFlows datasets are publicly available, containing benign and malicious flows [15, 20, 25, 28]. The datasets can be used to train and evaluate the effectiveness of machine learning (ML) algorithms for the detection of network attacks. Before training a ML algorithm on a given dataset, the usual practice is to investigate and preprocess the dataset. This process helps remove any properties that could influence the final model learned from the data. Additionally, this process can help find the most important features for learning accurate models. Several techniques can be applied to preprocess a NetFlow dataset, including normalization of the numerical features [18, 32, 34], feature selection [30, 31, 36], encoding numerical features [10, 21, 29], and extracting aggregated features from existing dataset [9, 17, 26, 27].

Most of these techniques treat the NetFlow data like a traditional data source, encoding the number of transferred bytes, number of transferred packets, and flow duration as standard numerical features. The similarity between flows is determined based on the similarity in the numerical values of these features. We believe that flows generated by different (software) services are not uniform and the similarity between flows should not be determined solely based on the raw feature values. For instance, similarly sized flows (e.g. based on the number of transferred bytes) can have entirely different meanings due to variations in the frequency distribution of their feature values. To address this, we propose a new NetFlow encoding technique (ENCODE) inspired by Word2Vec [24], which treats every unique feature value as a different “word”. The similarities between these “words” (i.e., unique feature values) are determined by their context and frequency. Specifically, we consider features values to be similar based on two criteria. First, the feature values must occur in similar frequencies. Second, the frequencies of the previous and subsequent values (extracted from the previous and subsequent flows) must have a similar distribution. Similar to Word2Vec, our encoding algorithm constructs a co-occurrence matrix to store all the computed frequencies. Each row of this matrix represents the context computed for a unique feature value. We cluster the rows to find groups of values exhibiting similar behavior and use the cluster labels to encode values of the corresponding feature. Our encoding algorithm computes a co-occurrence matrix and a set of cluster labels for each input NetFlow feature. Once the given input features are encoded, the resulting encoded dataset is constructed and passed to the ML algorithm for training.

To evaluate the effectiveness of our algorithm, we preprocessed NetFlow data from three different datasets. Our encoding was applied to three flow features: duration, number of transferred bytes, and number of transferred packets. The protocol was included as a discrete feature, while all other features, such as the used ports, were ignored for

training. Although ports can be useful for detecting network intrusions, they frequently contain spurious patterns due to the data collection setup. Our experiments include a new dataset created for detecting anomalies in a Kubernetes cluster. Across all datasets, our encoding consistently improved the performance of several ML models in the network anomaly detection tasks, suggesting that ML models benefit from our encoding.

Our evaluation indicates that state machine models experienced the greatest performance improvement when using our encoding. For this method, we provide a practical use-case. We describe what an analyst receives as output from the system and how they should use it to infer an attack is taking place. While a sophisticated attacker might attempt to evade detection, this is not straightforward. An attacker who randomizes or systematically modifies the generated flows is still detected by our system. In fact, such modifications often make attack flows more anomalous, thereby simplifying their detection. The attack could avoid detection by mimicking the benign flows used during training, but this requires an undetected pre-existing attack. Moreover, truly mimicking benign behavior would bypass any ML-based detector.

2.1.1 Our Contribution

Our key contributions in this work are summed up as follows:

- We provide a new encoding algorithm that can be used to preprocess NetFlow data. The algorithm extracts context and frequency information, and uses these to build clusters of feature values exhibiting similar behavior.
- We show an application of our encoding algorithm by training ML models from the encoded NetFlow data in an unsupervised manner and using it for anomaly detection.
- We provide a new NetFlow dataset that researchers can use to evaluate anomaly detection methods in a Kubernetes cluster. We also provide the testbed that researchers can use to create their dataset.
- We demonstrate that ML models benefit from using our encoding.
- We show how an anomaly detection system based on our encoding can be used in practice and is resilient to malicious data permutations.

2.1.2 Outline of Chapter

The rest of this paper is structured as follows: we first describe the threat model that we have considered in this work in Section 2.2. Then we list some related works in Section 2.3. We provide the intuition behind our encoding in Section 2.4. In Section 2.5, we describe our NetFlow encoding algorithm. Then in Section 2.6, we describe our procedure in how we evaluate our new algorithm, and the empirical results of our evaluation are presented in Section 2.7. In section 2.8, we discuss a use-case and the robustness of our encoding. Finally, we conclude this paper and provide future work in Section 2.9.

2.2 Threat Model

In this work, we consider a specific threat model for network anomaly detection using models learned from ML algorithms. We briefly describe the goals and the capabilities of the adversary.

2.2.1 Goals of the Adversary

In our threat model, we consider an adversary aiming to launch network attacks against a target network for various objectives, such as infiltrating the network to find sensitive data, taking down a network through a denial-of-service (DoS) attack, operating a botnet, and installing malware. Furthermore, the adversary seeks to evade any attack detection system deployed within the network. However, the adversary does not have access to the network or the detection system's rules and models.

2.2.2 Capabilities of the Adversary

We assume that the adversary is aware of the presence of a ML-based anomaly detection system within the targeted network. However, they lack information about the specific models or algorithms used within the anomaly detection system. The adversary does not have the capabilities to deploy probes within the network to monitor the flows occurring in the network. Moreover, adversary can attempt to evade detection by manipulating the flows they generate, but cannot make these flows appear benign through direct analysis of the anomaly detection system of by studying the benign network data.

2.3 Related Works

There are various techniques for preprocessing NetFlow data to train ML algorithms. One common technique involves normalizing the numerical (non-categorical) feature data using the Z-score method [18, 32, 34]. This method scales numerical feature values to fall within the range of 0 to 1, removing large differences in the feature data that could have a large impact on model prediction results. Normalization ensures that each (numerical) feature would have an equal impact on the model prediction results. However, one drawback of this method is that behavioral patterns present in the NetFlow data may be lost. The normalization is performed by first computing the mean of the feature values for a given input feature, which assumes that there is an average behavior shared between all hosts in the network. This assumption may not hold in networks with diverse software services, as each may exhibit distinct behaviors tied to specific contexts. Consequently, this method may remove important contextual information, potentially reducing the model's accuracy.

Another technique to preprocess NetFlow data is to rank and select features using various methods [30, 31, 35]. These methods focus on using features that provide a substantial contribution to the model prediction results, reducing the number of features needed to train an accurate model. However, these methods do not explicitly use the frequencies and context of the feature values to create an encoding for each flow.

Besides normalization and feature selection, there are techniques that create an encoding for numerical feature values [10, 21, 29]. These methods divide the range of a numerical feature into discrete bins and encode each feature value based on based on its correspond-

ing bin. However, these encoding techniques ignore the frequencies of the features values and the context in which they occur.

Several studies compute aggregated features from the existing NetFlow features and use these to train ML algorithms [9, 17, 26, 27]. These aggregates can reveal new information from the dataset, potentially enhancing the training of a ML model. In our work, we also compute new aggregated features to capture the context in which specific feature values occur and use solely these new features to train a model. Compare to the previous studies [9, 17, 26, 27], we do compute fewer aggregated features and ensure there is no (semantic) overlap among these features. Additionally, these studies do not explicitly use frequencies of the feature values as an aggregated feature, nor do they apply an encoding to the (numerical) feature data.

2.4 Intuition Behind ENCODE

We believe the context and frequency of a unique feature value provide valuable information about the type of network behavior occurring in a flow. To illustrate this, consider the following example: suppose we have extracted the frequencies of the following byte values from NetFlow data: 37, 39, 80, 81, and 37548. The frequencies of the byte values are presented in Table 2.1. Based on the extracted frequencies, byte values 80 and 81 differ only by one byte in size, yet their frequencies vary substantially. This substantial difference likely indicates that they are generated by different type of software services and should therefore be treated differently by ML methods.

Using a NetFlow preprocessing technique such as the one employed by Camacho et al. [10], the range of the byte values can be divided into three regions; 37 and 39, 80 and 81, and 37548. Despite the substantial difference in the frequencies of byte values 80 and 81, they are placed within the same region. This causes ML algorithms to treat these two values as similar.

In contrast, ENCODE divides the range of byte values based on their corresponding frequencies. Byte values with a very low frequency are placed together into one region, while byte values 80 and 81 are each assigned to their own separate region. This results into three different regions: the first region contains bytes values 37, 39, and 37548. The second region contains solely byte value 80, and third region contains solely byte value 81. With this separation, ML algorithms can now treat bytes values 80 and 81 as two distinct values.

Table 2.1: Frequencies of byte values 37, 39, 43, 80, and 81. Frequencies are extracted from a subset of flows in the UGR-16 dataset.

Byte Value	Frequency
37	1
39	4
80	24771
81	3158
37548	4

2.5 Encoding Algorithm

In this section, we describe our encoding algorithm (ENCODE) for NetFlow data. We begin by explaining how the context is computed for each unique flow feature value and how this is used to create its encoding. Then we provide the implementation details of our encoding algorithm. ENCODE is publicly available at our GitHub repository [3]. ENCODE takes as input a set of numerical (non-categorical) features with their corresponding values. Its output is a mapping between each feature value and its corresponding encoded feature value.

2.5.1 Computing the Context

Our method for computing the context of values from the provided input feature is inspired by the work of Word2Vec, particularly the Count Bag of Words (CBOW) method [24]. For each value v of an input (numerical) NetFlow feature f , we compute the context in which v occurs by computing the frequencies of the values that occurred immediately before and after v . The intuition is that these frequencies capture the behavior of v within the network i.e. which other feature values v co-occurs with and how often.

The computed frequencies are used to generate a vector for value v . Before computing the context frequencies for each value v , the flows are grouped by connection (i.e. grouped by the pair of source and destination hosts). This ensures the context of v is not influenced by other flows co-occurring in the same network. As a result, the context represents the surrounding flows generated by similar software services.

2.5.2 Creating the Encoding

From the context generation, we obtain vectors representing the co-occurrence frequencies of feature values. We use the Euclidean distance to determine whether two feature values v and v' occur in similar contexts. A smaller distance between two vectors indicates a higher similarity in the contexts of the two corresponding feature values. This enables us to create groups of feature values that are contextually similar.

The encoding for the feature values is created by clustering the vectors of these values, with the assigned cluster labels used as their encoding. Thus, the encoding for v is the corresponding cluster label that was assigned to its vector. Feature values with similar frequencies and contexts are assigned to the same cluster and receive the same encoding. For the clustering of the vectors, we employ the KMeans algorithm because it allows us to control the number of clusters that can be formed, thereby also giving us control over the size of the encoding for each input feature. While we have opted to use KMeans as our clustering algorithm, other clustering algorithms can also be used to compute the encoding. ENCODE is modular by design, allowing the clustering algorithm to be easily replaced with one that better fits the user's needs.

2.5.3 An Illustrative Example

We now provide a small illustrative example to demonstrate how our encoding is computed for NetFlow data. The flows used in this example are a very small subset of flows extracted from the UGR-16 dataset, one of the datasets used in our evaluation. Figure 2.1 presents the small subset of flows used in this example. More information on this dataset

timestamp	id:src_ip	id:dst_ip	num_bytes
827146377	100.10.104.43	42.219.155.90	3149
827146459	100.10.104.43	42.219.155.90	977
827146506	100.10.104.43	42.219.155.90	64
827146532	100.10.104.43	42.219.155.90	64
827146573	100.10.104.43	42.219.155.90	2852
827146631	100.10.104.43	42.219.155.90	165
827146712	100.10.104.43	42.219.155.90	1540
827146786	100.10.104.43	42.219.155.90	3169
827146861	100.10.104.43	42.219.155.90	2909
827146936	100.10.104.43	42.219.155.90	1224

Figure 2.1: A small subset of NetFlow data extracted from the UGR-16 dataset.

Byte Value	Frequencies of the direct previous byte values									Frequencies of the direct next byte values								
	64	165	977	1224	1540	2852	2909	3149	3169	64	165	977	1224	1540	2852	2909	3149	3169
64	1	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
165	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0
977	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0
1224	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
1540	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
2852	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
2909	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0
3149	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
3169	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0

Figure 2.2: Frequencies of the direct previous and next bytes values for each of the unique byte values that are present in Figure 2.1.

can be found in Section 2.6. In this example, we demonstrate the computation of the context and the encoding for unique byte values.

The first step of ENCODE is to find the unique byte values. From the byte values that are shown in Figure 2.1, we observe a total of nine unique byte values: 64, 165, 977, 1224, 1540, 2852, 2909, 3149, and 3169. Note that all data in this example comes from the same connection. Next, we compute the frequencies of the direct previous and next byte values for each of these nine byte values. The computed frequencies are shown in Figure 2.2. Each row in Figure 2.2 represents the vector computed for the corresponding unique byte value. These vectors are then clustered together using the KMeans algorithm and the assigned cluster labels serve as the encoding for the corresponding unique byte values. Figure 2.3 presents the assigned cluster labels for each unique byte value.

2.5.4 Implementation Details

Figure 2.5 provides a high-level overview of ENCODE’s data processing pipeline. In the following paragraphs, we describe in detail the challenges we faced when constructing the

Byte Value	Cluster Label (Encoding)
64	1
165	4
977	1
1224	0
1540	0
2852	3
2909	0
3149	0
3169	2

Figure 2.3: Cluster labels assigned to the unique byte values used in the example. The assigned cluster labels are used as the encoding for the unique byte values. In this example, we have used five clusters to cluster the vectors.

co-occurrence matrix within ENCODE and how we addressed the challenges. Furthermore, we also describe how we cluster the rows of the co-occurrence matrix and how we evaluate the quality of the formed clusters for the encoding.

Generalized Vector Structure

As shown in our example, a vector is generated for each unique feature value, v . Since these vectors are clustered using KMeans, the sizes of the vectors must be uniform. Additionally, all vectors must store the same type of information (co-occurrence frequency values corresponding to a particular feature value) in the exact same positions.

As illustrated in Figure 2.2, the first half of each vector (highlighted in orange) represents the frequencies of feature values occurring directly before v , while the second half (highlighted in blue) represents the frequencies of feature values occurring directly after v . The size of each vector is equal to twice the number of unique feature values, ensuring that all vectors share the same structure and size. However, this structure comes with several limitations:

- It can lead to the creation of many sparse vectors.
- The size of vectors depends on the number of unique values for a given feature. As the number of unique values increases substantially, the size of the vectors also expands proportionally.
- We cannot capture the behavior in which a feature value v co-occurs frequently with itself. It is common to observe v co-occur with itself in NetFlow data, indicating repetitive software behavior.

The last limitation is somewhat nuanced. The co-occurrences of a value v with itself are present in the table. For instance, for value 64 in Figure 2.2, the co-occurrences with itself are stored in the column labeled 64. The challenge is that, for different data rows, these self-contexts (co-occurrence with itself) are located in different columns. Consequently, rows with only self-contexts would have a large distance from rows showing a similar

Byte Value	Logarithm of Frequency	Frequencies of bins for the direct previous values						Frequencies of bins for the direct next values					
		SELF_PREV	BIN 1	BIN2	BIN3	...	BIN 10	SELF_NEXT	BIN 1	BIN2	BIN3	...	BIN 10
Byte value 1	...	...	...	...	...	...	...	...	...	...	...	...	...
Byte value 2	...	...	...	...	...	...	...	...	...	...	...	...	...
Byte value 3	...	...	...	...	...	...	...	...	...	...	...	...	...
.....	...	...	...	...	...	...	...	...	...	...	...	...	...
Byte value N	...	...	...	...	...	...	...	...	...	...	...	...	...

Figure 2.4: Generalized vector structure used for each unique feature value within ENCODE.

pattern. In our view, such repetitive behavior reflects similar software activity (behavior) and similarity should not depend on the exact feature values used in the flow.

To address the last limitation, we add two separate columns to store the “self-frequency” i.e., one column for v occurring directly before itself and another for v occurring directly after itself. In addition to storing the self-frequency, we have also included an additional entry that stores the logarithm of the frequency of v (marked in red in Figure 2.2). This way the frequency of v also influences the distance computation between vectors. Furthermore, this entry helps to differentiate features that have similar previous (orange) and next (blue) values but different frequencies.

To resolve the first two limitations, we divide the space of the previous and next values into ten bins each using percentiles. This results in a vector of length 20 for each feature value v . A benefit of this solution is that it enables us to control the size of the vectors by adjusting the number of bins used to divide the space of the previous/next values. Figure 2.4 presents the final generalized structure used in the implementation of ENCODE.

Cluster Evaluation within ENCODE

To cluster the rows of the co-occurrence matrix constructed by ENCODE, we use the KMeans implementation provided by the scikit-learn Python package [5]. We opted for KMeans as it allows us to control the number of clusters, thereby fixing the size of our encoding. Since KMeans selects the initial centroids at random, each run could lead to different final clusters. ENCODE therefore runs KMeans ten times and selects the best run to create the encoding for the input feature. The Silhouette Coefficient is used to evaluate the quality of the formed clusters (of the vectors). To compute the Silhouette Coefficient, we make use of scikit-learn’s Python package [7].

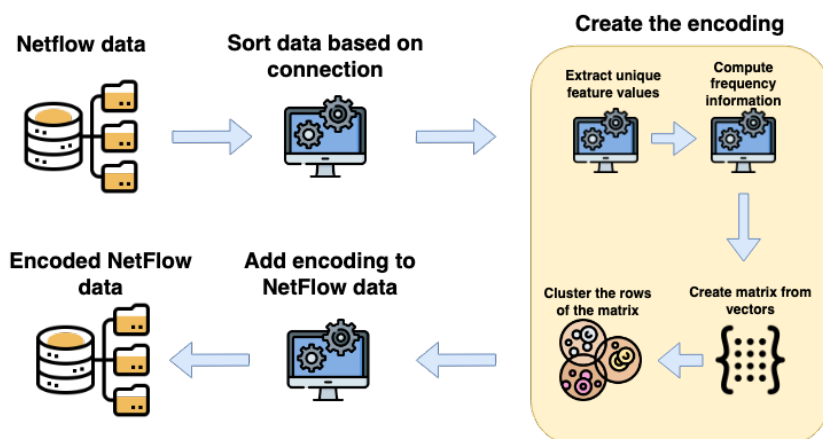


Figure 2.5: High-level overview of our data processing pipeline. The pipeline shows how the encoding is created for the given NetFlow input data using our encoding algorithm.

2.6 Evaluation of ENCODE

In this section, we describe how we evaluate our encoding in the context of network anomaly detection. The goal of our evaluation is to show that ML algorithms leverage our encoding to extract contextual information from NetFlow data, enabling them to learn more accurate models for network anomaly detection. To this end, we train several ML models in an unsupervised manner to detect various network attacks. For each selected ML algorithm, we train two models; one that includes our encoding as part of the input features and one that does not. The performance of these two models is then compared. Specifically, we compare their performance using Balanced Accuracy and the F_1 score

2.6.1 Selection of ML Models

To evaluate the effectiveness of ENCODE on learning ML models for network anomaly detection, we selected several ML methods for unsupervised network anomaly detection.

State Machine

A state machine, formerly known as a finite state automaton, is a mathematical model used to represent the sequential behavior of a system. These models can be learned using a passive-learning approach that leverages state-merging algorithms. Since state machines inherently capture sequential behavior, they are well-suited to modelling NetFlow. Prior works have demonstrated the effectiveness of using state machines for network anomaly detection [16, 22, 29]. In the context of unsupervised network anomaly detection, we generate fixed-size behavioral sequences exclusively from benign flows to learn a state machine. The resulting model is used to detect any sequential behavior at test time deviating from the learned behavior. To learn a state machine from NetFlows, we use Flexfringe [33], a framework that supports various merging heuristics for the learning of state machines.

Isolation Forest

Isolation Forest (IF) is a well-known unsupervised anomaly detection method. The core idea is anomalies are easier to isolate because these tend to be farther from other data points within the feature space, resulting in shorter path lengths to the root of the tree. Consequently, instances with shorter average path lengths to the root nodes of the trees are considered more anomalous [19]. In contrast to state machines, IF does not model sequential behavior and is specifically designed to detect point anomalies. For our evaluation, we use the implementation of IF provided by the scikit-learn Python package [6].

Local Outlier Factor

Local Outlier Factor (LOF) is another well-known anomaly detection method. The core idea behind LOF is that instances grouped within a dense neighbourhood are considered to be benign, while those in sparsely populated neighborhoods are deemed anomalous. For our evaluation, we use the implementation of LOF provided by the scikit-learn Python package [8].

Deep Neural Network

In recent years, deep neural networks have gained significant popularity for anomaly detection. DeepLog is a well-known work that uses an LSTM to detect anomalies from logs

collected from a system [14]. Much like state machines, LSTMs are used to model sequential behaviors that occur within a system. Furthermore, only benign data is used to train the model, which is then employed to detect sequential behaviors deviating from those observed during training. For our evaluation, we use the PyTorch implementation of DeepLog provided by van Ede et al. [2].

2.6.2 Comparison Between Encoding Methods

To evaluate the effectiveness of ENCODE, we have selected two encoding methods to serve as baselines: an existing percentile-based encoding and a naive frequency-based encoding. The percentile-based encoding divides the range of a given input feature into b bins, and the encoding of a feature v corresponds to the bin that it falls into. The frequency-based encoding creates an encoding based on the frequency of the feature values. If a feature value v occurs at least x times, then the raw value is used as its encoding. Otherwise, it is grouped into a low-frequency bin along with other feature values that do not meet the threshold of x . These low-frequency bins are further subdivided into multiple bins using the percentile-based encoding method.

2.6.3 Dataset Selection

To assess the generalizability of ENCODE across different network architectures, we selected three datasets to train models for anomaly detection. The first dataset, the AssureMOSS dataset, is a new dataset created for the AssureMOSS project [1]. It contains both benign and malicious NetFlows collected from a Kubernetes cluster running multiple microservice applications [11]. The architecture of the cluster is shown in Figure 2.6. To generate benign flows for this dataset, 30 participants were recruited to use the microservice applications deployed within the cluster [12].¹ Furthermore, three different attack scenarios were constructed (DoS attack, lateral movement, and malicious code deployment) using a threat matrix proposed by Microsoft [23]. These attack scenarios were launched against the cluster, while the participants were actively using the applications.

As the second dataset, we selected a well-known dataset widely used in previous studies: the CTU-13 dataset, created by Garcia et al. [15]. This dataset contains both benign and malicious flows. Benign flows were generated by real users on a university network, while the malicious flows were generated by various botnet malware. In total, the dataset comprises thirteen captures collected from different botnet samples.

For the third and final dataset, we selected the UGR-16 dataset created by Maciá-Fernández et al. [20]. We chose this dataset because it contains a substantial collection of NetFlows data, totaling to 100 GB. The flows were collected over a period of 6 months from a Spanish ISP. The benign flows originated from real users accessing the internet for various purposes, while the malicious flows were produced by various types of network attacks.

2.6.4 Feature Selection

We run ENCODE on the same feature set across all three datasets: the protocol used in a flow, the total number of bytes sent in a flow, the total number of packets sent in a

¹This study is presented in Chapter 3.

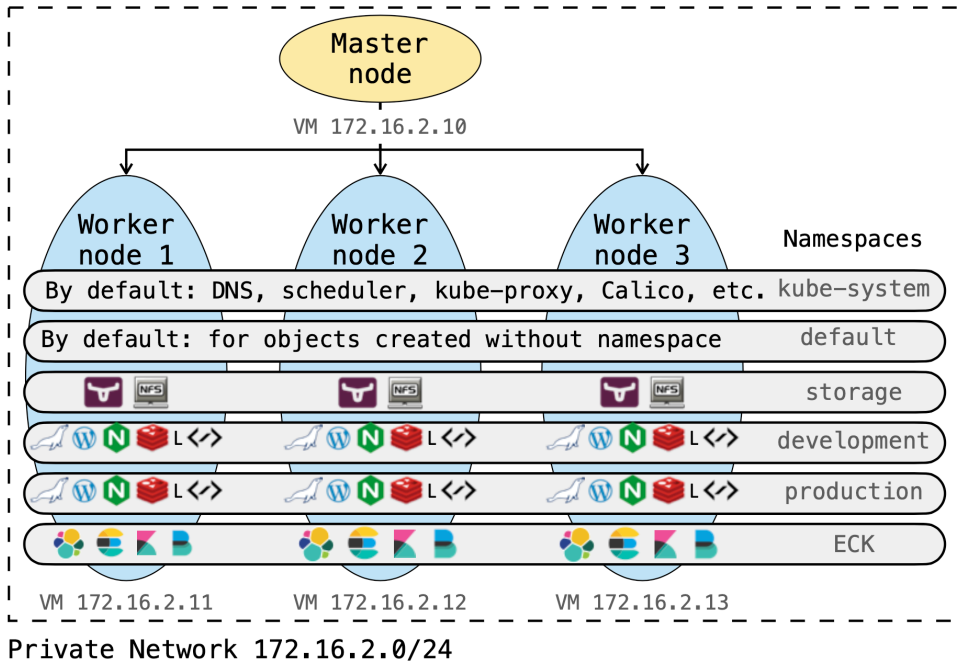


Figure 2.6: Architecture of the Kubernetes Cluster used to generate the AssureMOSS dataset. Image is taken from the work done by Cao et al. [12].

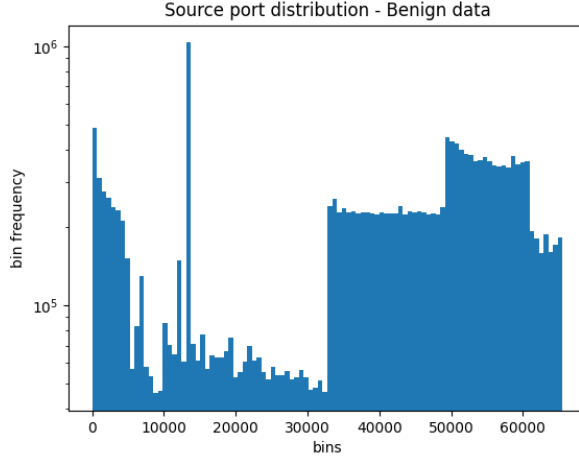
flow, and the duration of the flow. Using only these four features, ENCODE does not rely on operating system- or network-specific features, making it applicable to various types of networks. We exclude the source and destination ports as a feature for ENCODE as these tend to be operating system or network-specific. Additionally, port information can sometimes act as a spurious feature, i.e., a feature that correlates with the class label without being truly indicative of the class.

As an example, we use the CTU-13 dataset to illustrate how the source port is a spurious feature of the dataset. We believe that the source port in the CTU-13 dataset leaks information about the flow’s label and should not be blindly used to train a model. Garcia et al. generated the malicious flows using virtual machines running Windows XP. In this operating system, the default ephemeral ports range from 1025 to 5000 [4].

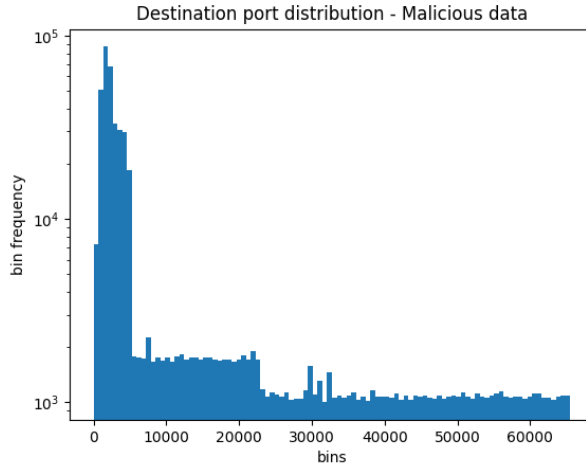
By plotting the distribution of the protocols for both benign and malicious flows, we observe a clear difference in the distributions of the source ports. Figure 2.7 presents the source ports distribution for benign and malicious flows across all thirteen scenarios. Within the malicious flows, we observe a substantial portion of the flows appear to use a source port lower than 6000. The top three ports used in malicious flows are 2077, 1025, and 2079. All three source port numbers fall within the range used by the virtual machines. We believe that by training a model using the source port causes it to identify whether a flow originates from a virtual machine rather than determining if it is malicious or benign. Consequently, using the source port to train a model would falsely lead to a better model

performance. Therefore, we opted not to run ENCODE on the source and destination ports of the flows.

2



(a) Source port distribution of benign flows



(b) Source port distribution of malicious flows

Figure 2.7: Source port distribution in the CTU-13 dataset.

2.6.5 Encoding Train & Test Data

As we perform anomaly detection in an unsupervised manner, we use only benign flows as training data for the ML models. The test data can contain both benign and malicious flows. Below, we outline how we selected the training and test sets for each dataset.

For the CTU-13 dataset, we used the split proposed by the original authors of the dataset to create our training and test sets [15]. To construct the training set, we randomly sampled benign flows across all training scenarios, resulting in approximately 720000 benign flows. Similarly, we randomly sampled benign and malicious flows across the test scenarios, resulting in a test set containing approximately 522000 flows.

As for the UGR-16 dataset, Maciá-Fernandez et al [20]. have provided the split between training and test data. We followed the same procedure as with CTU-13 to construct our training set. Specifically, we randomly sampled benign flows from the listed calibration data, resulting in approximately 600000 flows. As for our test set, we randomly sample benign and malicious data from the listed test data, resulting in a test set containing approximately 664000 flows.

For the AssureMOSS dataset, Cao et al [11]. provided the split between training and test data. The training data contains solely benign flows and has approximately 120000 flows. The test data contains both benign and malicious flows and has approximately 275000 flows. We directly use the training data as our training set and the test data as our test set.

After creating the training and testing sets for each dataset, the training set, along with the selected features, is provided as input to ENCODE to learn an encoding for the corresponding dataset. The resulting encoding is then applied to the input features of the corresponding dataset and the encoded features are added as new columns to the training and testing sets.

2.6.6 Creating Sequences For Sequential Models

Since we are using two ML models that learn the sequential behavior from sequences, we need to transform the NetFlow data into sequences (or traces). Each sequence consists of a list of symbols, where each flow is transformed into a symbol. A symbol for an arbitrary flow is constructed by concatenating the encoded values of the selected features into a single string in the following format: *PROTOCOL_BYTES_PACKETS_DURATION*. Note that the protocol feature is already a categorical feature and does not need to be encoded using ENCODE. Instead, we apply an integer encoding that converts each protocol into a numerical value. To generate traces, we use a sliding window of length 20 on the symbols constructed from flows of the same connection (i.e. combination of source and destination host). Traces shorter than the sliding window size are discarded, as we deem these insufficient to provide meaningful information for model learning.

2.7 Results

In this section, we empirically demonstrate the effectiveness of ENCODE. Each selected dataset is encoded using each encoding method and following the aforementioned encoding procedure. The encoded training flows for each dataset are subsequently used to train the selected ML models for unsupervised anomaly detection. To evaluate the effectiveness of ENCODE, we compare the performance of the ML models trained on the encoded feature values provided by ENCODE with the performance of the ML models trained on encoded feature values provided by the baseline encoding methods. Figures 2.8-2.16, and Table 2.2 present the performance results of the ML models trained on feature values en-

coded using the three encoding methods. The performance results represent the average performance computed over ten independent runs for each ML model. The only exception is the DeepLog model, for which we were unable to compute average results due to the repeated out-of-memory issues.

2

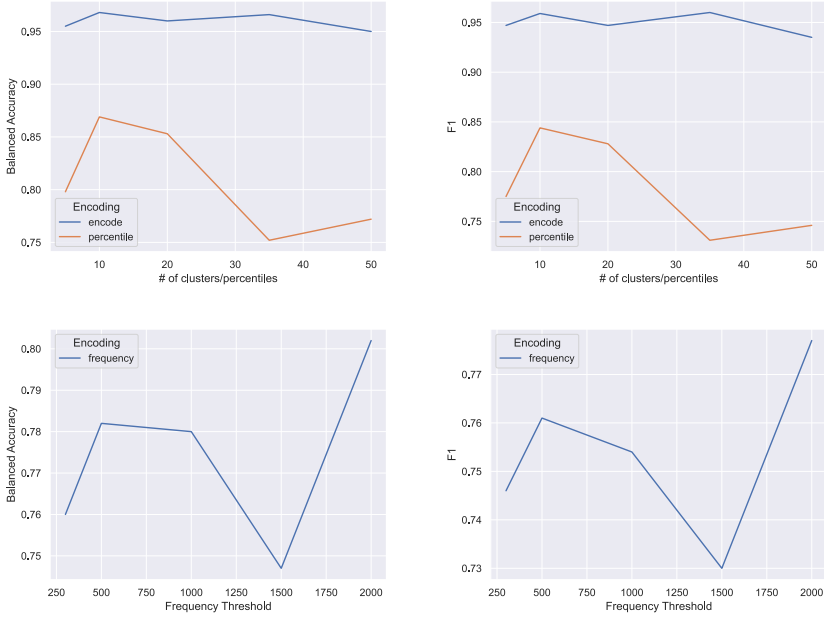


Figure 2.8: Performance results of State Machine (for each encoding method) in detecting anomalies on the AssureMOSS dataset. The figure presents the model's average results over 10 runs.

Table 2.2: Partial performance results of DeepLog on AssureMOSS dataset. Performance results are computed from a single run (due to repeated out-of-memory issues).

Encoding	Balanced Accuracy	F_1
ENCODE (5 clusters)	0.782	0.734
Percentile (5 bins)	0.482	0.332
Frequency (Threshold = 300)	-	-

2.7.1 Overall Results

When we inspect the performance results presented in Figures 2.8-2.16 and Table 2.2, we observe that the models achieve considerably higher balanced accuracy and F_1 scores when ENCODE was used to preprocess the NetFlow data. This empirically demonstrates that the models can leverage the context defined by our encoding to learn a better context-

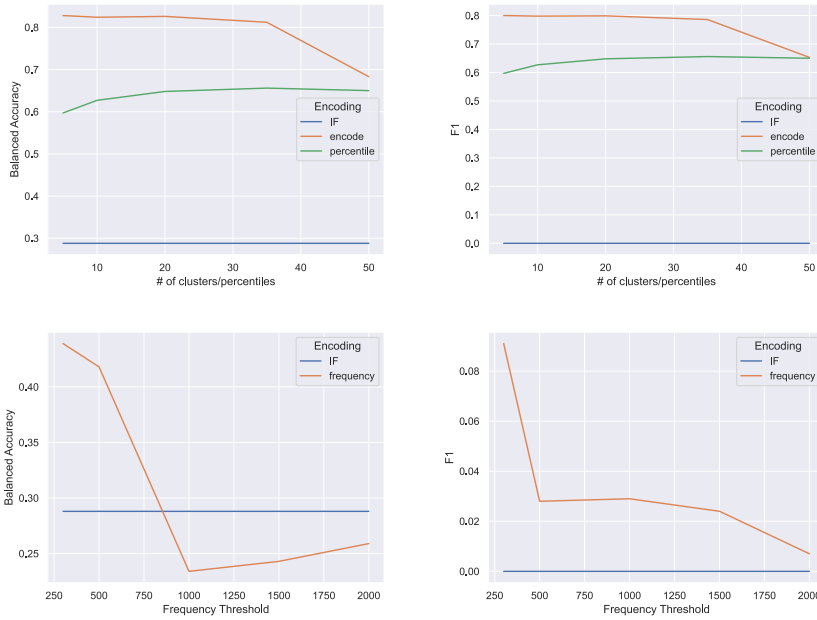


Figure 2.9: Performance results of IF (for each encoding method) in detecting anomalies on the AssureMOSS dataset. The figure presents the model’s average results over 10 runs.

tual correlation between the feature values of the NetFlow data, thereby improving their ability to detect anomalies within the data.

Although the percentile-based encoding is a standard used in prior works, it performs in almost all cases worse than ENCODE. We believe this low performance is due to the fact that flow similarity is computed based on the similarity between feature values and flows with similar (feature) values do not necessarily mean that they are the same. Additionally, the percentile-based encoding does not store any contextual information on how these values co-occur.

While sequential ML models could theoretically learn such patterns, they still perform considerably better using ENCODE to preprocess NetFlow data. Since ENCODE stores contextual information within each encoded value, even non-sequential ML methods can capture some sequential information using our encoding. This effect is evident for the IF detector, which performs considerably better using our encoding than using the raw feature values. The low performance of the naive frequency-based encoding can be explained by the fact that it only stores information about frequently occurring feature values (based on a given threshold), while infrequent feature values are then binned together. Similar to percentile-based encoding, this method does not capture the context of how feature values co-occur, regardless of whether they are frequent or infrequent.

When comparing the different ML methods, we observe that LOF performs worse on the CTU-13 and UGR-16 datasets when either ENCODE or percentile-based encoding is used to preprocess the NetFlow data. In fact, LOF obtained better results when it was

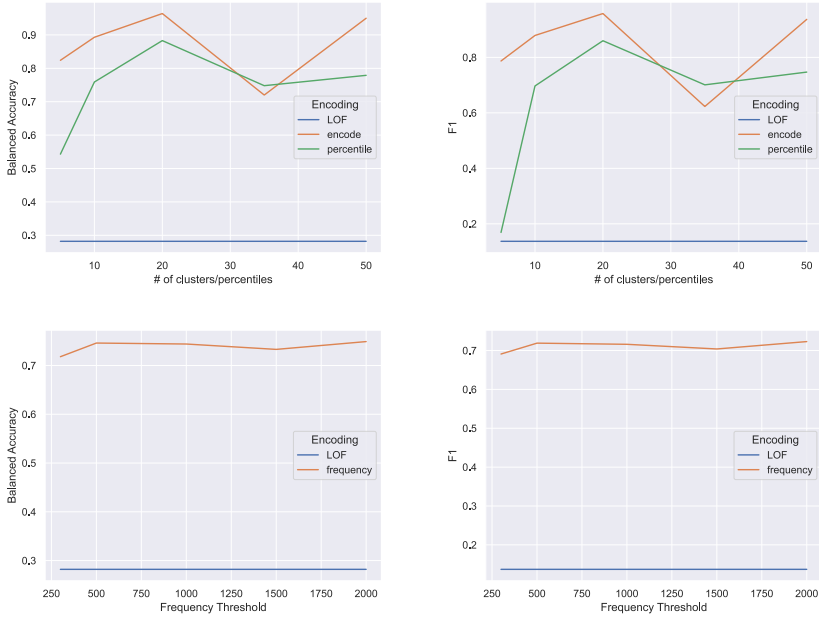


Figure 2.10: Performance results of LOF (for each encoding method) in detecting anomalies on the AssureMOSS dataset. The figure presents the model’s average results over 10 runs.

trained using the raw feature values. LOF treats the encoded values as integers and computes distances based on them. These distances are not very meaningful, which explains the poor performance. LOF is also the only method for which it was better to the naive frequency-based encoding to preprocess the NetFlow data (see Figure 2.13). The main reason for this observation is that the raw values remain intact for high frequencies, making the distances somewhat meaningful. Still, LOF performs considerably worse than the other two methods that can leverage our encoding in a meaningful way.

The results obtained using DeepLog are promising but are based on a single run (of one epoch) of the model on the AssureMOSS dataset. We only managed to get DeepLog working on the AssureMOSS dataset for the case where we used 5 clusters in ENCODE and 5 bins for the percentile-based encoding. We tested DeepLog both with and without GPU. In cases with a large number of clusters or bins, our machine ran out of memory. This issue is likely caused by the increase in the number of unique symbols introduced when the size of the clusters or bins increases (more clusters or bins means that we have more symbols). Although the results show that DeepLog achieves considerably better performance using ENCODE to preprocess the NetFlow data, future work is needed to determine the significance of this improvement.

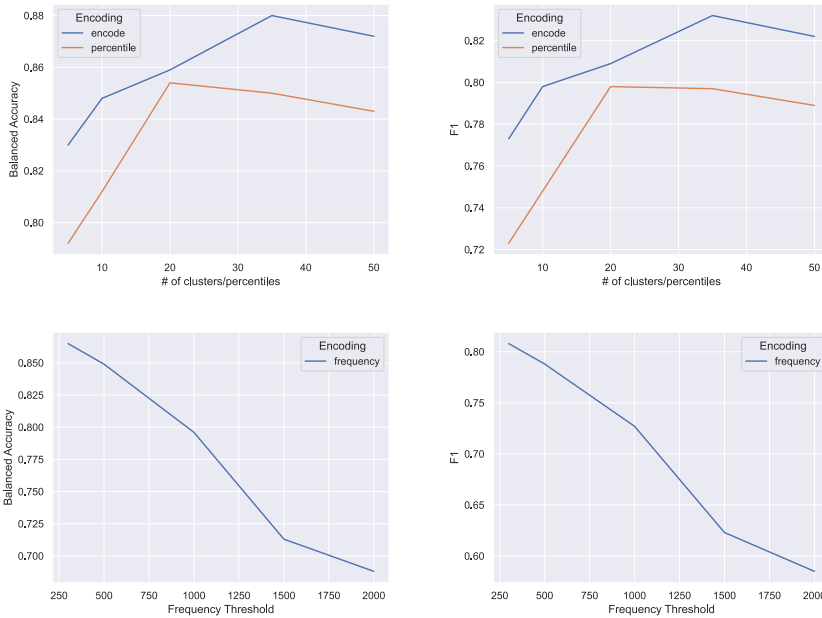


Figure 2.11: Performance results of State Machines (for each encoding method) in detecting anomalies on the CTU-13 dataset. The figure presents the model's average results over 10 runs.

2.7.2 The Effect of Cluster Number On Model Performance

A key parameter of ENCODE is the number of clusters used in KMeans clustering. The larger this number, the more distinctions are made between flows, but at a cost of increased complexity. Based on the performance results presented in Figures 2.8-2.16, we observe that increasing the number of clusters can have varying effects on the performance of the ML models: (1) it can increase the model's performance (see Figure 2.10, Figure 2.11 and Figure 2.14), (2) it can decrease the model's performance (see Figure 2.9, Figure 2.12 and Figure 2.15), or (3) have no considerable effect on the model's performance (see 2.8). The increase in performance appears to be more common with state machines, while the decrease in performance is more frequent in IFs. This highlights the importance of tuning this parameter within ENCODE to determine the optimal number of clusters for a given model to achieve a performance boost.

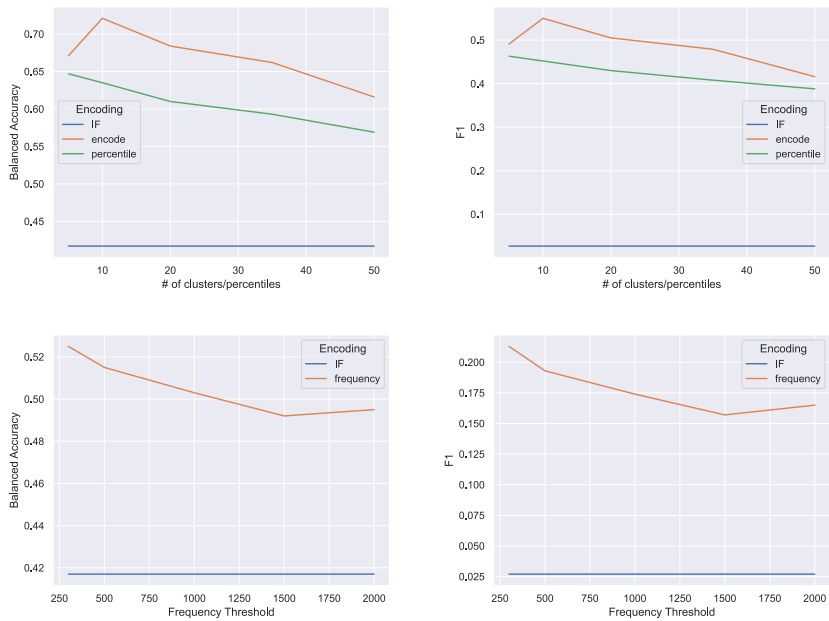


Figure 2.12: Performance results of IF (for each encoding method) in detecting anomalies on the CTU-13 dataset. The figure presents the model's average results over 10 runs.

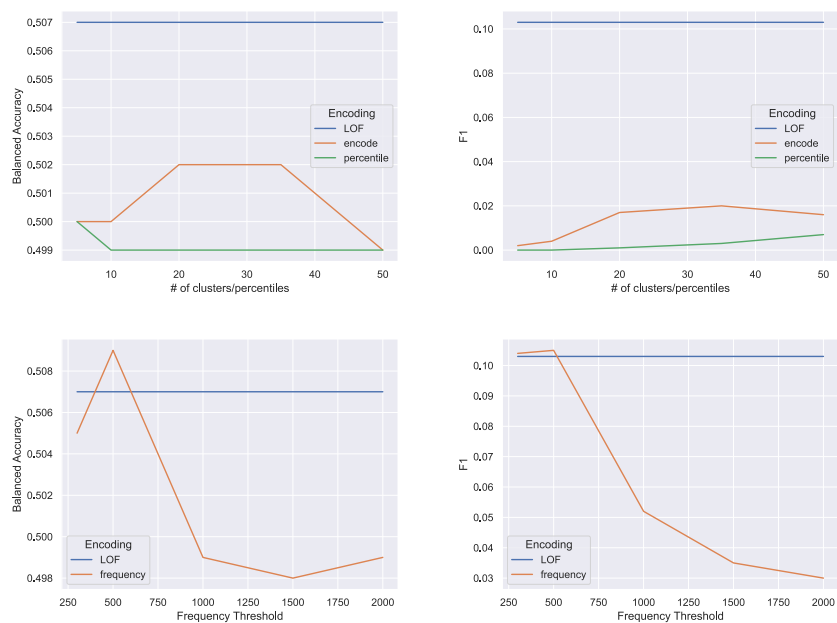


Figure 2.13: Performance results of LOF (for each encoding method) in detecting anomalies on the CTU-13 dataset. The figure presents the model’s average results over 10 runs.

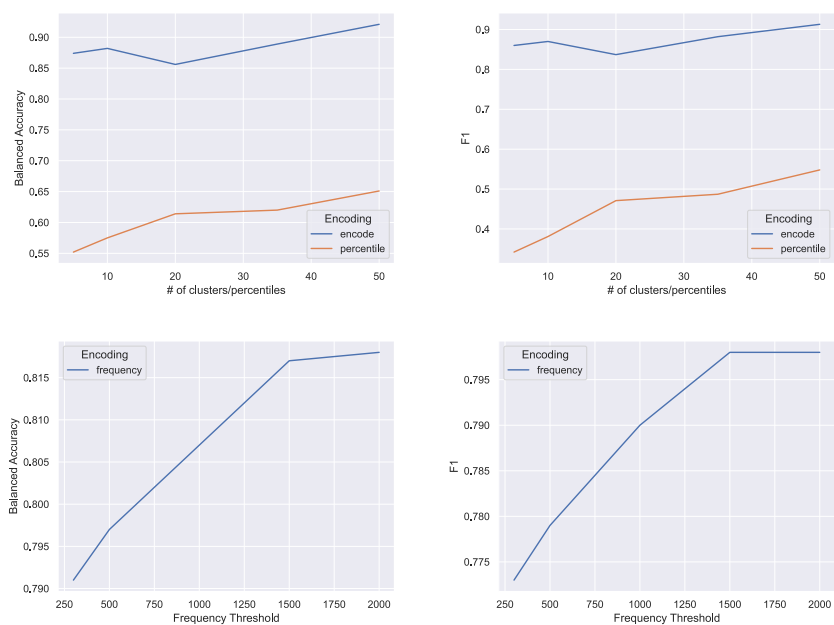


Figure 2.14: Performance results of State Machines (for each encoding method) in detecting anomalies on the UGR-16 dataset. The figure presents the model’s average results over 10 runs.

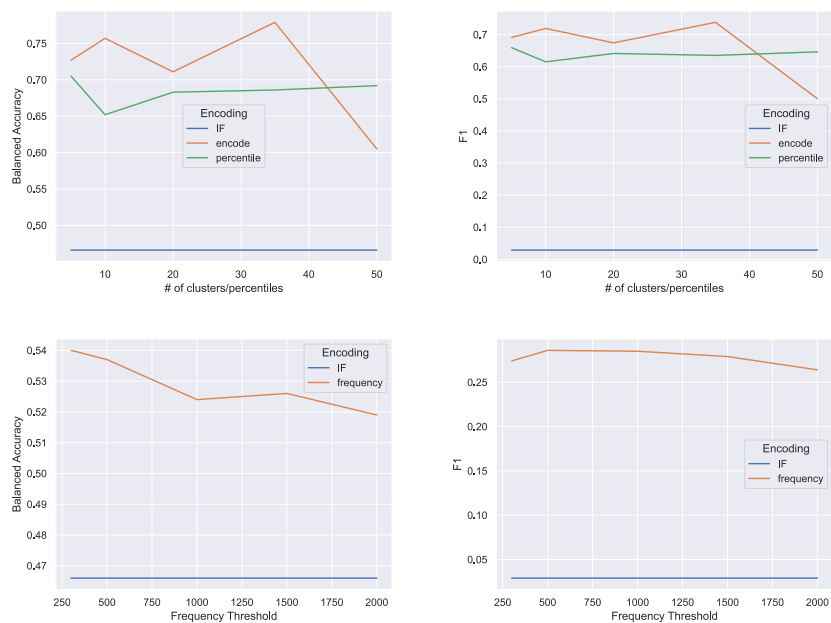


Figure 2.15: Performance results of IF(for each encoding method) in detecting anomalies on the UGR-16 dataset. The figure presents the model’ s average results over 10 runs.

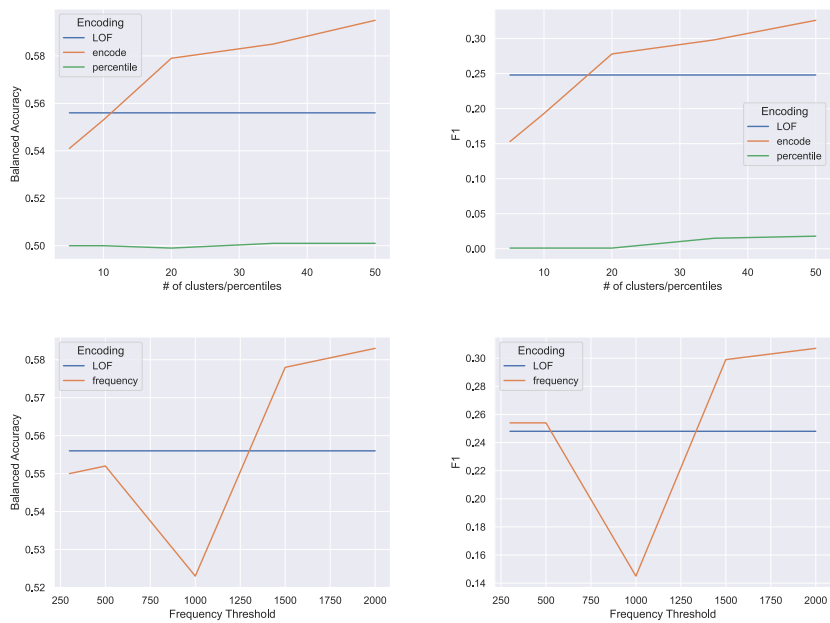


Figure 2.16: Performance results of LOF (for each encoding method) in detecting anomalies on the UGR-16 dataset. The figure presents the model's average results over 10 runs.

2.8 Use-case and robustness

From our experimental results, we observe using 35 clusters provides the best performance for state machines. As ENCODE was originally developed for state machine learning, we now present a use case demonstrating how ENCODE can be used to derive deeper insights into the model's predictions. To illustrate this, we display the output an analyst would see when they are running an anomaly detection system that utilizes state machines. The state machine produces (log)likelihood values, which the anomaly detection system can use to determine the degree of abnormality for an arbitrary trace. In our experiments, we use a fixed decision threshold, tuned on the training data, to count false/true positives and negatives. However, in real-world applications, we believe it is more effective to visualize the likelihood values over time rather than relying solely on threshold-based decisions. Figure 2.17 presents example visualizations of likelihood values generated for the UGR-16 dataset.

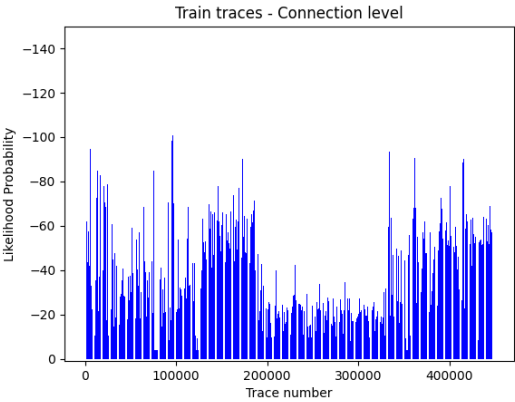
The likelihood values of the traces are visualized over time, enabling an analyst to quickly determine whether an alert corresponds to an attack or a false alarm. We often observe a large cluster of low-likelihood values (represented by tall vertical lines) when malicious traces occur. Additionally, we observe some anomalous traces in the train data simply because these connections are infrequent. The likelihood values for the test data tend to be lower (indicating a higher anomaly score) when new, infrequent flows occur. The figure also demonstrates that, although the performance values in our experiments are computed on a per-flow basis, it is more practical to visualize these values over time to observe (considerable) changes in network behavior. One could even argue that the method only triggers a limited number of false alarms, as indicated by the small number of blue peaks in the test data. Furthermore, it is evident that a single attack can produce multiple peaks (tall vertical lines) in the likelihood scores, even though just one peak would be sufficient to trigger an investigation

2.8.1 Experimental Results

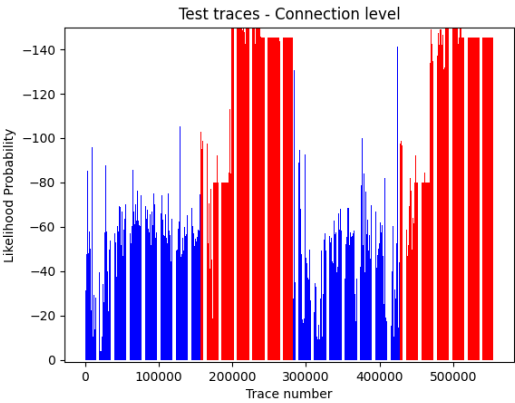
To test the robustness of ENCODE combined with state machines for anomaly detection, we present the performance of the state machine model after introducing various levels of perturbation to the number of bytes and packets in the test data (see Table 2.3). Three types of perturbations were applied in our experiments: (1) adding a fixed value to both features, (2) adding a random value between 0% and 50% of the original feature values, and (3) adding a random value between 50% and 80% of the original feature values. Applying these perturbations to the feature values causes them to be assigned to the cluster that groups all infrequent values (with low co-occurrence). Since these perturbed feature values are grouped together with other infrequent values, it becomes difficult to derive meaningful distances and contextual relationships from these values. Interestingly, the performance of the state machine model improves when perturbations are added to the features. This suggests that our encoding, when combined with a state machine model, is robust against attackers who attempt to randomly or structurally modify their network traffic to evade detection.

Table 2.3: Performance results of state machines on perturbed test data.

Amount of Perturbation	Balanced Accuracy	F_1	Precision	Recall
Fixed 10	0.915	0.860	0.754	1.0
Random between 0% and 50%	0.915	0.860	0.754	1.0
Random between 50% and 80%	0.915	0.860	0.754	1.0
No perturbations	0.850	0.797	0.754	0.844



(a) Likelihood values computed for training data of UGR-16 dataset.



(b) Likelihood values computed for test data of UGR-16 dataset.

Figure 2.17: Likelihood values computed by the state machine model for the UGR-16 dataset. Blue vertical lines represent benign traces, while red vertical lines represent malicious traces.

2.9 Conclusion & Future Work

We present a new encoding algorithm, ENCODE, that preprocesses NetFlow datasets for machine learning. ENCODE uses the frequency of feature values and the co-occurrence frequencies between feature values to compute the context in which the feature values appear. This context is then used to group feature values, creating an encoding for the given input feature. The intuition is that feature values that occur in similar contexts and with similar frequencies are indicative of similar software services and should, therefore, receive the same encoding label.

We empirically demonstrate the effectiveness of ENCODE by applying it to NetFlow data from three different datasets and training four different ML models to detect network anomalies in an unsupervised manner. For each dataset, we evaluated the performance results of the ML models and compared them to the performance achieved using two baseline encoding methods. The empirical results show that the ML models performed considerably better when trained on NetFlows preprocessed by ENCODE.

From our empirical results, the state machine model achieved the best performance in detecting network anomalies. For this method, we present a use case that demonstrates the type of information an analyst receives from the system and how it can be used to perform further analysis. Additionally, we show that this method is resilient to perturbations introduced by attackers attempting to evade detection. In fact, such perturbations make the data appear more anomalous, resulting in higher detection rates.

In this work, we have applied ENCODE to NetFlow data. In future work, we aim to evaluate ENCODE on other types of data, such as software log data. We believe that our encoding could also be beneficial for preprocessing software log data, and it would be valuable to investigate its effectiveness on additional data types.

Furthermore, it would be interesting to evaluate ENCODE on a broader range of ML models. For instance, although our primary focus is unsupervised learning, we believe our encoding could also be useful in supervised learning tasks.

Finally, we are developing a streaming version of ENCODE that can be deployed in real-time with NetFlow collectors.

References

- [1] AssureMOSS.
- [2] DeepLog - PyTorch implementation of Deeplog: Anomaly detection and diagnosis from system logs through deep learning. <https://github.com/Thijsvanede/DeepLog>.
- [3] ENCODE: Encoding NetFlows for Network Anomaly Detection. <https://github.com/tudelft-cda-lab/ENCODE>.
- [4] Service overview and network port requirements - Windows Server | Microsoft Docs. <https://docs.microsoft.com/en-us/troubleshoot/windows-server/networking/service-overview-and-network-port-requirements>.
- [5] sklearn.cluster.KMeans —scikit-learn 1.1.1 documentation. <https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html>.
- [6] sklearn.ensemble.IsolationForest —scikit-learn 1.1.2 documentation. <https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.IsolationForest.html>.
- [7] sklearn.metrics.silhouette_score —scikit-learn 1.1.2 documentation. https://scikit-learn.org/stable/modules/generated/sklearn.metrics.silhouette_score.html.
- [8] sklearn.neighbors.LocalOutlierFactor —scikit-learn 1.1.2 documentation.
- [9] Abdulghani Ali Ahmed, Waheb A. Jabbar, Ali Safaa Sadiq, and Hiran Patel. Deep learning-based classification model for botnet attack detection. *Journal of Ambient Intelligence and Humanized Computing*, 1:1–10, mar 2020.
- [10] Jose Camacho, Gabriel Macia-Fernandez, Noemi Marta Fuentes-Garcia, and Edoardo Saccenti. Semi-supervised multivariate statistical network monitoring for learning security threats. *IEEE Transactions on Information Forensics and Security*, 14(8):2179–2189, aug 2019.
- [11] Clinton Cao and Agathe Blaise. AssureMOSS Kubernetes Run-time Monitoring Dataset, 2022.
- [12] Clinton Cao, Agathe Blaise, Sicco Verwer, and Filippo Rebecchi. Learning state machines to monitor and detect anomalies on a kubernetes cluster. In *Proceedings of the 17th International Conference on Availability, Reliability and Security, ARES '22*, New York, NY, USA, 2022. Association for Computing Machinery.
- [13] Cisco Systems. Cisco IOS NetFlow - Cisco. <https://www.cisco.com/c/en/us/products/ios-nx-os-software/ios-netflow/index.html>, 2018.
- [14] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, pages 1285–1298, 2017.

- [15] S. Garcia, M. Grill, J. Stiborek, and A. Zunino. An empirical comparison of botnet detection methods. *Computers & Security*, 45:100–123, sep 2014.
- [16] Gudmund Grov, Wei Chen, Marc Sabate, and David Aspinall. Towards intelligible robust anomaly detection by learning interpretable behavioural models. In *Vol. 12 (2019): NISK 2019; Proceedings of the 12th Norwegian Information Security Conference*, NISK. Akademika, November 2019. 12th Norwegian Information Security Conference : Co-located with NIKT, NISK 2019 ; Conference date: 25-11-2019 Through 27-11-2019.
- [17] Mohammad Hashem Haghighat and Jun Li. Intrusion detection system using voting-based neural network. *Tsinghua Science and Technology*, 26(4):484–495, aug 2021.
- [18] Xavier Larriva-Novo, Mario Vega-Barbas, Víctor A. Villagrà, Diego Rivera, Manuel Álvarez-Campana, and Julio Berrocal. Efficient Distributed Preprocessing Model for Machine Learning-Based Anomaly Detection over Large-Scale Cybersecurity Datasets. *Applied Sciences* 2020, Vol. 10, Page 3430, 10(10):3430, may 2020.
- [19] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. Isolation forest. In *2008 Eighth IEEE International Conference on Data Mining*, pages 413–422, 2008.
- [20] Gabriel Maciá-Fernández, José Camacho, Roberto Magán-Carrión, Pedro García-Teodoro, and Roberto Therón. UGR ‘16: A new dataset for the evaluation of cyclostationarity-based network IDSs. *Computers and Security*, 73:411–424, mar 2018.
- [21] Roberto Magán-Carrión, Daniel Urda, Ignacio Díaz-Cano, and Bernabé Dorronsoro. Towards a Reliable Comparison and Evaluation of Network Intrusion Detection Systems Based on Machine Learning Approaches. *Applied Sciences* 2020, Vol. 10, Page 1775, 10(5):1775, mar 2020.
- [22] Petr Matoušek, Vojtěch Havlena, and Lukáš Holík. Efficient modelling of ics communication for anomaly detection using probabilistic automata. In *2021 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, pages 81–89, 2021.
- [23] Microsoft. Threat matrix for Kubernetes. <https://www.microsoft.com/security/blog/2020/04/02/attack-matrix-kubernetes/>, 2020.
- [24] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. In *1st International Conference on Learning Representations, ICLR 2013 - Workshop Track Proceedings*. International Conference on Learning Representations, ICLR, jan 2013.
- [25] Nour Moustafa and Jill Slay. UNSW-NB15: A comprehensive data set for network intrusion detection systems (UNSW-NB15 network data set). *2015 Military Communications and Information Systems Conference, MilCIS 2015 - Proceedings*, dec 2015.
- [26] Quoc Phong Nguyen, Kar Wai Lim, Dinil Mon Divakaran, Kian Hsiang Low, and Mun Choon Chan. GEE: A Gradient-based Explainable Variational Autoencoder for Network Anomaly Detection. *2019 IEEE Conference on Communications and Network Security, CNS 2019*, pages 91–99, jun 2019.

- [27] Beny Nugraha, Anshitha Nambiar, and Thomas Bauschert. Performance Evaluation of Botnet Detection using Deep Learning Techniques. *Proceedings of the 11th International Conference on Network of the Future, NoF 2020*, pages 141–149, oct 2020.
- [28] University of New Brunswick. Datasets | Research | Canadian Institute for Cybersecurity | UNB. <https://www.unb.ca/cic/datasets/index.html>.
- [29] Gaetano Pellegrino, Qin Lin, Christian Hammerschmidt, and Sicco Verwer. Learning behavioral fingerprints from Netflows using Timed Automata. In *Proceedings of the IM 2017 - 2017 IFIP/IEEE International Symposium on Integrated Network and Service Management*, pages 308–316. Institute of Electrical and Electronics Engineers Inc., jul 2017.
- [30] Michal Piskozub, Riccardo Spolaor, and Ivan Martinovic. MalAlert: Detecting malware in large-scale network traffic using statistical features. In *Performance Evaluation Review*, volume 46, pages 151–154, 2019.
- [31] Shahab Shamshirband and Anthony T. Chronopoulos. A new malware detection system using a high performance-elm method. *ACM International Conference Proceeding Series*, jun 2019.
- [32] Duygu Sinanc Terzi, Ramazan Terzi, and Seref Sagiroglu. Big data analytics for network anomaly detection from netflow data. In *2nd International Conference on Computer Science and Engineering, UBMK 2017*, pages 592–597. Institute of Electrical and Electronics Engineers Inc., oct 2017.
- [33] Sicco Verwer and Christian A. Hammerschmidt. Flexfringe: A passive automaton learning package. In *Proceedings - 2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017*, pages 638–642, 2017.
- [34] Ibrahim Yilmaz and Rahat Masum. Expansion of Cyber Attack Data From Unbalanced Datasets Using Generative Techniques. 2019.
- [35] Tommaso Zoppi, Andrea Ceccarelli, Tommaso Capecchi, and Andrea Bondavalli. Unsupervised anomaly detectors to detect intrusions in the current threat landscape. *ACM/IMS Trans. Data Sci.*, 2(2), apr 2021.
- [36] Tommaso Zoppi, Mohamad Gharib, Muhammad Atif, and Andrea Bondavalli. Meta-learning to improve unsupervised intrusion detection in cyber-physical systems. *ACM Transactions on Cyber-Physical Systems*, 5(4), sep 2021.

3

3

Monitoring MSAs Deployed on Kubernetes For Anomalies

These days more companies are shifting towards using cloud environments to provide their services to their client. While it is easy to set up a cloud environment, it is equally important to monitor the system's runtime behavior and identify anomalous behaviors that occur during its operation. In recent years, the utilization of RNN and DNN to detect anomalies that might occur during runtime has been a trending approach. However, it is unclear how to explain the decisions made by these networks and how these networks should be interpreted to understand the runtime behavior that they model. On the contrary, state machine models provide an easier manner to interpret and understand the behavior that they model. In this work, we propose an approach that learns state machine models to model the runtime behavior of a cloud environment that runs multiple microservice applications. To the best of our knowledge, this is the first work that tries to apply state machine models to microservice architectures. The state machine model is used to detect the different types of attacks that we launch on the cloud environment. From our experiment results, our approach can detect the attacks very well, achieving a balanced accuracy of 99.2% and a F_1 score of 0.982.

The content of this chapter is based on the publication “Clinton Cao, Agathe Blaise, Sicco Verwer, and Filippo Rebecchi. 2022. Learning State Machines to Monitor and Detect Anomalies on a Kubernetes Cluster. In *Proceedings of the 17th International Conference on Availability, Reliability and Security (ARES '22)*. Association for Computing Machinery, New York, NY, USA, Article 117, 1–9. <https://doi.org/10.1145/3538969.3543810>”. This work is also part of two public reports from the AssureMOSS project (D4.1 and D4.3), which are available at [1] and [2], respectively.

3.1 Introduction

These days, more devices are being connected to the Internet. The Internet enables users to access data and services with high availability. There are a plethora of cloud services that allow users to store their data online on a cloud server. Dropbox¹ and Google Drive² are two well-known examples of such cloud services. More companies are shifting towards the usage of a cloud environment to provide their services to their customers. Setting up such an environment these days is also not difficult as there exist tools that can be used to quickly set up a cloud environment. A set of detailed instructions is provided by the tools to guide developers during the setup. Kubernetes (k8s)³ is an example of such a tool that provides instruction on how to set up a cluster and use it to deploy different services.

Though the tools make it easier for a company to deploy a cloud environment and use it to provide their services to their customer, it is equally important to keep this environment secure from any malicious actor that might want to gather private (confidential) data from the customers or take down the whole environment. In this respect, microservice-based applications must be monitored at runtime to identify anomalous behaviors that occur during their operations. This requires having a constantly updated view of the actions and events happening and then comparing them to an expected model of operation. Software logs and network traffic information such as NetFlow can be collected and monitored by the services running within the environment. The challenge here is to show how to deal with the large volume of software log or NetFlow data that are generated by the services.

In recent years, the utilization of Recurrent Neural Networks (RNNs) or Deep Neural Networks (DNNs) to detect anomalies that occur during the runtime of a software system has been a trending approach investigated by many researchers [9, 16, 21, 26]. A neural network can be trained with the software log data or NetFlow data produced by the applications to model the behavior of the applications. While it has been shown that using this approach can detect anomalies with high accuracy, it is not clear how to explain the classification decisions made by the neural network. Furthermore, it is also not clear how the neural network should be interpreted (and visualized) to understand the behaviors that are modelled. Not being able to understand or explain the decisions that are made by a model could lead a company to not adopt these models in practice as there might be legal/regulatory compliance requirements that they need to follow.

On the contrary, is it easier to understand the behavior that is modeled using a state machine similar to what is done for the design and analysis of software [13]. A state machine is a mathematical model used to model sequential behavior and can be learned automatically from the traces produced by any software system. Based on a given input sequence, one can understand in which state a complex system will end up. Hence, state machines provide insights into the inner working of a software system and the models can be used to verify whether the software system is behaving as it should be [7, 13]. Extending such a state machine approach to microservice architectures is currently an open challenge.

When it comes to the learning of state machines, there are two approaches: the active-learning approach and the passive-learning approach. In the former, the learning algo-

¹<https://www.dropbox.com/?landing=dbv2>

²<https://www.google.com/drive/>

³<https://kubernetes.io/>

rithm continuously queries the system to generate training data for the learning process. LearnLib [10] is a well-known framework that utilizes active learning to learn state machines. In the latter, the algorithm does not query the system to generate training data, but it simply observes the data that is produced by the system. The observations are then used for the learning of a state machine.

To detect the unusual behavior using state machines, a probabilistic state machine can be employed. The probabilistic state machine can compute the probabilities for the traces that were produced by the system. Traces with low probabilities are considered to be unusual and an alarm should be raised. One effective method for learning probabilistic state machines in the passive-learning approach is the state-merging heuristic method.

In this work, we provide a novel anomaly detection approach that uses a passive-learning approach to learn probabilistic state machines from the NetFlow data gathered from a k8s cluster. The state machines are used to model the normal network behavior of the services running with the cluster. Large deviations that are detected within the models are considered to be suspicious and an alarm should be raised. We show that we can use state machines to model the runtime behavior of a k8s cluster and detect attacks that were launched against the cluster.

The remainder of this chapter is structured as follows. Section 3.2 addresses related work and positions our work. Section 3.3 introduces different approaches to model attacks in k8s. Section 3.4 details the approach using the learning of probabilistic state machine models to detect anomalies in the cluster. Section 3.5 provides details on the microservice orchestration testbed that we designed, the multi-step attack scenarios deployed, and the creation of a labelled dataset. Section 3.6 presents the evaluation of the methodology, including the detection results that we obtained. Finally, Section 3.7 draws our conclusions and points out future work.

3.2 Related Work

In this section, we focus our attention on the relevant literature in the field of k8s security management and the general approaches, including state-merging algorithms.

3.2.1 Kubernetes Security Management

While simple applications may be composed of only a few tens of components, the largest ones, such as those operated by Netflix and Uber, can run hundreds or thousands of containers. Their composition leads to complex interaction patterns that can be a source of security issues. This complexity is well understood by professionals, who require a thorough security assessment of both k8s clusters and microservice behaviors.

By limiting the scope of our analysis to k8s, different approaches exist to monitor the run phase of a microservice lifecycle, with the objective to detect anomalous behaviors at runtime. Application security in runtime is very different from the build and deployment stages. Dynamic analysis is in charge of assessing the security level in a k8s cluster running in production. Compared to the analysis of monolithic applications, a microservice analysis must monitor a large number of services, each one of them running on different clusters possibly hosted on heterogeneous application platforms. Tools in this category also allow the user to define security policies, and prevent specific vulnerabilities to be

Tool	Company	Open-source	Type	Use eBPF
<i>Falco</i> [25]	Sysdig	Yes	Signatures	Yes
<i>Container Security Platform</i> [4]	Aqua	No	Signatures and ML	No
<i>kAudit kArt</i> [3]	Rapid7 (Alcide)	No	Signatures and ML	No
<i>Cilium</i> [11]	Isovalent	Yes	L7 policy enforcement	Yes
<i>Calico cloud</i> [27]	Tigera	Yes	Signatures and ML	Yes
<i>Qualys Cloud Platform</i> [23]	Qualys	No	Signatures and ML	No

Table 3.1: Classification of state-of-the art tools for dynamic analysis. eBPF (extended Berkeley Packet Filter) is a technology for running sandboxed programs directly on the Linux kernel, without modifying the kernel's source code or loading module(s) from the kernel.

exploited, providing networking segmentation and access control. Some techniques only rely on a set of signatures, while hybrid techniques use both signatures and Machine Learning (ML) applied to logs, traffic, or both.

As a summary, Table 3.1 lists the main tools for performing dynamic analysis in a k8s cluster. Falco [25] detects unexpected application behavior and alerts on threats at runtime as defined by a customizable set of rules. Aqua Container Security Platform [4] monitors orchestration at runtime and includes a configurable firewall. kAudit [3] and kArt perform log analysis to spot unusual or suspicious network behavior. A firewall is embedded to allow the collection of traffic for policy compliance and anomalous behaviors detection based on ML. Cilium [11] observes network security, to enforce network policies, networking segmentation, load balancing or security policies at the kernel level. Calico cloud [27] provides intrusion and malware detection. Qualys Cloud Platform [23] provides visibility over running services through the collection of communication data. It detects anomalous container behaviors, file access, or process activity.

Such tools have a number of shortcomings that reduce their effectiveness. First, the effectiveness of signature-based tools depends on the completeness of their definition and they may not detect zero-day attacks whose signature is not already known. Also, the commercial tools using ML are not open-source and the methods they use are quite opaque.

3.2.2 General Approaches

Several approaches exist in the literature to detect malicious behaviors in microservice architectures, including signature-based approaches, ML approaches, and state-merging algorithms.

Signature-based Approaches

Dynamic analysis in the microservice architecture can be performed by monitoring systems, similar to Intrusion Detection Systems (IDS) that monitor all traffic exchanged by

microservices for signs of malicious activity or security policy violations. The signature-based analysis is performed for identifying known threats, employing a set of signatures for known threats, and their connected Indicators of Compromise (IOC). However, these tools are not specific to the microservice world, and usually, they do not consider the specifics of a k8s deployment, e.g., network policies, and pod security policies, which can be beneficial to provide a context to the analysis.

ML Approaches

On the other hand, ML-based approaches provide anomaly detection either via neural network or state machines approaches that can be effectively applied to learn base characteristics and identify risky deviations in behavior. In recent years, the utilization of neural networks has been used to detect anomalies occurring during runtime. However, it is not clear how to explain the classification decisions made by the neural network.

State-merging Algorithms

State-merging algorithms first construct a Prefix Tree Acceptor (PTA) and then iteratively go over the states of the PTA to check which states can be merged to form a smaller state machine model. Different heuristics can be used to compute the similarities between states and to decide which similar states should be merged. There exist several tools that use state-merging algorithms to learn state machine models.

CSight is a tool that uses state-merging algorithms to learn state machine model from partial log data gathered from concurrent systems. The state machine model is used to model the behavior of a distributed system [5].

Flexfringe is another tool that uses state-merging algorithms to learn state machine models from input data. It provides a list of heuristics that can be used to decide which states should be merged. The tool allows the user to use their own heuristic so that they can compare the performance of their own heuristic to the one that is already provided [28].

MINT uses state-merging algorithms to learn extended state machine models from input data. The difference between an extended state machine model and the definition that we provided in this chapter is that a transition would only occur on an extended state machine model if several conditions are satisfied instead of one single condition. MINT also uses Genetic Programming to reduce the number of transition computations between the states during the learning process [31].

Several works proposed different state-merging algorithms; Recently, Mediouni et al. provided an improvement on the RTI algorithm that was proposed in [29] and with this improvement, the algorithm can learn more accurate models [15]. Pastore et al. proposed the TkT algorithm that computes k number of transitions for each state s from the initial state machine model. The k transitions are considered a future transition for a state s and if two states have the same k transitions, these states are then merged together [22]. Vodenčarević et al. proposed an algorithm that learns a hybrid state machine model by taking different aspects of a system into account during the learning process [30].

3.2.3 Our Contribution

With respect to related works, our methodology can be assimilated to using state-merging algorithms to learn behavioral models from log data gathered from the runtime of mi-

crosservice applications. Our models can be used to gather insights, context and explanations on why a particular behavior has occurred during the runtime of a system. This can be crucial for a system administrator that needs to understand whether their system is being attacked or that there is a bug in their system. Additionally, we show that we can learn a state machine model that models the runtime behavior of a system with a microservice architecture. Furthermore, our approach uses open-source tools, which gives transparency to its inner working. The overall objective is to be able to reconstruct a baseline interaction model at runtime for the microservice applications, and then interpret deviations from the model as anomalies and potential attacks. Finally, we provide a new labelled dataset, containing both malicious and normal network data, that can be used by researchers to evaluate their methods on data that is collected from a k8s setup.

3.3 Threat Modelling in Kubernetes

Microservice architectures including k8s are composed of hundreds of containers with complex interaction patterns that can be a source of security issues. Different models have been designed to provide a taxonomy of the threats that may target such platforms.

3.3.1 Microservices Architectures

In a microservices architecture, the starting point of an attack may be achieved by exploiting a misconfiguration or a vulnerability on the orchestration platform (e.g., k8s), or compromising a vulnerability within an application (e.g., in a container). In k8s, platform-related vulnerabilities include a vulnerable cluster configuration, a kubelet's unauthenticated, an attacker joining as a fake worker node, Role-based Access Control (RBAC) issues, vulnerable network endpoints, or vulnerable service tokens. The attacker got access to the network and can then move within the cluster. Application-related vulnerabilities include misconfigured Docker, malicious Docker images, and vulnerable software applications within a pod. An application that has been compromised then leads to a foothold into the container.

With the functioning of traditional networks in mind, intuitively, such attacks are supposed to remain confined to the initially compromised pod as network-security measures are in place. Yet, exploiting the connectivity of k8s components and the software isolation of resources, the cluster may still be compromised.

3.3.2 Kubernetes Threat Matrices

These sophisticated attacks usually consist of several steps that may require the detection of different indicators (e.g., malicious network traffic, CPU overload, mounting sensitive directories) to identify specific attack patterns. As of today, the correlation of data from different sources remains complex. The MITRE ATT&CK (Adversarial Tactics, Techniques, and Common Knowledge) framework is an up-to-date database of attack techniques grouped by objectives known as *tactics*. The ATT&CK framework, originally generic, has been specialized for k8s, highlighting multi-stage cyber-attack patterns and techniques that attackers can use to infiltrate and perform damage on k8s clusters [17]. The threat matrix is made up of ten distinct families of tactics that an attacker can combine to reconstruct and exploit possible attack paths, e.g., lateral movement and privilege escalation.

Up to our knowledge, the k8s threat matrix has been very rarely used in the literature as a means to efficiently design systems considering real-world attack scenarios. Authors in [18] and [14] introduce several attack scenarios derived from the attack life-cycle introduced in the k8s threat matrix.

The Financial Services User Group (FSUG) from the Cloud Native Computing Foundation (CNCF) proposed a threat modelling in a k8s cluster with attack trees [8]. From their analysis, different end goals from an attacker have been identified:

- *Establishing foothold and persistence in the cluster.* The goal is to maintain persistence in a k8s system, by compromising the platform and ensuring an ability to persist without detection. Different levels of resilience exist, ranging from none, resilience to container restart, pod deletion, node restart, and entire cluster recycling;
- *Causing a denial of service towards the cluster.* The attacker aims to exhaust computing resources in the cluster, disrupt networking, or blackhole application traffic;
- *Running a malicious code in workload.* The attacker aims to inject malicious code into a container, deploy poisoned container images, or write new workloads into the etcd server;
- *Reading sensitive data.* The goal is to read or delete sensitive data, e.g., dumping the host memory, reading in the pod cache, or reading from the file system on the k8s host.

3.4 Methodology

The novel state anomaly detection approach presented in this work learns a state machine model from the NetFlow data collected from the k8s cluster. As it is an open challenge for using this approach on microservice architectures, we formulate the following research questions:

- *RQ1: How can state machine models be used for anomaly detection on a k8s cluster?*
- *RQ2: How effective are the state machine models for detecting anomalies on a k8s cluster?*

We first answer *RQ1* in the following subsections.

3.4.1 State-Machine Learning

For this work, we use Flexfringe to learn Probabilistic Deterministic Finite Automata (PDFA) from the NetFlow data gathered from a k8s cluster as this framework provides us with the ability to use different state-merging heuristics for the learning of PDFAs. The goal is to use Flexfringe to learn a PDFA that models the normal behavior of the k8s cluster. Any large deviation in the behavior is considered as an anomaly and an alarm is raised.

Probabilistic State Machines

State machines or automata are mathematical models that are used to model sequential behavior and they can be either deterministic or non-deterministic. In the former one,

the automaton can only switch to the next state if a symbol a_i has been read from the input; thus an automaton cannot change state if no symbol has been read from the input. Furthermore given the current state q_i of the automaton and the next input symbol a_{i+1} , there is a unique next state q_{i+j} that the automaton will switch to. On the contrary, a non-deterministic automaton can switch to another state if no symbol has been read from the input and the automaton can switch to multiple different states when the next input symbol has been read. Determinism can be important when we are modelling sequential behavior as deterministic models are easier to interpret and it is much more efficient to compute the transitions and probabilities on these models.

In this work, we use a specific variant of a state machine, namely the probabilistic state machine which is also known as a PDFA. A PDFA is a tuple $\mathcal{A} = \{\Sigma, Q, q_0, \delta, S, F\}$, where:

- Σ is a finite alphabet
- Q is a finite set of states
- $q_0 \in Q$ is a unique start state
- $\delta : Q \times \Sigma \rightarrow Q \cup \{0\}$ is the transition function
- $S : Q \times \Sigma \rightarrow [0, 1]$ is the symbol probability function
- $F : Q \rightarrow [0, 1]$ is the final probability function, such that $F(q) + \sum_{a \in \Sigma} S(q, a) = 1$ for all $q \in Q$

Given a sequence of symbols $s = a_1, \dots, a_n$, a PDFA can be used to compute the probability for the given sequence: $\mathcal{A}(s) = S(q_0, a_1) \cdot S(q_1, a_2) \dots S(q_{n-1}, a_n) F(q_n)$, where $\delta(q_i, a_{i+1}) = q_{i+1}$ for all $0 \leq i \leq n-1$. As an example, take the PDFA that is shown in Figure 3.1. The probability for the sequence $\mathcal{A}(aaab) = 0.75 \cdot 0.80 \cdot 0.80 \cdot 0.20 = 0.096$. A PDFA is called probabilistic because it assigns probabilities based on the symbols S and final F probability functions. It is called deterministic because the transition function δ (and hence its structure) is deterministic. The transition function is extended with a null symbol, representing that the transition does not exist, thus a PDFA model can be incomplete. A PDFA computes a probability distribution over Σ^* , i.e., $\sum_{s \in \Sigma^*} \mathcal{A}(s) = 1$. A PDFA can also be defined without final probability function F , in that case it computes probability functions over Σ^n , i.e., $\sum_{s \in \Sigma^n} \mathcal{A}(s) = 1$ for all $1 \leq n$.

Learning State Machine Models with State-merging Algorithms

One effective method for learning probabilistic state machines in the passive-learning approach is the utilisation of the state merging heuristic method [12]. This method essentially starts with a state machine that has a shape of a tree, also called a PTA. The algorithm then iteratively goes over the states of the PTA and does several comparisons between the states. A score is given for each comparison and this score is used to merge similar states, forming a smaller state machine. This process repeats until the algorithm cannot find similar states that can be merged. Figure 3.2 shows an example of a merge that could occur in a PTA.

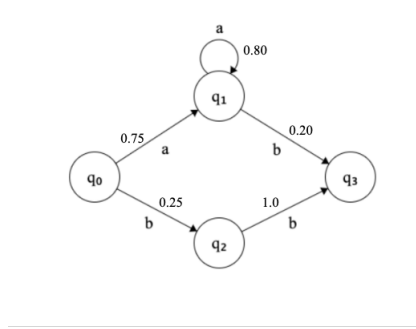


Figure 3.1: An example of a probabilistic state machine. Notice that each transition has a probability for it to occur and the sum of all probabilities of all outgoing transitions for a given state q_i is equal to one.

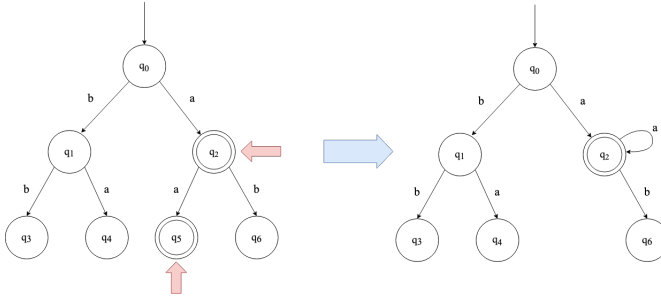


Figure 3.2: An example of merge performed a PTA.

3.4.2 Feature Selection & NetFlow data Encoding (Step 1)

For the learning of a PDFFA, we have selected three features that we used to create the traces, namely: the protocol that was used in the flow, the total number of bytes sent within the flow, and the duration of the flow. We have selected these three features as we consider these three features to best describe the communication behavior between the microservice applications and our approach does not use any service-dependent information (e.g. source/destination ports used by a service) to learn a model, making our approach more generic. Furthermore, our approach is less privacy intrusive as we try to use as few as possible features for the learning of our models.

As we are using Flexfringe as our framework for learning of PDFFA, we need to transform the raw NetFlow data into the corresponding input format of Flexfringe i.e. traces. Each flow also needs to be transformed into a single symbol. To create traces from the NetFlow data, a fixed-length sliding window can be used to create multiple traces. To transform a flow into a single symbol, it is possible to concatenate the raw values of three selected features together to form a single symbol that has the following form: *PROTOCOL_BYTES_DURATION*. However, using the raw values would lead to a very large PTA as the alphabet would be very large and this also increases the learning time. To reduce the learning time, we use three encoding methods: percentile encoding, frequency encoding and contextual frequency encoding (ENCODE). The encoding methods work as

follows:

Percentile Encoding

For the percentile encoding, we take each non-categorical feature (number of bytes sent within a flow and the duration of a flow) and divide its space into bins. The encoding for a given non-categorical feature of a given flow would be the bin number that the feature value falls into.

3

Frequency Encoding

For the frequency encoding, we use a threshold to decide whether a feature value occurs very frequently or not. If it is above the threshold, it is considered a frequently occurring value and we give the value a unique encoding label. If it is below the threshold, it would then be put together with other infrequent feature values and then the space is then further divided using percentiles. The drawback of using percentiles is that frequency information of the feature values is lost when the space is divided into bins. Infrequent values could be put together with frequent values and this causes the model to treat these values the same and it could cause the model to treat these infrequent values as frequently occurring values. This could lead the model to make a wrong prediction. Thus we have decided to use this encoding that takes frequency information into account when we are encoding the feature values.

Contextual Frequency Encoding (ENCODE)

The drawback of the frequency encoding is that it would create a very large alphabet as there could be many feature values that have a frequency that exceeds the threshold that we set. Consequently, this would create a very large PTA and thus also increases the learning time. To resolve this issue, we use a new encoding (introduced as ENCODE in Chapter 2) that computes counts of the next and previous feature values for each non-categorical feature. The counts are then used to create a matrix and the rows are then clustered using KMeans. The clustering labels are then the encoding for the given feature. The intuition of this method is that we can use the past and future information of a flow to compute the contextual information for which a feature value occurs. Similar feature values would occur in a similar context and would therefore be clustered together, and thus also have the same encoding. With this method, we use the frequency information of the feature values and we have control over the size of the alphabet.

3.4.3 Anomaly Detection using PDFA (Step 2)

Once a PDFA is learned from the NetFlow data, we can use it to compute likelihood probabilities for new traces that have been gathered during the runtime of the k8s cluster. Whenever a trace has a very low likelihood probability, it means that the behavior of the model deviates a lot from what is seen during training and an alarm would be raised. This trace will be then considered an anomaly.

3.5 Cluster Environment Setup

The state-of-the-art lacks representative datasets and relevant runtime examples, mostly due to the privacy and operational specificity of deployed systems in operation. Therefore,

we designed an open-source microservice orchestration testbed to collect real-world traffic and evaluate our methodology. Multi-step attack scenarios were then deployed, exploiting vulnerabilities of the platform or the applications. We generated a labelled dataset with mixed background traffic generated by benign users and malicious traffic collected from the generation of the attack scenarios.

3.5.1 Kubernetes Cluster Setup

We deploy a k8s testbed that acts as a playground on which experiments can be performed, which is easy to replicate on different platforms and adapt to different hardware requirements. The practical testbed is built on Vagrant and Ansible for reproducibility reasons, where the above scenarios can be replicated through containers and Virtual Machines (VMs). VMs are created and deployed created from a host machine in a private network, not accessible from the internet. VMs can, however, reach the Internet via a Network Address Translation (NAT).

The testbed is a single-cluster setup, composed of one coordinator and at least one worker node (three in our setting), with resource isolation performed by k8s namespaces. The applications aimed at the users are deployed on two “development” and “production” namespaces: they contain the same set of applications but with different levels of security, typically more restricted for the production namespace. Figure 3.3 provides an overview of the multi-cluster setup and its main components.

The *kube-system* namespace hosts core k8s services, such as the DNS and the Application Programming Interface (API) server. The *default* namespaces host the objects created without specifying the namespace. By default, resources hosted in these two namespaces are accessible from any other namespace. Four additional k8s namespaces were deployed in the cluster:

- *Storage*: providing distributed storage through Longhorn and persistent storage through the NFS server;
- *Development*: three applications – a Joomla blog (with MySQL database), an OpenSSH server, and a guestbook (with a Redis leader, Redis follower, and front end);
- *Production*: containing the same applications as the development namespace;
- *Monitoring*: Elastic stack composed of Elasticsearch, Kibana and Beats agents (Packbeat, Metricbeat, Auditbeat, and Filebeat).

3.5.2 Description of the Attack Scenarios

We propose a scenario-based view that illustrates different goals that an attacker could achieve in a k8s cluster. Each attack scenario is fragmented into several steps to follow the k8s threat matrix pattern. Figure 3.3 introduces three sample attack scenarios related to each category in the k8s threat matrix.

Scenario 1: Establishing foothold and persistence. This scenario illustrates how an attacker can establish sophisticated pod persistence by exploiting a Remote Code Execution (RCE) vulnerability in a web application and getting a reverse shell. The guestbook application, composed of the front end and the Redis leader and followers, is exploited.

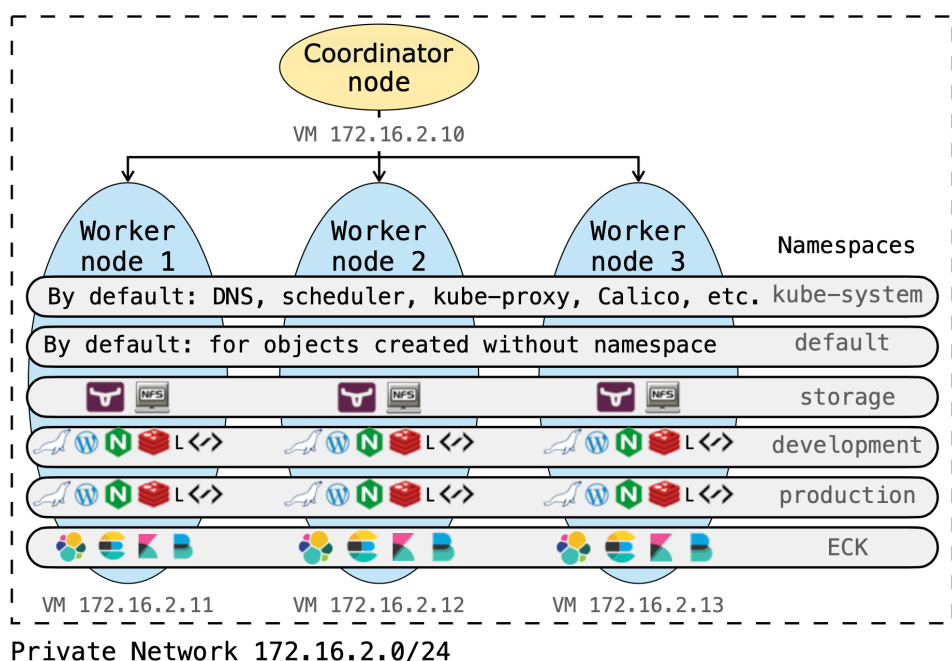


Figure 3.3: Application deployment on top of the Kubernetes testbed.

1. *Initial access*: discovering the pages of the web application and potential vulnerabilities (dirb);
2. *Execution*: exploiting an RCE to open a reverse shell on the guestbook frontend;
3. *Discovery*: fingerprinting the type of system and environment, discovering secrets mounted in the container, and getting the list of actions and permissions of the attacker based on its service account, e.g., get/list/exec pods located under the same namespace of the currently compromised pod.
4. *Lateral Movement*: the attacker uses this information to move to another pod that has root privileges.
5. *Privilege Escalation*: deploying a pod to own the node, e.g., enabling the “hostPID” parameter to remove the isolation and the “nsenter” command to give low-level access to handle the Linux kernel namespace subsystem. The attacker then breaks out of the container and gains root in the host.
6. *Persistence*: two variants corresponding to different persistent levels are deployed. First, a daemon set resilient to container restart but not to deployment deletion is created; then a Docker container is deployed outside the cluster, thus resilient to the entire cluster recycling.

Initial Access	Execution	Persistence	Privilege Escalation	Defense Evasion	Credential Access	Discovery	Lateral Movement	Collection	Impact
Using Cloud credentials	Exec into container	Backdoor container	Privileged container	Clear container logs	List K8s secrets	Access the K8s API server	Access cloud resources	Image from a private registry	Data destruction
Compromised images in registry	Bash/cmd inside container	Writable hostPath mount	Cluster-admin binding	Delete K8s events	Mount service principal	Access Kubelet API	Container service account		Resource Hijacking
Kubeconfig file	New container	Kubernetes Cronjob	hostPath mount	Pod/container name similarity	Access container service account	Network Mapping	Cluster internal networking		Denial of Service
Application vulnerability	Application exploit (RCE)	Malicious admission controller	Access cloud resources	Connect from proxy server	Application credentials in config files	Access Kubernetes Dashboard	Applications credentials in config files		
Exposed Dashboard	SSH server running inside container				Access managed identity credential	Instance Metadata API	Writable volume mounts on the host		
Exposed sensitive interfaces	Sidcar injection				Malicious admission controller		Access Kubernetes Dashboard		
							Access tiller endpoint		
							CoreDNS poisoning		
							ARP poisoning and IP spoofing		

Figure 3.3: Threat matrix for k8s including the three multi-step attack scenarios.

Scenario 2: Denial of Service (DoS) towards the cluster. This scenario illustrates a CPU DoS via the use of a vulnerable OpenSSH server. The front end of the guestbook has been built from a Docker image based on Ubuntu 16.04. This Ubuntu version includes OpenSSH v7.2, affected by *CVE-2016-6515* [20] that causes the CPU to overload as it does not limit password lengths for password authentication.

1. *Initial access*: attacker able to communicate remotely with the pod hosting the vulnerable OpenSSH server.
2. *Execution*: the remote attacker simultaneously sends 1000 requests with very long passwords to the front end pod, to cause a CPU overload.
3. *Impact*: the CPU is overloading as it receives many simultaneous requests. With the k8s Metrics Server [24], we identify that the CPU overload comes from the front end pod running the OpenSSH server. The memory of this pod goes from 10 to 150 MiBytes and from 5 to 550m CPU cores, only 5 seconds after launching the attack.

Scenario 3: Malicious code in workload. This scenario illustrates how an attacker can open a container shell through an RCE vulnerability in a Joomla application. The attacker then deploys malicious code in workload, e.g., a bitcoin miner in the cluster.

1. *Initial access*: exploiting the *CVE-2015-8562* vulnerability [19] on Joomla, which allows remote attackers to conduct PHP object injection attacks and execute arbitrary PHP code via the HTTP User-Agent header;
2. *Execution*: RCE attack deployed by unauthenticated remote attackers to gain instant access to the target server on which the vulnerable Joomla core version is installed.
3. *Discovery*: fingerprinting the type of system and environment and listing the running pods in the cluster.

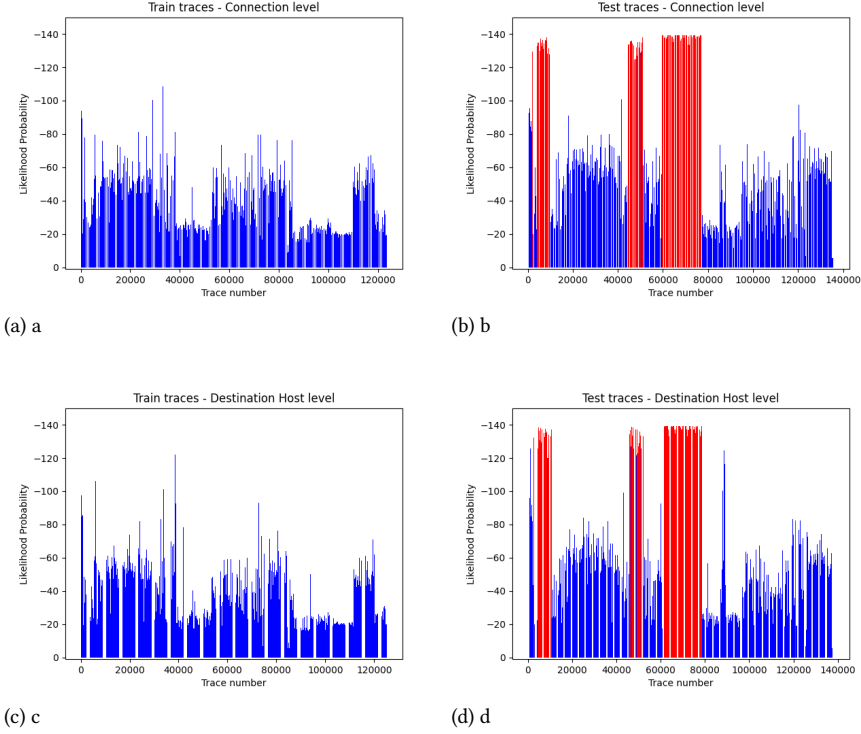


Figure 3.4: Likelihood probability plots of the traces created using the contextual frequency encoding (ENCODE). The higher the peak, the lower the likelihood probability of a given trace. Subfigures *a* and *b* present the probabilities of the train and test traces sorted by the connections, *c* and *d* present the probabilities of the train and test traces sorted by destination hosts.

4. *Lateral Movement*: accessing a privileged pod.

5. *Execution*: the attacker can deploy a bitcoin miner in the cluster once it gained access to a root pod.

3.5.3 Data Collection

To compensate for the lack of representative datasets, we generated our own labeled dataset collected on the microservice orchestration testbed. The objective is to collect both background traffic, produced by benign users with realistic behaviors, and malicious traffic simultaneously. A data collection session has been organized for two hours: during the first hour, thirty benign users were using the applications on the k8s testbed, generating background traffic, and during the second hour, the attack scenarios were deployed on the cluster while benign users were still using the applications. Then, the training set is composed of traces collected during the first hour, and the test set is composed of traces collected afterward. Traces were manually labeled depending on the IP addresses of users

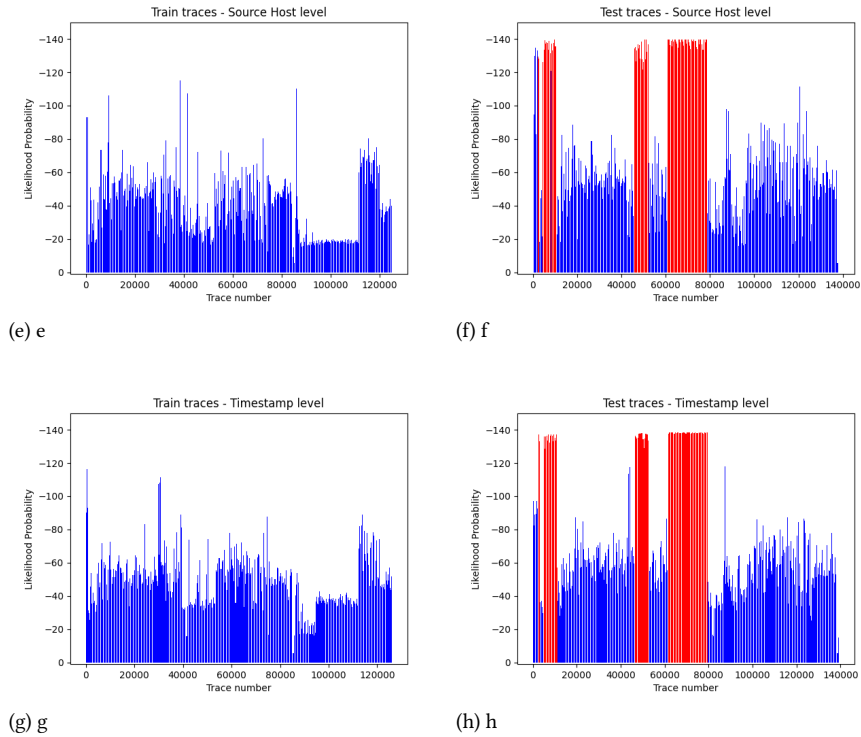


Figure 3.4: Likelihood probability plots of the traces created using the contextual frequency encoding (ENCODE). The higher the peak, the lower the likelihood probability of a given trace. Subfigures *e* and *f* present the probabilities of the train and test traces sorted by source hosts, and the remaining two subfigures (*g* and *h*) present the probabilities of the train and test traces sorted by timestamp.

and the timestamps of attacks. The dataset is publically available at [6].

3.6 Experiment Results

In this section, we present the performance results of our novel anomaly detection method that uses a state machine model. The performance results that are presented in this section form the answer to *RQ2*.

3.6.1 Performance Results

The state machine model was trained on the NetFlow data that were gathered before the attacks were launched against the k8s cluster. When creating the traces, we sorted the NetFlow data in four different manners: connection, source host, destination host, and timestamp of the flows. The intuition for sorting the NetFlow data in different manners is that some behavior might only be seen when a particular sorting level is used. We have computed the performance results of our approach using each of the different encodings that we have mentioned in Section 3.4. For comparison, we have trained an isolation forest with 100 estimators on the same training data that was used to learn our state machine model. We have used the isolation forest that is provided by the scikit-learn machine-learning package⁴.

Table 3.2 presents the performance results of our approach using the percentile encoding, frequency encoding, and contextual frequency encoding (ENCODE), and the performance results of the isolation forest that we have trained. Based on the results that are shown in the table, our approach performs much better than an isolation forest, achieving a balanced accuracy of 99.2% and a F_1 score of 0.982.

3.6.2 Likelihood Probability Plots

In our approach, we use the state machine model to compute likelihood probabilities for the network traces and use the probabilities to assess whether there is a large deviation in the behavior that was learned during training. Figure 3.4 presents the plots of the likelihood probabilities of train and test traces for each sorting level. We only present the plots that were generated using the state machine model learned with the contextual frequency encoding (ENCODE) as this encoding method gave the best results. For the sake of simplicity, we have colored all malicious traces in red and all benign traces in blue. From the probabilities plots, we can see that there is a large deviation in the network traces whenever an attack has been launched against the k8s cluster. We also see that the normal network traces produce similar probabilities as what was seen during training. From these plots, we can see why our model can detect attacks very well; visually, it is very obvious that some strange behavior is occurring in the network of the cluster.

⁴<https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.IsolationForest.html>

Table 3.2: Performance results of our approach

Method	Encoding	Sorting Level	Balanced Accuracy	F_1 Score
Our Approach	Percentile	Connection	0.888	0.714
		Source Host	0.883	0.702
		Destination Host	0.884	0.704
		Timestamp	0.885	0.715
	Frequency	Connection	0.828	0.689
		Source Host	0.817	0.675
		Destination Host	0.815	0.673
		Timestamp	0.796	0.655
	Contextual Frequency (ENCODE)	Connection	0.991	0.975
		Source Host	0.988	0.967
		Destination Host	0.989	0.972
		Timestamp	0.992	0.982
Isolation Forest	N/A	Connection	0.374	0.0006
		Source Host	0.375	0.0006
		Destination Host	0.373	0.0006
		Timestamp	0.373	0.0006

3.7 Conclusion & Future Work

In this work, we presented a novel anomaly detection method that uses a state machine model to detect anomalies within a k8s cluster. To the best of our knowledge, this work is the first that tries to apply state machine models to microservice architectures. The state machine model was learned from NetFlow data gathered from the k8s cluster when it was used normally. We have set up an experiment in which we launched three different attack scenarios on the cluster. Using our state machine approach, we managed to detect all three attack scenarios, achieving a balanced accuracy of 99.2% and a F_1 score of 0.982.

For our future research directions, we would like to use other types of log data gathered from the cluster to learn our models as currently we only use the NetFlow data. Different types of log data contain different types of information that are useful for learning a more accurate model. We can assess whether using another type of log data or a combination of different log data can help us learn a more accurate model.

Furthermore, we want to evaluate whether our approach is generalizable and applicable to other types of systems by using additional datasets. In this work, we only used data collected from a Kubernetes (k8s) cluster to run our experiments. This future research direction also allows us to compare our approach to other existing approaches.

References

- [1] D4.1. methodologies and algorithms for qualification, certification, and learning from deployed service.
- [2] D4.3. prototype for learning state machine models from moss component logs during run-time.
- [3] AlcideIO. KAudit.
- [4] aqua. Your Cloud Native Applications Secured From the Start.
- [5] Ivan Beschastnikh, Yuriy Brun, Michael D. Ernst, and A. Krishnamurthy. Inferring models of concurrent systems from logs of their behavior with csight. *Proceedings of the 36th International Conference on Software Engineering*, 2014.
- [6] Clinton Cao and Agathe Blaise. Assuremoss kubernetes run-time monitoring dataset, 2022.
- [7] Edmund M. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT Press, 1999.
- [8] CNCF. K8s attack tree - summary. (Accessed on: 30/06/2021).
- [9] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. DeepLog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 1285–1298. Association for Computing Machinery, oct 2017.
- [10] Malte Isberner, Falk Howar, and Bernhard Steffen. The open-source learnlib: A framework for active automata learning. In *Computer Aided Verification*, 07 2015.
- [11] Isovalent. Cilium - eBPF-based Networking, Observability, Security.
- [12] Kevin J. Lang, Barak A. Pearlmutter, and Rodney A. Price. Results of the abbadingo one DFA learning competition and a new evidence-driven state merging algorithm. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 1433, pages 1–12. Springer Verlag, 1998.
- [13] David Lee and Mihalis Yannakakis. Principles and methods of testing finite state machines - A survey. *Proceedings of the IEEE*, 84(8):1090–1123, 1996.
- [14] Ahmed Massoud. Threat Simulations of Cloud-Native Telecom Applications. Master’s thesis, Aalto University, 2021.
- [15] Braham Lotfi Mediouni, Ayoub Nouri, Marius Bozga, and Saddek Bensalem. Improved learning for stochastic timed models by state-merging algorithms. In Clark Barrett, Misty Davies, and Temesghen Kahsai, editors, *NASA Formal Methods*, pages 178–193, Cham, 2017. Springer International Publishing.

- [16] Weibin Meng, Ying Liu, Yichen Zhu, Shenglin Zhang, Dan Pei, Yuqing Liu, Yihao Chen, Ruizhi Zhang, Shimin Tao, Pei Sun, and Rong Zhou. Logomaly: Unsupervised detection of sequential and quantitative anomalies in unstructured logs. In *IJCAI International Joint Conference on Artificial Intelligence*, volume 2019-Augus, pages 4739–4745, 2019.
- [17] Microsoft. Threat matrix for Kubernetes, 2020.
- [18] Francesco Minna, Agathe Blaise, Filippo Rebecchi, Balakrishnan Chandrasekaran, and Fabio Massacci. Understanding the Security Implications of Kubernetes Networking. *IEEE Security & Privacy*, 19(5):46–56, 2021.
- [19] MITRE. CVE-2015-8562. (Accessed on: 14/09/2021).
- [20] MITRE. CVE-2016-6515. (Accessed on: 14/09/2021).
- [21] Sheraz Naseer, Yasir Saleem, Shehzad Khalid, Muhammad Khawar Bashir, Jihun Han, Muhammad Munwar Iqbal, and Kijun Han. Enhanced network anomaly detection based on deep neural networks. *IEEE Access*, 6:48231–48246, 2018.
- [22] Fabrizio Pastore, Daniela Micucci, and Leonardo Mariani. Timed k-tail: Automatic inference of timed automata. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 401–411, 2017.
- [23] Qualys. Qualys Cloud Platform.
- [24] Kubernetes SIGs. Kubernetes metrics server. (Accessed on: 14/09/2021).
- [25] sysdig. The Falco Project - Cloud-Native runtime security.
- [26] Tuan A. Tang, Lotfi Mhamdi, Des McLernon, Syed Ali Raza Zaidi, and Mounir Ghogho. Deep learning approach for Network Intrusion Detection in Software Defined Networking. In *Proceedings - 2016 International Conference on Wireless Networks and Mobile Communications, WINCOM 2016: Green Communications and Networking*, pages 258–263, 2016.
- [27] Tigera. Get Calico up and running in your Kubernetes cluster.
- [28] Sicco Verwer and Christian A. Hammerschmidt. flexfringe: A passive automaton learning package. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 638–642, 2017.
- [29] Sicco Verwer, Mathijs Weerdt, and Cees Witteveen. Efficiently identifying deterministic real-time automata from labeled data. *Machine Learning*, 86:295–333, 03 2012.
- [30] Asmir Vodenčarević, Hans Kleine Bürring, Oliver Niggemann, and Alexander Maier. Identifying behavior models for process plants. In *ETFA2011*, pages 1–8, 2011.
- [31] Neil Walkinshaw, Ramsay Taylor, and John Derrick. Inferring extended finite state machine models from software executions. *Empirical Software Engineering*, 21(3):811–853, 2016.

4

SEQUENT: State Frequency Estimation For Network Anomaly Detection

4

Many works have studied the efficacy of state machines for detecting anomalies within NetFlows. These works typically learn a model from unlabeled data and compute anomaly scores for arbitrary traces based on their likelihood of occurrence or how well they fit within the model. However, these methods do not dynamically adapt their scores based on the traces seen at test time. This becomes a problem when an adversary produces seemingly common traces in their attack, causing the model to miss the detection by assigning low anomaly scores. We propose SEQUENT, a new unsupervised approach that uses the state visit frequency of a state machine to adapt its scoring dynamically for anomaly detection. SEQUENT subsequently uses the scores to generate root causes for anomalies. These allow the grouping of alarms and simplify the analysis of anomalies. We evaluate SEQUENT's effectiveness in detecting network anomalies on three publicly available NetFlow datasets and compare its performance against various existing unsupervised anomaly detection methods. Our evaluation shows promising results for using the state visit frequency of a state machine to detect network anomalies.

4.1 Introduction

NetFlow encapsulates summarized network statistics of a connection between two arbitrary hosts and can be used for traffic analysis. It is a popular alternative to storing individual packets collected from a network due to its benefits: easier to collect, simpler to process, and less intrusive to the privacy of the network users as there is no possibility to inspect every packet. Despite these benefits, detecting network anomalies (e.g., network attacks) within NetFlows (or simply flows) remains a challenge, as essential information might be lost within the summarized statistics. Various works have studied the use of supervised and unsupervised machine learning (ML) models to detect network anomalies within NetFlow data [8, 13, 16, 27, 32, 36, 37].

While there has been success in applying ML models to detect anomalies in NetFlow data, a challenge faced by many security analysts is the interpretability of the predictions made by the ML model [26]. Understanding these predictions enables analysts to determine whether the model has made the correct decision and to gain confidence in the ML model deployed in the ADS.

To address this challenge, several studies have explored the use of state machines to detect network anomalies within NetFlow data, as these models are considered inherently interpretable [12, 15, 28]¹. Typically, these studies train the model using traces generated from benign flows and compute an anomaly score based on the likelihood of those traces according to the model.

Although a standard approach in ML, we argue that this has two limitations, making it not well-suited for detecting anomalies we see in practice. Firstly, many benign flows occur with a low probability, resulting in many false alarms. Secondly, malicious flows tend to resemble common flows, giving them large probabilities and thus making it hard to raise alarms. To address this, we introduce SEQUENT (short for **State Frequency Estimation for Anomaly Detection**) to overcome these limitations by using a key feature of the state machine model: its behavioral structure. Instead of computing likelihoods, we compute visit frequencies of different states within the model. Each state essentially represents a distinct type of network traffic behavior (a cluster or latent value). When we observe behavior that occurs much more frequently than expected, an alarm is raised. Intuitively, benign flows with low probability do not trigger alarms if they occur at the expected frequency. Moreover, malicious flows with high probability do raise alarms when they occur more frequently than expected. Note that this also detects infrequent malicious flows not observed at training since their expected count is assumed to be very small. The drawback is that normal flows may raise alarms when they occur more frequently. While this can happen, our experiments demonstrate that these flows have a smaller impact compared to the malicious flows.

Similar to previous studies that utilized state machines, SEQUENT learns a state machine from benign flows to model sequential network behavior. The process begins by discretizing the flow features into a symbolic representation. To achieve this, we developed a discretization method [9]² inspired by the GloVe algorithm [29]. Each selected feature is discretized independently and each flow is mapped to a symbol, created by concatenating the discretized feature values of the corresponding flow.

¹The study presented in [12] is discussed in Chapter 3.

²This is the ENCODE algorithm introduced in Chapter 2.

SEQUENT then runs each trace observed at test time through the learned model and computes the set of states visited by the trace. It uses the computed sets to update the visit frequency of each state in the model. SEQUENT compares the visit frequencies to the expected frequencies observed during training and raises an alarm whenever a behavior occurs more frequently than expected.

Another well-known challenge faced by many security analysts in their daily routine is the large volume of alerts raised by the anomaly detection system (ADS) [2, 18, 25]. The sheer volume of alerts makes it difficult for analysts to prioritize which anomalies to inspect first, turning the inspection of alerts into a labor-intensive task. SEQUENT addresses this issue by computing root causes (state identifiers) for each detected anomaly, which allows grouping and ranking of anomalies (e.g., using the frequency of a root cause). Additionally, SEQUENT establishes links between the root causes and their corresponding flows. The aim is to provide analysts with concrete instances of anomalous behavior, enabling a quick and in-depth analysis of anomalies.

We empirically demonstrate SEQUENT's effectiveness in the task of detecting various types of network anomalies using three publicly available NetFlow datasets [11, 14, 21]³. Our experiment results demonstrate that SEQUENT is highly effective in the detection of network anomalies, outperforming existing state-of-the-art unsupervised methods. The main contributions of this paper are summarized as:

- We present SEQUENT, a novel approach that employs state visit frequencies of a state machine for anomaly detection.
- SEQUENT offers the capability to group and rank anomalies, empowering analysts to filter and prioritize which anomalies they should investigate.
- We compare SEQUENT against existing state-of-the-art unsupervised methods on three different datasets, demonstrating its effectiveness and generalizability.

4.2 Preliminaries

4.2.1 State Machine

A state machine, formally known as a finite state machine or finite state automaton, is a mathematical model used for modeling the sequential behavior of a system. It is formally defined as a 5-tuple $(\Sigma, Q, q_0, \delta, F)$ [30], where:

- Σ is a finite alphabet.
- Q is a finite set of states.
- $q_0 \in Q$ is a unique start state.
- $\delta : Q \times \Sigma \rightarrow Q$ is the transition function that determines which transition should be taken based on the current state and the (new) symbol read from the input.
- $F \subseteq Q$ is the set of final states.

³ [11] is the study discussed in Chapter 3

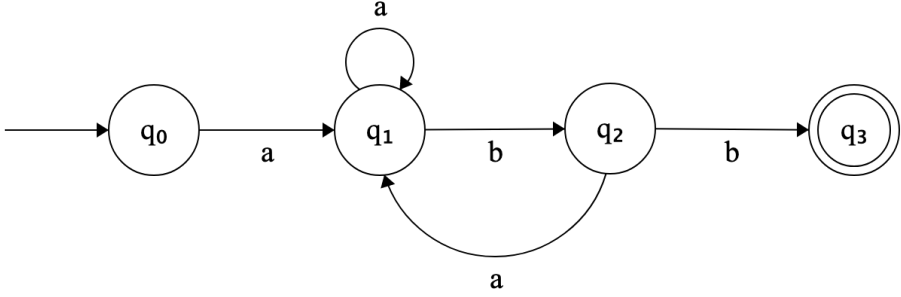


Figure 4.1: An illustrative example of a state machine. At the end of the input trace, *aabab*, we would end up in state q_2 .

4

Given an input trace t (a sequence of events, where each event $e \in \Sigma$) and a system S , one can use the state machine to determine in which state S would end up, thereby understanding the behavior the system would exhibit. In the context of this work, each event in a trace represents a single flow. An illustrative example of a state machine is shown in Figure 4.1. The state machine in this example consists of only one final state, q_3 .

4.2.2 State Machines Considered in SEQUENT

State machines come in two flavors, namely deterministic or non-deterministic state machines [30]. SEQUENT solely considers deterministic state machines as they are easier to interpret; simpler to predict the next event, and verifying the same sequence of events would execute the same sequential behavior is more straightforward [19]. With a non-deterministic state machine, multiple transitions can be triggered simultaneously, even when no event has occurred. These non-deterministic transitions can inaccurately increase the visit frequencies of states within a state machine, causing states (or behaviors) to be falsely flagged as anomalous. Since SEQUENT aims to address the challenge of large volume alerts by providing root causes for anomalies, non-deterministic transitions can result in providing false root causes to analysts.

Furthermore, the use of state machine is intended provide analysts with a way to inspect the sequential behaviors exhibited by an arbitrary system. However, when multiple transitions are triggered simultaneously within a non-deterministic state machine, it becomes difficult for an analyst to determine which path should be followed in the model given an arbitrary trace. For these reasons, SEQUENT focuses only on learning deterministic state machines.

We can employ two approaches for learning a state machine from trace data: active and passive learning. We opt for the latter approach as it only requires us to collect traces from the target system S instead of continuously interacting with S to learn a state machine [3]. State-merging heuristic algorithm is a highly effective method for learning state machines in the passive-learning approach [17]. It begins with a tree-like state machine, known as the Prefix Tree Acceptor (PTA) (see Figure 4.2). It then iteratively compares pairs of states to identify states that can be merged. Merges are done based on the scores computed during the comparisons; for example, states are merged when their merge score

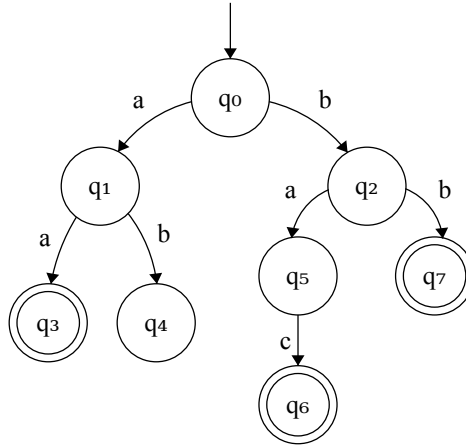


Figure 4.2: An example of a PTA.

has exceeded the minimum score threshold. The algorithm ends when it cannot find any new merges. The final output is an approximation of the smallest state machine consistent with the input data.

4.3 Assumptions of the Adversary

Our work is grounded in a set of assumptions we formulated concerning the objectives and capabilities of an adversary. We consider an adversary attacking an arbitrary target network TN , driven by various objectives such as infiltrating TN to find machines containing sensitive data, disrupting normal operations by launching denial of service attacks against specific services, orchestrating a botnet, or infecting machines with malware. In our main experiments, the training and test data from the datasets simulate an adversary attacking TN without taking our defense into account. In our robustness experiments, we assume a more adept adversary who is aware of an ADS within TN but unaware of its specifics. Their goal is to remain undetected until they have accomplished their objectives. To accomplish this, the adversary can monitor TN to collect normal network behaviors and then attempt to bypass the ADS by generating behaviors resembling those collected during normal operations within TN . The adversary can only listen to the data that was sent but cannot delete or alter any of the sent data.

4.4 Related Works

Traditional Methods. Several works have studied the efficacy of unsupervised learning for network anomaly detection. The primary focus has been on utilizing traditional unsupervised ML algorithms (e.g., Isolation Forest). Zoppi et al. conducted multiple studies evaluating the effectiveness of traditional unsupervised methods for anomaly detection on NetFlow data [36, 37]. Campazas-Vega et al. also studied the effectiveness of Isolation Forests in detecting anomalies within NetFlow data. In contrast to SEQUENT, these stud-

ies do not dynamically adapt the anomaly scores based on the events/traces seen at test time for anomaly detection.

State Machines. Numerous studies have investigated the efficacy of state machines for anomaly detection. Pellegrino et al. introduced BASTA, a system that learns probabilistic timed state machines from NetFlows and uses them as behavioral fingerprints for machines infected with botnet malware [28]. Alerts are raised when traces match behaviors captured in the fingerprints. Unlike SEQUENT, BASTA uses malicious data for learning rather than solely relying on benign data. Additionally, BASTA learns a state machine for each malware. This approach requires analysts to maintain an updated list of malware samples to learn fingerprints. Any missing malware samples may result in the system failing to detect new or unknown malware. SEQUENT learns a single state machine that models the behavior exhibited in a network and is not specific to any particular malware or network anomalies.

Lin et al., Cao et al., and Matoušek et al. learned probabilistic state machines to detect anomalies in different systems[12, 20, 22]⁴. These works raise alerts when a new trace deviates significantly from what was observed during training. The degree of deviation is quantified based on the likelihood of a trace. In contrast, SEQUENT quantifies the abnormality based on state visit frequency for a corresponding trace and compares this to what was observed during training.

Grov et al. presented a different approach for learning state machines to detect anomalies in NetFlows [15]. The approach learns state machines using solely traces of network protocols extracted from benign data. The degree of abnormality is based on the normalized edit distance of an arbitrary trace under scrutiny; the number of changes needed such that each trace event can be replayed in the state machine. In contrast to SEQUENT, this work does not utilize the frequency of the states to adapt the scores dynamically at test time. Furthermore, it solely considers network protocols for the learning, whereas SEQUENT considers other features.

Neural Networks. Deep Neural Networks (DNN) gained increasing popularity among researchers in recent years due to their ability to identify complex hidden patterns that traditional models may struggle to detect. Mirsky et al. introduced Kitsune, a state-of-the-art solution that utilizes an ensemble of Autoencoders (AE) to detect network anomalies in an unsupervised manner. It extracts various statistics from benign data and uses them for training. Kitsune then attempts to recreate instances it has seen during training by computing similar statistics for arbitrary new data. Alarms are raised for instances with large recreation errors[23]. Neural networks are notoriously known to be difficult to interpret, as it is not always clear what happens between the input and output layers. Nguyen et al. sought to address this issue with their framework, GEE. It uses a Variational Autoencoder (VAE) to detect anomalies and generates explanations by providing SHAP-like plots of the gradients computed for NetFlow features[27]. However, it is not clear how a security analyst should interpret these gradients. In contrast, SEQUENT learns an inherently interpretable model and generates root causes, which can be used to facilitate the understanding of (malicious) sequential behaviors. Root causes are linked back to concrete instances producing the anomalous behaviors.

⁴ [12] is the study discussed in Chapter 3.

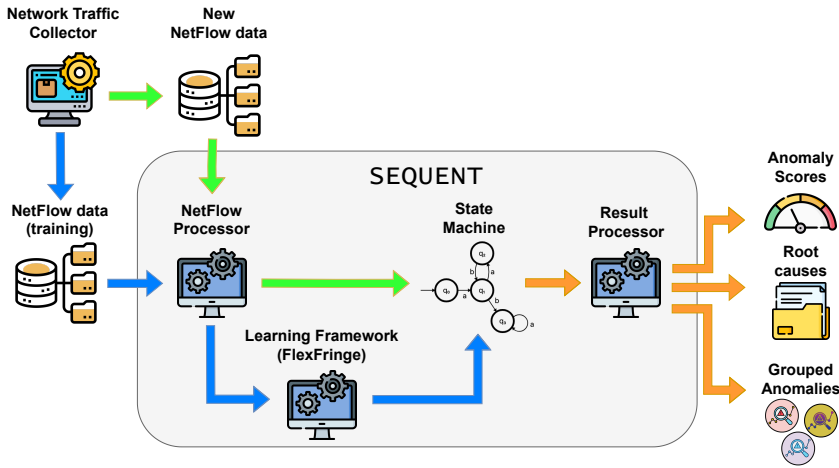


Figure 4.3: SEQUENT's anomaly detection pipeline.

4.5 SEQUENT

4.5.1 Intuition Behind SEQUENT

SEQUENT employs a novel approach for computing the anomaly score of an arbitrary trace under scrutiny. Unlike the measures of abnormality presented in prior works [12, 15, 20, 22, 28], SEQUENT leverages the frequencies of the state visits in a state machine to compute an adaptive anomaly score. The underlying principle of SEQUENT is that behaviors occurring significantly more often than those observed during training are indications of anomalies. Traces under scrutiny are replayed on a state machine to determine how well they fit within the learned model. Each trace visits a set of states, and the frequency of visits to each state is compared to the expected frequency observed during training. When the frequency of visits to a state is significantly higher than its expected frequency, it signals that unusual behavior is occurring in this state. The more significant the increase in visits for this state, the more likely it is to show anomalous behavior. SEQUENT employs this rationale to dynamically adapt its scores for arbitrary traces seen at test time. SEQUENT's intuition will also work for (anomalous) behaviors never seen before during training, as these will visit unusual paths during the replay.

4.5.2 SEQUENT's Anomaly Detection Pipeline

Figure 4.3 depicts SEQUENT's anomaly detection pipeline. It provides a high-level overview illustrating the flow of learning a state machine from training data (represented by the blue arrows), the flow for running predictions on new data (represented by green arrows), and the flow for generating the results (represented by the orange arrows). Before NetFlows are used for training or prediction, they are first transformed into traces by the NetFlow Processor component, as depicted in the figure. Each flow is described in the following paragraphs.

Processing NetFlows. NetFlow features need to be discretized before learning a state machine, given that each flow contains both continuous and discrete feature data. Fur-

thermore, each flow needs to be transformed into an event (discrete symbol) to generate traces (sequence of events). Although the discretization process is not strictly necessary for the learning of a state machine, using the raw NetFlow feature data (particularly continuous data) will lead to the combinatorial explosion of states. Discretization enables us to reduce the number of possible combinations in the feature data, thereby also reducing the number of states used in the state machine. To discretize NetFlow features, we develop an encoding algorithm inspired by the GloVe algorithm [29]. Our encoding algorithm computes co-occurrences between values for a given input flow feature f and stores them in a matrix. Each row denotes a unique value v of f and how often it occurs with another value v' . Feature values are discretized by first clustering the rows using KMeans and then using cluster assignments as their discretization label. We make our encoding algorithm available on Github [9].

4

To transform a flow into an event symbol, we concatenate the discretized feature values of the selected features with an underscore symbol to form a single event symbol. For example, if three features are used to represent an event, we get the following event symbol: *DFV1_DFV2_DFV3*, where *DFV* is the abbreviation for discretized feature value. In the case of one feature (e.g., bytes), we directly use its discretized values. To convert events into traces, we apply a sliding window of length N to generate traces. These are then subsequently used to learn a state machine. Traces that do not have a length of N are excluded from learning as they do not provide sufficient information.

Learning of a State Machine. SEQUENT utilizes FlexFringe [33] to learn a state machine from NetFlow traces. This framework offers, out of the box, various heuristics for learning a state machine. Additionally, it enables its users to learn other types of models such as Markov Chains. FlexFringe also automates the creation of traces from the CSV file provided as input data. The user has to specify which features to use to create events, the length of the sliding window, and how the data should be sorted (e.g., by connections) to create the traces.

Since Flexfringe's trace creation feature of FlexFringe matches the trace creation method mentioned in the previous paragraph, we leverage this feature of FlexFringe to generate traces from the provided input NetFlows. We opt for FlexFringe in the design of SEQUENT based on these aforementioned features.

Once the selected features of the training data are discretized, we forward the CSV containing discretized feature values of the training to FlexFringe to create traces and learn a state machine. We specify to FlexFringe to use the discretized features values for the creation of the event symbols.

When learning is completed, FlexFringe outputs a state machine along with various statistics, such as how often each state of the state machine was visited during training. We store the model and the statistics for the computation of the rolling anomaly score for traces observed at test time.

Computing Rolling Anomaly Scores. At test time, the new NetFlow data is processed in the similar manner as with the training data and is forwarded to FlexFringe. FlexFringe simulates each trace generated from the test data and computes the sequence of states visited by each trace. For each computed state sequence, we compute its rolling anomaly scores with the help of the statistics extracted during training of the model. Algorithm 1 depicts the pseudocode for computing rolling anomaly scores for an arbitrary

trace t .

The first step in computing the rolling anomaly scores is to extract the state visit frequencies of each state and the sum of all visit frequencies (see lines 4 to 8). The sum of all visit frequencies (stored in “*TotalVisits*” from line 3) is used for normalization, as each state may have been visited a different number of times during training, causing some states to be more frequent than others. Note that lines 2 to 8 do not have to be repeated for each arbitrary trace we observe at test time. However, we have included it in Algorithm 1 for the sake of completeness. In our implementation of SEQUENT, we compute this once and store the result in memory.

The next step in computing rolling anomaly scores is to simulate the (test) trace t on the state machine and extract the path it traverses in the model (line 10). For each state q' in the extracted state sequence (i.e., the path), we compute its anomaly score (lines 11 to 18). Note that since we are working with visit frequencies, large counts can have a considerable impact on the score. To address this, we use the logarithm of the frequencies to compute the score for q' (as shown on lines 12 to 17). The addition of one within the logarithm is used to ensure that we do not encounter the undefined behavior when of calculating the logarithm of zero. The base of the anomaly score for q' starts with its visit frequency observed so far at test time (line 12). If the q' has not been observed yet, a value of zero will be returned. The observed frequency (at test time) is then compared to the corresponding frequency extracted from the training statistics (line 13) and the score is normalized (lines 14 and 15). Finally, the visit frequency of state q' is updated (line 16), and the score is added to the results list (line 17). The anomaly score of trace t is the sum of all individual scores in the result list. We keep the individual scores, as they are used to compute the root cause of an anomaly (explained in the next paragraph).

Computing Root Causes. An advantage of using a state machine to model sequential behavior is the ability to follow and analyze the path that an arbitrary trace t takes within the model. We leverage this capability to compute a root-cause symbol for t . We do so by finding the state with the largest rolling anomaly score visited by t and using the state identifier as the root-cause symbol. It denotes the state that causes the largest anomalous activity in t . As an example, let us use the model shown in Figure 4.1 as the model that was learned from benign flows. Let $t=[a, a, b, a, b, a, a]$ be the trace under scrutiny, $seq=[1, 1, 2, 1, 2, 1, 1]$ be the sequence of states visited by t and $a=[0.19, 0.25, 0.18, 0.30, 0.20, 0.35, 0.40]$ be the rolling anomaly scores computed for seq . State 1 has the highest anomaly score (0.40), making “1” the root-cause symbol for t .

Grouping Anomalies. A key benefit of SEQUENT is the ability to create groups of anomalies exhibiting similar network behavior. SEQUENT leverages the root-cause symbols to group anomalies. The rationale is that traces sharing the same root-cause symbol denote behaviors exhibiting a notable level of anomalous activity in the same state. As traces signify the paths taken within the state machine, and the anomaly score of a state increases based on its visit frequency, traces sharing the same root-cause symbol will frequently visit this particular state. These traces will partly consist of the same events.

Linking Root Cause Back to Data. Another benefit of SEQUENT is its ability to link root causes to the corresponding flow, producing a notable level of anomalous activity. SEQUENT achieves this by utilizing the insights produced by FlexFringe after making predictions on a set of arbitrary traces. For each prediction of a trace t , FlexFringe reports

Algorithm 1: SEQUENT's Rolling Anomaly Score**Input:** State Machine S , Trace t , Training Statistics TS **Output:** Rolling anomaly scores of t

```

1  $AnomalyScores \leftarrow []$ 
2  $StateFreq_{train} \leftarrow \emptyset$ 
3  $TotalVisits \leftarrow 0$ 
4 for each state  $q$  in  $S$  do
5    $Freq_q \leftarrow extractVisitFreq(TS, q)$ 
6    $StateFreq_{train}[q] \leftarrow freq_q$ 
7    $TotalVisit \leftarrow TotalVisit + Freq_q$ 
8 end
9  $StateFreq_{test} \leftarrow \emptyset$ 
10  $StateSequence \leftarrow computeStateSequence(S, t)$ 
11 for each state  $q'$  in  $StateSequence$  do
12    $StateScore \leftarrow \log(StateFreq_{test}[q'] + 1)$ 
13    $StateScore \leftarrow StateScore - \log(StateFreq_{train}[q'] + 1)$ 
14    $StateScore \leftarrow StateScore + \log(TotalVisit + 1)$ 
15    $StateScore \leftarrow StateScore + \log(|StateFreq_{test}| + 1)$ 
16    $StateSeq_{test}[q'] \leftarrow StateSeq_{test}[q'] + 1$ 
17    $AnomalyScores.add(StateScore)$ 
18 end
19 return  $AnomalyScores$ 

```

details on which lines of the CSV file were used for the creation of t . Specifically, the start and end points are reported. SEQUENT utilizes this information to pinpoint the exact flow within the CSV file, producing a notable level of anomalous activity in t and links this to the root-cause symbol. This enables us to collect concrete instances of NetFlows for anomalies that are grouped under the same root-cause symbol.

4.6 Experiment Setup

4.6.1 Dataset Selection

To showcase SEQUENT's effectiveness in identifying network anomalies in an unsupervised setting, we have selected three publicly accessible datasets. These datasets comprise malicious NetFlows originating from a wide range of attacks. This allows us to assess SEQUENT's ability to identify various types of attacks. Furthermore, the varied nature of these datasets enables us to evaluate SEQUENT's generalizability across different network architectures.

AssureMOSS Dataset. The first dataset consists of NetFlows collected from a Kubernetes cluster [11] and used in the study conducted by Cao et al. [12]⁵. It contains benign flows generated by real-life users and malicious flows generated by three attack scenarios. The three scenarios are constructed using a specialized threat matrix of the MITRE ATT&CK framework, designed explicitly for Kubernetes [12]. Each attack is fragmented into various steps that an adversary has to execute (e.g., initial access, execution, discovery, etc.). The three attack scenarios cover attacks such as (1) exploiting a vulnerability to discover secrets and using them to do privilege escalation, (2) executing a denial of service (DoS) attack, and (3) exploiting a vulnerability that allows the deployment of malware. Cao et al. created the dataset to investigate the effectiveness of state machines for the aforementioned network attacks. For this reason, we selected this dataset as one of the baselines for our work.

CTU-13 Dataset. The second dataset is a well-known NetFlow dataset consisting of a large collection of realistic botnet network traffic mixed with normal and background traffic. Specifically, there are thirteen captures of various botnet samples. Each comprises various actions and protocols exhibited by the deployed malware [14]. The captures cover a wide range of attacks, e.g., distributed DoS, port scans, Click-Fraud, and sending spam. The benign and background flows were collected from normal machines deployed on a university network, and malicious flows were collected from virtual machines that were running the different botnets. Each virtual machine was bridged onto the university network to collect flows for the final dataset. This dataset has been used in multiple prior studies that investigated the effectiveness of state machines in detecting botnet traffic (e.g., Pellegrino et al. [28] and Grov et al. [15]) and serves as another suitable baseline for our work.

UGR-16 Dataset. The third and final dataset consists of 100 GB of NetFlows collected from a Spanish ISP. The benign flow data was collected from real-life clients of the ISP, and the malicious flow data was collected from five machines that were used to launch the attacks. The attacks were not launched against any clients but rather five machines that were set up as victim machines [21]. The dataset mostly covers various types of DoS and

⁵This study is discussed in Chapter 3

scanning attacks. Furthermore, it also covers attacks in which spam emails are sent and botnet traffic. The botnet traffic was collected from the CTU-13 dataset and was modified before they were inserted into this dataset. The modifications consist of matching the IP addresses and timestamps of the flows to the ones used in their setup. We selected this dataset as one of our baseline datasets because it contains a wide range of network behaviors produced by various users.

4.6.2 Creating Train/Test Splits

We outline the procedure for creating the train/test split for each dataset. Before the split is created, NetFlows are first sorted by timestamp and then by connections. We believe connection data has less noise than data sorted by timestamps, and this would help with learning more accurate models.

The AssureMOSS dataset was collected over a two-hour period, during which no attack scenarios were initiated in the first hour. In the study conducted by Cao et al., the first hour was used as the training data because it contains only benign flows, while the second hour was used as the test data. Because SEQUENT also learns a state machine from benign flows for anomaly detection, we adopt the same train/test split as used by Cao et al[12]. This results in a training set containing 120,000 flows and a test set containing 275,000 flows.

As for the CTU-13 dataset, we follow the split presented by its authors [14]: scenarios 3,4,5,7,10,11,12 and 13 for training and scenarios 1,2,6,8,9 for test. The training scenarios contain benign and malicious flows, so we extracted only benign flows for training. Additionally, using all data from each scenario will lead to an overly large training and test set. To address this, we sampled a subset of flows from each scenario, resulting in 640,000 flows for training and 522,000 flows for test.

For the UGR-16 dataset, we follow the split presented by its authors [21]: data from March to June are used as training data, while data from July and August are used as test data. As this dataset contains, by far, the largest amount of NetFlow data, we use the same sampling technique as with CTU-13 to create our training and test set. This yields a total of 600,000 flows as training data and 664,000 flows as test data.

4.6.3 Methods for Comparisons

We collected methods previously evaluated on the selected datasets to assess how SEQUENT compares to existing unsupervised and state-of-the-art methods. Additionally, we included various methods that serve as baselines.

From Literature. The AssureMOSS dataset was recently published by Cao et al. and remains relatively unexplored in the context of evaluating other unsupervised anomaly detection methods. Cao et al. conducted a preliminary evaluation comparing their approach against Isolation Forest and Local Outlier Factor [12].

Regarding the CTU-13 dataset, Zoppi et al. evaluated various unsupervised methods for identifying network anomalies [36, 37]. Specifically, the authors evaluated the efficacy of Isolation Forest and Fast Angle-Based Outlier Detection (FastABOD). Additionally, Grov et al. evaluated the efficacy of Markov Chains in detecting botnets [15]. Pellegrino et al. also conducted a study in which they evaluated the efficacy of (probabilistic) state machines and Bigram models (which are essentially Markov Chains) in detecting botnets [28].

For the UGR-16 dataset, Zoppi et al. evaluated the use of two different methods to detect network attacks [36, 37]. However, due to their inability to handle the size of our train/test splits, we dropped them for comparison. Nguyen et al. utilized this dataset to evaluate the efficacy of GEE. Similar to Kitsune [23], they use a state-of-the-art deep learning technique to detect anomalies. However, GEE explicitly trains a single VAE instead of an ensemble of AE.

We select the following from the compiled list of methods to compare against SEQUENT: Isolation Forest, FastABOD, Markov Chain, (probabilistic) state machines, and GEE. As for the implementation of the selected methods, we use Isolation Forest from the scikit-learn library [7], FastABOD from the PyOD library [35], Flexfringe to learn a Markov Chain and (probabilistic) state machine, and the implementation of GEE written by Wong Mun Hou [34]. For each method, except for the Markov Chain, we use the same parameters reported by their authors to train a model on the corresponding dataset. In cases where no parameter information is provided, we make our best effort to train a model by following the reported training steps. We chose to learn a Markov Chain using FlexFringe as we employ a distinct set of features, and learning a Markov Chain follows a standard procedure.

Additional Methods. We included additional methods beyond those previously mentioned to compare against SEQUENT. One of these methods is the straightforward Boxplot Method presented by Arp et al. [5], which employs the frequency of data to detect anomalies. Furthermore, it enables us to determine whether a simple method is already sufficient to detect anomalies in our constructed test sets or if a more complex approach is required. For our experiment, we wrote our implementation of the Boxplot Method following the description provided by the authors. Zoppi et al. also used Histogram-Based Outlier Detection (HBOD) to detect network anomalies. Although not applied to any of the selected datasets, it utilizes the frequency of bins to detect anomalies. This serves as a suitable alternative baseline in our comparison. We use PyOD's implementation of HBOD. Finally, we included Kitsune as it is a state-of-the-art unsupervised ADS. We use the implementation provided by the authors [24]. As it was designed to run at the packet level, we have to make minimal changes for it to work at the flow level. We recognize that this might negatively impact its performance, but we believe that it is beneficial to have this as a baseline for assessing SEQUENT's usefulness.

4.6.4 Feature Selection & FlexFringe Setup

As selected datasets do not report the same flow features, we are required to select a subset of features shared among the selected dataset. The features shared between the selected datasets are timestamp, duration, protocol, number of bytes sent, number of packets sent, source and destination IP addresses, and source and destination ports. From this subset of features, we select the duration, protocol, number of bytes sent, and the number of packets sent to generate traces for SEQUENT. Port numbers are not considered within SEQUENT as they are spurious features within the CTU-13 dataset [26]. We discretize all the selected features except for protocol, as it is already a categorical feature.

We configure FlexFringe to sort the flows based on the source and destination IP pairs as we want to learn a state machine from connection data. Furthermore, we configure FlexFringe to use a sliding window of length ten to generate traces for the training and

test sets. By default, FlexFringe stops replaying a trace if it reaches a state and no outgoing transition exists for the next event in the trace. We, therefore, configure FlexFringe to allow the replay to reset back to the root state and continue with the remaining events. This enables us to compute the visit frequencies of the states for the complete trace. Our FlexFringe configuration settings are available on our repository [10].

4.7 Evaluation Results

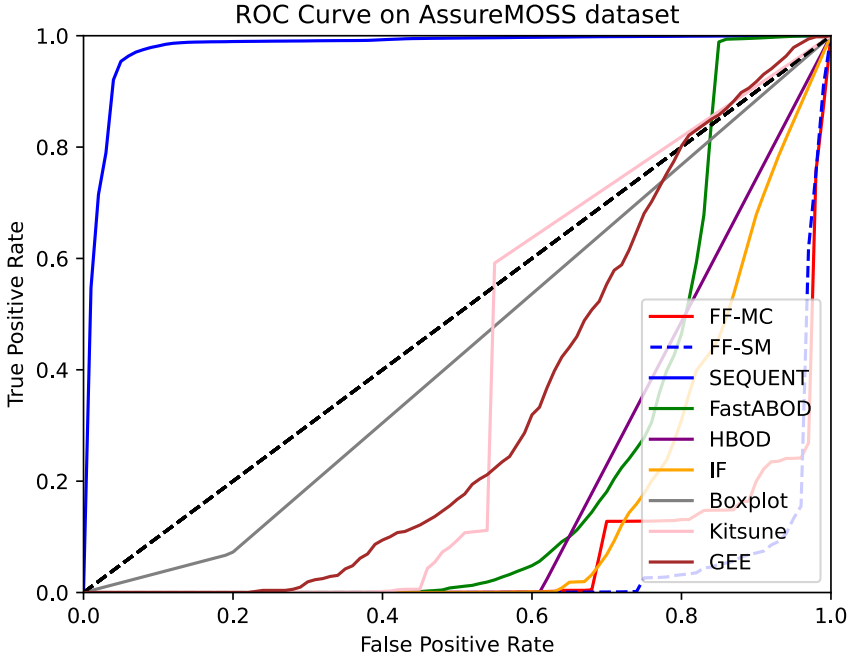


Figure 4.4: Average ROC curves computed for the AssureMOSS dataset.

4.7.1 Effectiveness of SEQUENT

We report SEQUENT's efficacy in detecting various network anomalies on the selected datasets. We use the Area Under Curve (AUC) score as the comparison metric between the various types of models. We deem this a suitable metric for comparison as each method generates anomaly scores for different traces, and we are uncertain about the optimal classification threshold that would yield the most favorable outcomes. Besides the AUC scores, we also report the Receiver Operator Characteristics (ROC) curves computed for each model. The AUC scores and ROC curves for each dataset are reported in Table 4.1, Figure 4.4, Figure 4.5, and Figure 4.6, respectively. These are average performance results computed from 10 independent runs. All models were trained on the same features con-

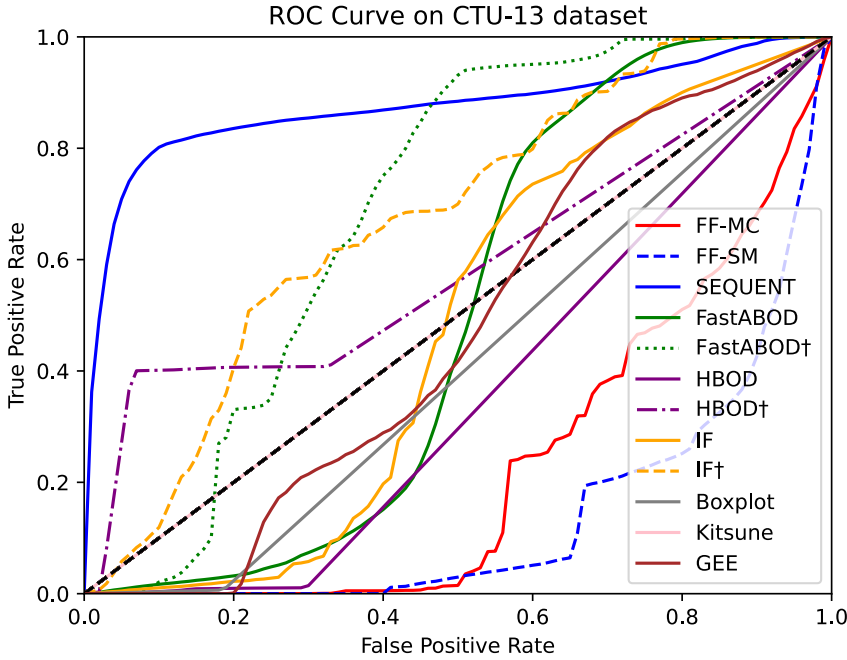


Figure 4.5: Average ROC curves computed for the CTU-13 dataset.

sidered by SEQUENT. We also include models trained on the original feature set used in the original study. These are marked with the $\dagger$ -symbol in the table. Furthermore, *FF-MC* denotes the Markov Chain learned using the FlexFringe framework and *FF-SM* the (probabilistic) state machine. The reported results demonstrate that SEQUENT consistently outperforms all other models in detecting network anomalies on each respective dataset. Moreover, the results also suggest that SEQUENT is generalizable over different network architectures.

4.7.2 Runtime Analysis of SEQUENT

To evaluate SEQUENT's runtime costs, we measured its training times and compared them to those of the other ML models. Additionally, we computed SEQUENT's average runtime for computing rolling anomaly scores at test time. Table 4.2 presents the average training times of all ML models across all datasets. Table 4.3 presents the average time SEQUENT takes to compute a rolling anomaly score for a given state at test time. The runtime averages were obtained from the ten independent runs of our experiments.

From the training times presented in Table 4.2, we observe that SEQUENT takes approximately half a minute to learn a model from benign flows. While it is not the fastest of all models, it is considerably faster than models such as GEE and FastABOD. In our experiments, we used a GPU to train GEE, whereas SEQUENT does not require a GPU

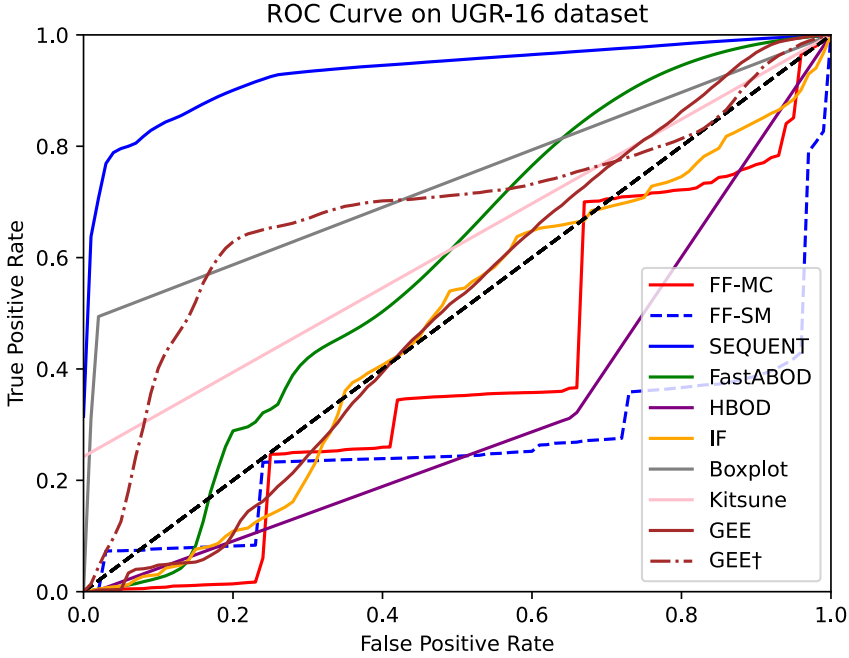


Figure 4.6: Average ROC curves computed for the UGR-16 dataset.

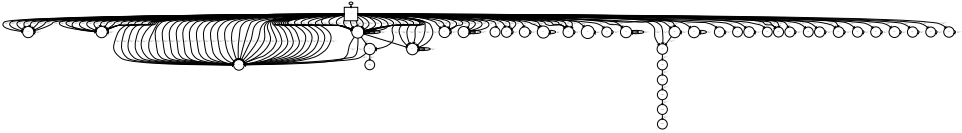


Figure 4.7: Simplified state machine model learned from CTU-13 dataset.

to learn a state machine. Although many other models have considerably lower training times, they are not as effective as SEQUENT in detecting network anomalies. This reflects a trade-off between training time and a model's detection performance. We believe that spending a bit more time to train SEQUENT can be justified, as it achieves a considerably higher detection performance than the other models. We further support our belief based on SEQUENT's runtimes in computing rolling anomalies scores across the datasets (see Table 4.3). SEQUENT computes anomaly scores in microseconds on average, making it a practical solution for network anomaly detection.

4.7.3 Example Analysis Using Root Causes

One of SEQUENT's advantages is the ability to group anomalies using the generated root-cause symbols. This enables an analyst to reduce the time spent analyzing false positives

Model	AssureMOSS	CTU-13	UGR-16
Isolation Forest	0.140	0.473	0.426
Isolation Forest [†]	-	0.672	-
FastABOD	0.229	0.492	0.576
FastABOD [†]	-	0.688	-
HBOD	0.195	0.357	0.326
HBOD [†]	-	0.589	-
Boxplot Method	0.436	0.410	0.738
FF-MC	0.067	0.232	0.392
GEE	0.335	0.473	0.505
GEE [†]	-	-	0.683
Kitsune	0.371	0.5	0.621
FF-SM	0.041	0.142	0.257
SEQUENT	0.978	0.874	0.933

Table 4.1: Average AUC of each model across all datasets.

Model	AssureMOSS	CTU-13	UGR-16
Isolation Forest	0.18 ± 0.01	1.34 ± 0.11	1.00 ± 0.04
Isolation Forest [†]	-	1.26 ± 0.13	-
FastABOD	16.13 ± 0.65	88.30 ± 1.26	82.34 ± 2.83
FastABOD [†]	-	101.29 ± 6.92	-
HBOD	0.02 ± 0.0	0.25 ± 0.50	0.10 ± 0.0
HBOD [†]	-	0.27 ± 0.54	-
Boxplot Method	7.24 ± 0.25	67.14 ± 0.41	100.50 ± 0.53
FF-MC	1.07 ± 0.03	6.51 ± 0.13	5.22 ± 0.09
GEE	99.00 ± 0.10	127.44 ± 0.57	169.69 ± 0.57
GEE [†]	-	-	169.55 ± 0.35
Kitsune	1.24 ± 0.08	6.38 ± 0.26	6.06 ± 0.22
FF-SM	4.02 ± 0.12	17.41 ± 1.18	22.88 ± 1.53
SEQUENT	5.11 ± 0.30	19.23 ± 0.22	27.21 ± 2.55

Table 4.2: Average training time (seconds) computed across all datasets.

Dataset	Avg. Runtime for Score Computation
AssureMOSS	2.17 ± 3.96
CTU-13	2.20 ± 5.17
UGR-16	2.36 ± 3.62

Table 4.3: Average runtime (microseconds) for computing rolling anomaly score for a state.

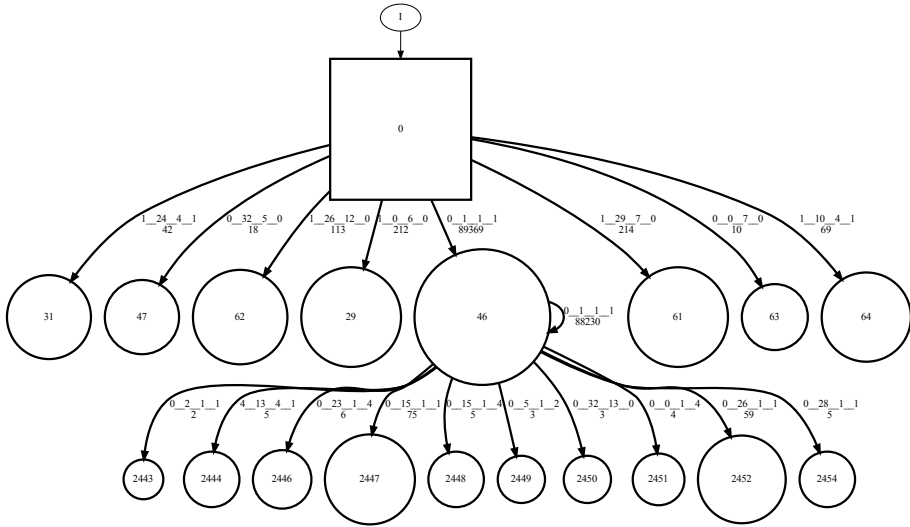


Figure 4.8: Zoomed-in state machine of Figure 4.7, showing the top three anomalous states (47, 62 and 31) presented in Table 4.4.

by filtering out symbols deemed to be false positives. We provide an illustrative example to demonstrate how these symbols can aid with the analysis of anomalies. Let's say that an analyst has set an arbitrary threshold on the (rolling) anomaly scores to select “interesting” alerts for further analysis. The analyst then groups the selected traces based on their root-cause symbols and computes the size of each group. Table 4.4 presents the top 3 largest groups of anomalies computed from the CTU-13 dataset. Additionally, Figure 4.7 illustrates the simplified state machine learned by SEQUENT. The analyst decides to first analyze group “62” by investigating the corresponding (starting) flows collected and linked by SEQUENT (see Table 4.5). The most anomalous flows are shown at the top. SEQUENT deems these flows to be anomalous due to the significant increase in network behavior that coincidentally correlates with the use of port 25, which is used for Simple Mail Transfer Protocol (SMTP). The analyst deems these flows to be suspicious due to the destination port used in the communication and marks all traces with the root-cause symbol “62” as malicious. Examining the actual labels of these flows, they correspond to the malicious traffic produced by the Neris botnet. This malware specifically sends spam mail [14], which explains the use of the SMTP port (25).

Following the same approach, the analyst now investigates group “47” using the linked flows depicted in Table 4.6. SEQUENT does not know the actual nature behind these flows, but due to the significant increase in this type of network behavior, it deems these flows to be anomalous. Upon inspection, the analyst observes that these are correlated with the use of port 53, which is typically used for domain name resolution (DNS). This significant increase could be related to a DNS flood attack, or it could simply be that this type of network behavior was rarely seen during training. The analyst deems these flows

Root-Cause Symbol	Size
62	18646
47	3182
31	2556
1501	2448
72	2244

Table 4.4: Anomaly group sizes computed using root-cause symbols generated for the CTU-13 dataset

src_ip	dst_ip	sport	dport	num_bytes	num_packets
147.32.84.165	66.94.237.64	3271	25	186	3
147.32.84.165	98.139.175.225	3539	25	186	3
147.32.84.165	205.188.146.194	2474	25	186	3
147.32.84.165	76.96.30.116	3268	25	186	3
147.32.84.165	216.157.130.15	1490	25	186	3
147.32.84.165	98.137.54.237	3630	25	186	3
147.32.84.165	64.12.138.161	1789	25	186	3
147.32.84.165	209.85.143.27	3328	25	186	3
147.32.84.165	205.188.190.2	3857	25	186	3
147.32.84.165	98.139.175.225	3508	25	186	3

Table 4.5: Simplified flows linked to the top 10 anomalous traces for root-cause symbol "62".

unrelated to an attack and marks all traces grouped under the symbol "47" as false positives. By investigating only a subset of the linked flows, the analyst can filter out 3182 false positive alerts and reduce the time spent analyzing every alert. Furthermore, the analyst can visualize the part of the model causing the most anomalous activity (see Figure 4.8) to understand the sequential behavior exhibited by the most anomalous groups of traces.

4.7.4 SEQUENT's Robustness

An essential characteristic of an effective ADS is its ability to withstand attacks launched by adversaries seeking to evade detection. With various works showing that ML models are susceptible to adversarial attacks [4, 6, 31], it is imperative to evaluate SEQUENT's efficacy against (potential) adversarial attacks. To so do, we constructed four distinct types of adversarial attacks following the adversary's capabilities presented in Section 4.3. We strictly consider attacks that an adversary could launch at test time, and we do not consider adversarial attacks such as poisoning of training data. An adversary is only allowed to adapt the features considered by SEQUENT for anomaly detection, and they are allowed to collect benign flows from the targeted network before the start of an attack. The constructed adversarial attacks are defined as follows:

- **Padding Attack:** The adversary is allowed to mimic normal behavior by adapting each feature value to its nearest value within the collected data.
- **Random Replacement Attack:** The adversary mimics normal behavior by replacing their flows with random flows selected from the collected data.

src_ip	dst_ip	sport	dport	num_bytes	num_packets
147.32.84.138	147.32.80.9	45170	53	214	2
147.32.84.138	147.32.80.9	57876	53	214	2
147.32.84.138	147.32.80.9	50385	53	214	2
147.32.84.138	147.32.80.9	45753	53	214	2
147.32.84.138	147.32.80.9	34683	53	214	2
147.32.84.138	147.32.80.9	42564	53	214	2
147.32.84.138	147.32.80.9	56081	53	214	2
147.32.84.138	147.32.80.9	58016	53	214	2
147.32.84.138	147.32.80.9	53136	53	214	2
147.32.84.138	147.32.80.9	40272	53	214	2

Table 4.6: Simplified flows linked to the top 10 anomalous traces for root-cause symbol "47".

4

- **Window Replacement Attack:** The adversary mimics normal behavior by replacing each sliding window (10 flows) of their flows with a randomly selected window of benign flows. Each window is selected at most once.
- **Frequency Replacement Attack:** The adversary mimics normal behavior by finding frequently occurring flows (at least 100 times) from the collected data and replaces their flows with randomly selected frequently occurring flows. The intuition behind this attack is that an adversary would be able to blend in well with the normal flows if they mimic the frequently occurring communication behavior.

Tables 4.7 presents SEQUENT's robustness against the aforementioned adversarial attacks on each selected dataset. The results are the average AUC scores across 10 independent runs. Based on the reported results, we see that SEQUENT achieved the best performance against two types of adversarial attacks (*Padding Attack* and *Random Replacement Attack*) compared to all other models. This can be explained by the nature of these two attacks; when an adversary tries to mimic the normal network behavior by padding or doing replacements on their flows, the traces will resemble the traces observed at training. However, this also changes the sequential behavior exhibited in the traces, which also results in changes in the state visit frequencies during the replay. SEQUENT detects this unusual increase in visit frequencies in the states and will raise an alarm. Existing methods trained on the raw feature value will not detect this deviation as these values are similar to what was observed during training. Probabilistic models such as Markov Chains and probabilistic state machines also suffer from this issue as the adversarial traces are similar to what was observed during training. The models will assign higher probabilities to these traces, thus resulting in poor detection performance.

Another interesting observation is that SEQUENT's performance deteriorates significantly on the AssureMOSS and UGR-16 datasets but not on the CTU-13 dataset when an adversary launches a *Window Replacement Attack*. This decline in performance is expected since all malicious windows are randomly replaced by benign windows. Unlike the other three attacks, flows or feature values are not randomly replaced; instead, the flows and their sequential behaviors resemble the ones seen during training. As each window is allowed to be selected at most once, the sequential behavior exhibited in the (copied) win-

dow occurs at a slightly higher frequency, though not significantly more than observed during training. The complexity behind this attack makes it challenging for SEQUENT to detect the adversary. However, SEQUENT exhibits robustness against this specific attack on the CTU-13 dataset. We speculate that this result is influenced by the type of attack captured in the dataset.

One discussion point that we would like to initiate concerning three of the constructed adversarial attacks (in which replacement is involved) is whether an adversary could still accomplish their objectives by producing exact copies of the normal traffic data to avoid detection by an ADS. We argue that by producing literal copies and even going to the extent of recreating the sequential behaviors of benign flows, the adversary is essentially replaying collected network traffic data. No system, not even the analyst doing manual inspection of each flow or packet, would be able to explain how the data produced by an adversary differs from the normal traffic data as it is a literal copy.

Model	Padding	Random	Window	Frequency
AssureMOSS				
Isolation Forest	0.162	0.185	0.431	0.155
FastABOD	0.222	0.220	0.177	0.031
HBOD	0.195	0.226	0.404	0.195
GEE	0.576	0.441	0.427	0.916
Kitsune	0.859	0.862	0.573	0.835
FF-MC	0.014	0.062	0.0319	0.985
FF-SM	0.007	0.038	0.278	0.985
Boxplot	0.435	0.465	0.502	0.441
SEQUENT	0.991	0.903	0.155	0.771
CTU-13				
Isolation Forest	0.464	0.484	0.515	0.478
FastABOD	0.492	0.475	0.436	0.290
HBOD	0.357	0.369	0.392	0.497
GEE	0.470	0.372	0.522	0.490
Kitsune	0.5	0.5	0.5	0.5
FF-MC	0.231	0.2	0.283	0.298
FF-SM	0.142	0.091	0.323	0.441
Boxplot	0.5	0.5	0.5	0.5
SEQUENT	0.970	0.966	0.861	0.974
UGR-16				
Isolation Forest	0.431	0.429	0.480	0.470
FastABOD	0.588	0.566	0.468	0.336
HBOD	0.327	0.333	0.364	0.373
GEE	0.115	0.122	0.113	0.052
Kitsune	0.621	0.621	0.621	0.621
FF-MC	0.323	0.308	0.491	0.640
FF-SM	0.174	0.166	0.518	0.811
Boxplot	0.740	0.758	0.426	0.742
SEQUENT	0.983	0.919	0.407	0.927

Table 4.7: The performance (average AUC) of each model against the four adversarial attacks.

4.8 Conclusion & Future Work

We introduced SEQUENT, a framework that employs state machines to detect various types of network anomalies. SEQUENT leverages the structure of a state machine to collect state visit frequencies and subsequently uses them to continually adapt its scoring for arbitrary traces seen at test time. We observed that SEQUENT achieves a better overall performance in detecting various types of network anomalies, even outperforming existing state-of-the-art unsupervised anomaly detection methods. Furthermore, we demonstrated how SEQUENT computes root causes for the detected anomalies and uses them to group and rank anomalies. Moreover, SEQUENT links root causes back to the concrete NetFlow instances. The root causes, and the linked instances enable an analyst to quickly filter out a large number of false alerts, reducing the time spent on analyzing them. Finally, we evaluated SEQUENT's robustness against four types of adversarial attacks. Reported results show that SEQUENT achieved the best performance in two of the adversarial attacks.

One potential future research direction is to investigate how the intuition underlying SEQUENT's scoring method can be applied to other existing models, such as isolation forests or neural networks. It is not trivial how the internal structure of these models can be employed to continuously adapt its anomaly scores for new data seen at test time.

Another research direction is to apply SEQUENT to software log data. We believe that SEQUENT might also be effective in detecting anomalies in software logs, and it would be interesting to compare SEQUENT against existing anomaly detection methods developed for software log data.

4.9 Acknowledgments

This work is funded under the Assurance and certification in secure Multi-party Open Software and Services (AssureMOSS) Project, (<https://assuremoss.eu/en/>), with the support of the European Commission and H2020 Program, under Grant Agreement No. 952647.

References

- [1] D4.4. runtime access-based control –report and prototype.
- [2] Bushra A. Alahmadi, Louise Axon, and Ivan Martinovic. 99% false positives: A qualitative study of SOC analysts’ perspectives on security alarms. In *31st USENIX Security Symposium*, pages 2783–2800, Boston, MA, August 2022. USENIX Association.
- [3] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987.
- [4] Giovanni Apruzzese, Michele Colajanni, Luca Ferretti, and Mirco Marchetti. Addressing adversarial attacks against security systems based on machine learning. In *2019 11th International Conference on Cyber Conflict (CyCon)*, volume 900, pages 1–18, 2019.
- [5] Daniel Arp, Erwin Quiring, Feargus Pendlebury, Alexander Warnecke, Fabio Pierazzi, Christian Wressnegger, Lorenzo Cavallaro, and Konrad Rieck. Dos and don’ts of machine learning in computer security. In *31st USENIX Security Symposium*, pages 3971–3988, 2022.
- [6] Battista Biggio, Iginio Corona, Davide Maiorca, Blaine Nelson, Nedim Šrndić, Pavel Laskov, Giorgio Giacinto, and Fabio Roli. Evasion attacks against machine learning at test time. In Hendrik Blockeel, Kristian Kersting, Siegfried Nijssen, and Filip Železný, editors, *Machine Learning and Knowledge Discovery in Databases*, pages 387–402, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.
- [7] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. `sklearn.ensemble.IsolationForest` —scikit-learn 1.1.2 documentation. <https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.IsolationForest.html>, June 2024.
- [8] Adrián Campazas-Vega, Ignacio Samuel Crespo-Martínez, Ángel Manuel Guerrero-Higueras, Claudia Álvarez-Aparicio, Vicente Matellán, and Camino Fernández-Llamas. Malicious traffic detection on sampled network flow data with novelty-detection-based models. *Scientific Reports*, 13(1):15446, 2023.
- [9] Clinton Cao. ENCODE: Encoding NetFlows for Network Anomaly Detection. <https://github.com/tudelft-cda-lab/ENCODE>, 2022.
- [10] Clinton Cao. SEQUENT: State Frequency Estimation for Anomaly Detection. <https://github.com/tudelft-cda-lab/SEQUENT>, 2022.
- [11] Clinton Cao and Agathe Blaise. AssureMOSS Kubernetes Run-time Monitoring Dataset. https://data.4tu.nl/articles/_/20463687/1, 2022.

- [12] Clinton Cao, Agathe Blaise, Sicco Verwer, and Filippo Rebecchi. Learning state machines to monitor and detect anomalies on a kubernetes cluster. In *Proceedings of the 17th International Conference on Availability, Reliability and Security, ARES '22*, New York, NY, USA, 2022. Association for Computing Machinery.
- [13] Igor Fosić, Drago Žagar, Krešimir Grgić, and Višnja Križanović. Anomaly detection in netflow network traffic using supervised machine learning algorithms. *Journal of Industrial Information Integration*, 33:100466, 2023.
- [14] S. Garcia, M. Grill, J. Stiborek, and A. Zunino. An empirical comparison of botnet detection methods. *Computers & Security*, sep 2014.
- [15] Gudmund Grov, Marc Sabate, Wei Chen, and David Aspinall. Towards intelligible robust anomaly detection by learning interpretable behavioural models. *NISK J*, 32:1–16, 2019.
- [16] Waseem Hassan, Seyed Ebrahim Hosseini, and Shahbaz Pervez. Real-time anomaly detection in network traffic using graph neural networks and random forest. In Yevgeni Koucheryavy and Ahmed Aziz, editors, *Internet of Things, Smart Spaces, and Next Generation Networks and Systems*, pages 194–207, Cham, 2024. Springer Nature Switzerland.
- [17] Kevin J. Lang, Barak A. Pearlmutter, and Rodney A. Price. Results of the abbadingo one dfa learning competition and a new evidence-driven state merging algorithm. In Vasant Honavar and Giora Slutzki, editors, *Grammatical Inference*, pages 1–12, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.
- [18] Maxime Lanvin, Pierre-François Gimenez, Yufei Han, Frédéric Majorczyk, Ludovic Mé, and Eric Totel. Towards understanding alerts raised by unsupervised network intrusion detection systems. In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses, RAID '23*, page 135–150, New York, NY, USA, 2023. Association for Computing Machinery.
- [19] Edward A. Lee. Determinism. *ACM Trans. Embed. Comput. Syst.*, 20(5), may 2021.
- [20] Qin Lin, Sridha Adepu, Sicco Verwer, and Aditya Mathur. Tabor: A graphical model-based approach for anomaly detection in industrial control systems. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security, ASIACCS '18*, page 525–536, New York, NY, USA, 2018. Association for Computing Machinery.
- [21] Gabriel Maciá-Fernández, José Camacho, Roberto Magán-Carrión, Pedro García-Teodoro, and Roberto Therón. UGR '16: A new dataset for the evaluation of cyclostationarity-based network IDSs. *Computers and Security*, 73:411–424, mar 2018.
- [22] Petr Matoušek, Vojtěch Havlena, and Lukáš Holík. Efficient modelling of ics communication for anomaly detection using probabilistic automata. In *2021 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, pages 81–89. IEEE, 2021.

- [23] Yisroel Mirsky, Tomer Doitshman, Yuval Elovici, and Asaf Shabtai. Kitsune: An ensemble of autoencoders for online network intrusion detection. In *Proceedings 2018 Network and Distributed System Security Symposium*. Internet Society, 2018.
- [24] Yisroel Mirsky, Tomer Doitshman, Yuval Elovici, and Asaf Shabtai. Kitsune-py: A network intrusion detection system based on incremental statistics (afterimage) and an ensemble of autoencoders (kitNET). <https://github.com/ymirsky/Kitsune-py>, May 2024.
- [25] Azqa Nadeem, Sicco Verwer, Stephen Moskal, and Shanchieh Jay Yang. Alert-driven attack graph generation using s-pdfa. *IEEE Transactions on Dependable and Secure Computing*, 19(2):731–746, 2022.
- [26] Azqa Nadeem, Daniël Vos, Clinton Cao, Luca Pajola, Simon Dieck, Robert Baumgartner, and Sicco Verwer. Sok: Explainable machine learning for computer security applications. In *2023 IEEE 8th European Symposium on Security and Privacy*, pages 221–240. IEEE, 2023.
- [27] Quoc Phong Nguyen, Kar Wai Lim, Dinil Mon Divakaran, Kian Hsiang Low, and Mun Choon Chan. GEE: A Gradient-based Explainable Variational Autoencoder for Network Anomaly Detection. In *2019 IEEE Conference on Communications and Network Security (CNS)*, pages 91–99. Institute of Electrical and Electronics Engineers Inc., jun 2019.
- [28] Gaetano Pellegrino, Qin Lin, Christian Hammerschmidt, and Sicco Verwer. Learning behavioral fingerprints from Netflows using Timed Automata. In *Proceedings of the IM 2017 - 2017 IFIP/IEEE International Symposium on Integrated Network and Service Management*, pages 308–316. Institute of Electrical and Electronics Engineers Inc., jul 2017.
- [29] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543. ACL, 2014.
- [30] Michael Sipser. *Introduction to the Theory of Computation*. Course Technology, Boston, MA, third edition, 2013.
- [31] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks, 2014.
- [32] Andrea Venturi, Giovanni Apruzzese, Mauro Andreolini, Michele Colajanni, and Mirco Marchetti. Drelab - deep reinforcement learning adversarial botnet: A benchmark dataset for adversarial attacks against botnet intrusion detection systems. *Data in Brief*, 34:106631, 2021.
- [33] Sicco Verwer and Christian A. Hammerschmidt. Flexfringe: A passive automaton learning package. In *Proceedings - 2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017*, pages 638–642. IEEE, 2017.

- [34] Mun Hou Wong. GEE. <https://github.com/munhouiani/GEE>, May 2024.
- [35] Yue Zhao. PyOD. <https://pyod.readthedocs.io/en/latest/>, June 2024.
- [36] Tommaso Zoppi, Andrea Ceccarelli, Tommaso Capecchi, and Andrea Bondavalli. Un-supervised anomaly detectors to detect intrusions in the current threat landscape. *ACM/IMS Trans. Data Sci.*, 2(2), apr 2021.
- [37] Tommaso Zoppi, Mohamad Gharib, Muhammad Atif, and Andrea Bondavalli. Meta-learning to improve unsupervised intrusion detection in cyber-physical systems. *ACM Trans. Cyber-Phys. Syst.*, 5(4), sep 2021.

Part II

Validating Deployment Against Implementation

5

Conformance Analysis Tool For Microservice Applications

5

The microservice architecture allows developers to divide the core functionality of their software system into multiple smaller services. However, this architectural style also makes it harder for them to debug and assess whether the system's deployment conforms to its implementation. We present CATMA, an automated tool that detects non-conformances between the system's deployment and implementation. It automatically visualizes and generates potential interpretations for the detected discrepancies. Our evaluation of CATMA shows promising results in terms of performance and providing useful insights. CATMA is available at <https://cyber-analytics.nl/catma.github.io/>, and a demonstration video is available at <https://youtu.be/WKP1hG-TDKc>.

The content of this chapter is based on the publication “Clinton Cao, Simon Schneider, Nicolas E. Diaz Ferreyra, Sicco Verwer, Annibale Panichella, and Riccardo Scandariato. 2024. CATMA: Conformance Analysis Tool For Microservice Applications. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion '24)*. Association for Computing Machinery, New York, NY, USA, 59–63. <https://doi.org/10.1145/3639478.3640022>”. This work was part of a public report from the AssureMOSS project, which is available at [1].

5.1 Introduction

Software systems following the microservice architectural paradigm have their core functionality split into multiple smaller components. These microservices (or just *services*) of a microservice application (MSA) communicate via lightweight communication protocols such as REST APIs or message brokers. The services of an MSA can be developed, maintained, and deployed independently, paving the way for an increasing trend in the adoption of this architectural style. Despite these benefits, MSAs pose a challenge in gaining a comprehensive overview due to their inherently decoupled and distributed nature. Consequently, debugging faults is a time-consuming process because the localization of the root cause is challenging. According to studies, developers usually take several days to debug and find the cause of a fault [10, 21]. Many approaches for the automatic extraction of architectural representations of MSA have been proposed [2, 6, 11, 16], thus addressing the challenge of gaining an overview of the applications' architecture. Some approaches combine static and dynamic analysis to build the architectural models. Also, multiple fault localization techniques for MSAs have been proposed [8, 22], which use dynamic analysis to identify faults and pinpoint the root cause in code. However, to the best of our knowledge, no work compares the results from static and dynamic analysis rather than merging them. Moreover, none of the existing fault localization approaches offer explainability in the form of possible interpretations for the faults.

In this paper, we present CATMA, a novel tool designed to analyze and compare statically and dynamically obtained architectural models. CATMA autonomously identifies potential non-conformances between these models, generating easily accessible visualizations for users and providing concise interpretations. These interpretations reduce the number of lines in source code that users need to scrutinize when investigating a non-conformance. We tested CATMA on four open-source MSAs and conducted a preliminary usability study with two participants. The results indicate that the tool effectively supports developers during the localization and debugging of non-conformances, demonstrating its usefulness and potential in the debugging landscape for microservices.

5.2 Running Example

The software engineering team of ZYX Inc. is working on their new web application for selling tech products. They embrace the microservice architectural style as this allows them to split up into smaller groups and work independently on the core functionalities of their application. Each member follows the best practices of software engineering; using static analysis to detect faults and testing each functionality before its deployment. After finishing the development, they deploy the application to test it out. To their surprise, they notice that the monitoring service does not receive any metrics data. They are unsure of the cause of this discrepancy since a static analysis tool correctly detects the line of code that implements the transmission of metrics data and does not raise any warnings. They spend several days analyzing different log files, but have no luck in finding the underlying cause. They scratch their heads and start wondering whether there is a tool that provides:

- the detection of discrepancies between the implementation and deployment of MSAs,
- a high-level overview of such discrepancies, and

- descriptions of the potential root causes.

5.3 CATMA

Workflow. Figure 5.1 depicts CATMA’s workflow. First, the *Model Processor* ① reads the input models (static and dynamic) to extract architectural components. The obtained data is passed on to the *Non-conformance Detector* ②, which checks whether there are any non-conformances (discrepancies) between static and dynamic models. If a non-conformance is detected, it is forwarded to both the *Interpretation Generator* ③ and the *Non-conformance Visualizer* ④. The latter ④ collects all detected non-conformances and generates a visualization of the system’s architecture that shows the non-conformances. The former ③ generates a set of possible interpretations for each detected non-conformance, which describe potential causes. These interpretations are forwarded to the *Interpretation Visualizer* ⑤, which generates HTML pages that visualize the interpretations. CATMA is designed to be modular. Each component can be replaced or expanded to fit the user’s needs. The tool is invoked via the command line (see Listing 5.1).

```
$ ~/Doc/Git/CATMA python3 CATMA.py \
--static_model_path data/ewolff_microservice/ewolff_microservice_static_model.json \
--dynamic_models_path data/ewolff_microservice/dynamic_models/ \
--output_path ./output/
Reading configuration file...
Processing static model...
Processing dynamic model...
Detecting non-conformances: 100% |====|13/13 [00:00<00:00, 83245.73it/s]
Detecting non-conformances: 100% |====|13/13 [00:00<00:00, 152733.76it/s]
Detected 2 static non-conformances and 1 dynamic non-conformance
between implementation and deployment of the system!
Generating non-conformance interpretations...
Generating non-conformance visualizations...
Generating interpretation visualizations...
```

Example 5.1: Command-line invocation of CATMA.

Detecting Non-conformances. As static models, CATMA accepts dataflow diagrams (DFDs) like the ones introduced by Schneider and Scandariato [14]. These DFDs are automatically extracted from source code and configuration files by searching for relevant keywords and using them as evidence to build relationships between services. As dynamic models, state machines inferred from HTTP events logs are expected. They are created

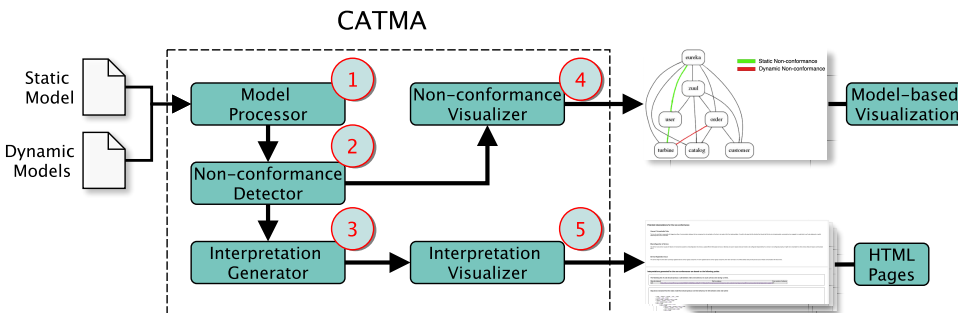


Figure 5.1: CATMA’s workflow. Input models are processed ① and non-conformances between them detected ② and visualized ④. Each non-conformance is visualized ⑤ and possible interpretations for it are generated ③.

using a similar model-inference approach as presented by Cao et al. [4]. The approach first extracts logs from a Kubernetes cluster using Packetbeat. It then utilizes Flexfringe [17] to generate behavioral traces and learns a state machine from these traces. The *Model Processor* extracts services and connections between them from both input models. They represent the application's architecture and are used to detect non-conformances. In the DFD, nodes and edges depict the services and information flows between them, respectively. We can, therefore, directly extract the nodes and edges. In a state machine, services and their corresponding relations are represented differently; each transition in a state machine indicates which services in the system have communicated with each other. Thus, nodes and edges are extracted from the transitions of the state machines. The *Model Processor* creates a set of nodes and edges for both input models, where edges are represented as “*service X* $\rightarrow$ *service Y*” and denote the communication relationship between the two services.

Non-conformances are detected by identifying differences between the sets of nodes and edges. The *Non-conformance Detector* iterates through the sets and checks for each item whether it exists in both corresponding sets. We define *static non-conformances* as nodes or edges missing from the static model (compared to the dynamic model) and *dynamic non-conformances* as those missing from the dynamic model. Each item is tagged according to this comparison, i.e., indicating whether it is present in both, only the static, or only the dynamic model. The tagged sets of nodes and edges are passed to components ③ and ④.

The *Non-conformance Visualizer* is responsible for creating a graphical representation of any detected non-conformances. It generates a PlantUML file (see plantuml.com) which presents the nodes and edges as a graph and where a coloring scheme highlights any found non-conformances. Model items observed in both models are colored black, items only observed in the static model (dynamic non-conformances) are colored blue, and items only observed in the dynamic model (static non-conformances) are colored orange. In addition, dynamic and static non-conformances are visually distinguished by means of dotted and dashed lines, respectively.

Interpreting Non-conformances. CATMA generates a set of possible interpretations for each detected non-conformance. These interpretations are visualized in an HTML page by the *Interpretation Visualizer*. The HTML page helps users analyze the potential causes of non-conformances. CATMA presents a specific set of interpretations for both types of non-conformance. The generated HTML pages contain (1) the type, definition, and involved services of the non-conformance, (2) the set of possible interpretations, and (3) additional details that support the understanding of the non-conformance. In the following, (2) and (3) are described further.

Providing Interpretations of Non-Conformances. A set of high-level textual interpretations is provided, which describes possible underlying causes of the detected non-conformances. The interpretations are meant to serve as possible starting points to debug found non-conformances. Currently, the generation is based solely on the type of non-conformance, i.e., whether it is static or dynamic. We formulated a text describing possible interpretations for both types of non-conformances, and the corresponding one is presented to the user. As the basis for these interpretations, we collected known causes of non-conformances from the literature (e.g., [9, 19]). These causes range from standard

Potential Interpretations For the Non-Conformance

Implicit Call via Third-Party Services

Communication flow has been detected between the two services but the responsible line of code is not detected in the source code. It could be that the call is implicitly triggered by a third-party service that is used in the implementation (e.g. via an annotation that is used by a framework or that code is injected during run-time).

Unintentional Endpoint Exposure

The communication flow detected between the two services could be caused by an endpoint that is exposed unintentionally by the developer. It could be the case that code has been refactored and the endpoint was not removed during refactoring, or that the endpoint was used for testing purposes and was not removed afterwards.

Code Located Outside of Default Source Location

The line of code that is responsible for triggering the flow of communication between the two services is not located in the default source-code folder of the project. It could be the case that the line of code was unintentionally introduced in a different folder (e.g. resources folder).

Figure 5.2: Example set of textual interpretations.

programming errors made in software development to common causes for issues encountered by developers of MSAs. As an example, misconfiguration of services is a common cause of dynamic non-conformances in MSAs. When services are not properly configured, they become undiscoverable by other services, leading to missing expected runtime behaviors. CATMA uses this information as a basis for the generation of one interpretation for a dynamic non-conformance. For the collection, we disregarded non-conformances rooted in hardware-related issues, e.g., due to non-deterministic behavior because of multi-threading or similar effects. Figure 5.2 presents the set of textual interpretations that are provided for a static non-conformance.

Our future work will predominantly focus on this part of the tool, specifically on implementing a more intelligent generation of applicable interpretations. In this regard, we will analyze indicators for each cause of non-conformances. These indicators will then be used to decide whether a cause is plausible or not for a given non-conformance. This will lead to the generation of a tailored set of possible interpretations for each found non-conformance. The already carried-out analysis of the related literature provides the basis for this future work.

Additional details. The generated HTML page also presents additional details that could aid the user with the understanding of the detected non-conformance. In the case of static non-conformances, a state machine is visualized that depicts the unexpected sequential communication behavior detected between the involved services. The most frequently occurring calls between the involved services are presented in a human-readable format right after the state machine model. This insight can be used to understand why such calls were made between the involved services. Figures 5.3 and 5.4 show an example of a state machine and the most frequent calls, respectively. In the case of a dynamic non-conformance, we instead leverage the traceability information contained in the static model to point to the code that shows the expected behavior. Specifically, the page presents (1) the line of code responsible for triggering the missing runtime event (i.e., the line of code that should have been executed), (2) the sequence of events that should

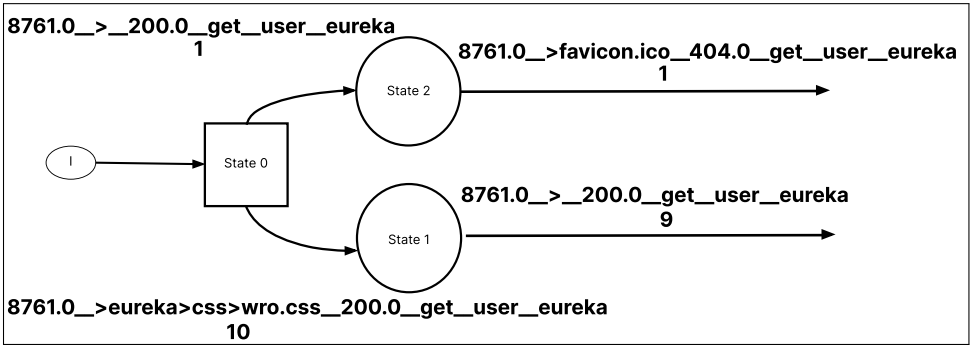


Figure 5.3: Part of state machine showing unexpected behavior.

5

Frequently occurring endpoint calls extracted from the dynamic model learned for the link between user and eureka:

Endpoint: /

- Port: 8761.0
- Call status code: 200.0
- Call direction: from user to eureka
- Call frequency: 38

Endpoint: /eureka/css/wro.css

- Port: 8761.0
- Call status code: 200.0
- Call direction: from user to eureka
- Call frequency: 10

Figure 5.4: Most frequent calls for unexpected behavior.

trigger the missing runtime event, and (3) human-readable call details extracted for the previous point. Figure 5.5 provides a snapshot of this set of details. Furthermore, the state machines learned for each involved service are presented on the HTML page.

Sequences that occurred in the dynamic model that should produce run-time behaviour for link between order and turbine

- user →(implicit) zuul → order → turbine
- zuul → order → turbine

For the occurred sequences, these are the unique sequence of endpoints (parameters) that were used in the sequence

- Sequence: user → zuul → order → turbine
 - Call started with "/order/line". Then followed by call with "/line".
 - Call started with "/order/". Then followed by call with "/"
 - Call started with "/order/18". Then followed by call with "/18".
 - Call started with "/order/11". Then followed by call with "/11".
- Sequence: zuul → order → turbine
 - Call started with "/line".
 - Call started with "/18".
 - Call started with "/3".
 - Call started with "/5".
 - Call started with "/11".
 - Call started with "/form.html".
 - Call started with "/"

Figure 5.5: Example details for dynamic non-conformance.

5.4 Tool Evaluation

Performance Analysis. We evaluated CATMA’s performance in terms of time to detect non-conformances in MSA. For this, we selected 4 DFDs of open-source MSAs from the dataset created by Schneider et al. [15], deployed these MSAs, and created state machine models for them. Then, we ran CATMA on the obtained models and measured the time of the analysis. Table 5.1 presents the time for analyzing the 4 selected MSAs (averaged over 10 executions per MSA). This evaluation allows us to quantify the benefits of utilizing CATMA compared to manual analysis.

The data clearly demonstrates that CATMA substantially accelerates the analysis process. While developers often invest days in resolving issues (as reported in the study conducted by Zhou et al. [21]), our tool accomplishes the same task in a matter of min-

¹<https://github.com/shabbirdwd53/springboot-microservice>

²<https://github.com/ewolff/microservice>

³<https://github.com/spring-petclinic/spring-petclinic-microservices>

⁴<https://github.com/sqshq/piggymetrics>

Table 5.1: CATMA’s performance statistics on multiple MSAs

Name	#LOC	# Services	# Detected Non-Conformances (static / dynamic)	Avg. Runtime (seconds)
<i>Springboot-Microservice</i> ¹	879	9	0 / 16	4.3
<i>Microservice Sample</i> ²	3117	7	2 / 1	3.0
<i>Spring PetClinic</i> ³	3990	12	1 / 26	78.6
<i>Piggy Metrics</i> ⁴	9977	17	3 / 11	53.9

Table 5.2: Trade-off between the size and correctness.

Avg. # Edges	Avg. # Nodes	Avg. Recall	Avg. Specificity	Avg. Balanced Accuracy
1	1	0.0	1.0	0.5
127	99	0.368	0.998	0.683
790	646	0.904	0.990	0.947
1982	1843	0.920	0.986	0.953
4714	4570	1.0	0.978	0.989

utes. Thus, CATMA can substantially reduce the time spent on debugging issues, offering a valuable resource for developers.

5

Pilot Study. We conducted a small-scale pilot study to investigate CATMA’s usefulness. We report an initial assessment of this pilot study based on a think-aloud interview setup with two participants. The participants were recruited from the lab of one of the authors (both with a computer science background) and have no relation to the work done for CATMA. The participants got an introduction to MSAs and were allowed to interact with CATMA before the start of the interview. During the interview, we asked several questions that would provide us insights on what are the most useful elements presented in the output generated by CATMA. A complete transcript of the interview can be found on our Figshare page [5] or in Appendix 5.6 at the end of this chapter. The following points summarize the most useful elements from CATMA’s output: (1) the model-based visualization that shows where non-conformances are detected, (2) the set of possible interpretations providing the potential causes for the corresponding non-conformance, (3) the ability to jump from the dynamic model (state machine) back to the source code, (4) static non-conformances provide insights on the security implication of the system, and (5) the type of the non-conformances: static non-conformances provide insights on the security implication.

Correctness of Dynamic Models. As the state machines approximate the provided log data, it is helpful to understand the trade-off between the correctness and the size of the model as it could influence the detection of non-conformances; a small state machine generalizes too much and introduces inaccuracies, a large state machine captures all possible behavior but might be hard to understand and process. To evaluate this aspect, we use a technique similar to the one proposed by Walkinshaw et al. [18]. Table 5.2 presents the average results computed from a 10-fold cross-validation experiment. As expected, smaller state machines introduce more inaccuracies, leading to lower balanced accuracy scores. This suggests that smaller state machines do lead to more inaccuracies in the detection of non-conformances. Furthermore, the accuracy scores appear to plateau as the state machine grows in size. This suggests that considerably larger state machines do not perform significantly better in the detection of non-conformances and selecting the largest possible model for the detection is redundant. Learning a moderate-sized state machine from input data should provide reliable performance for detecting non-conformances.

5.5 Related work

Several studies have demonstrated that architectural software representations can assist developers during manual system analysis activities [3, 7, 13]. To automate such processes, several approaches in the related literature combine static and dynamic analysis for architecture reconstruction of MSAs. *MicroArt* presented by Granchelli et al. [6], *MiSAR* presented by Alshuqayran et al. [2], and *μ TOSCA* presented by Soldani et al. [16] all extract the list of microservices statically by parsing deployment files. Connections between them are detected dynamically by leveraging service discovery services that exist in the analyzed applications or by injecting different monitoring tools. *VMAWV* presented by Ma et al. [11] instead queries existing service discovery services to retrieve the list of services and uses static analysis to detect connections. While these approaches combine static and dynamic analysis, none of them compare complete architectural models obtained via the two techniques. Since our approach performs this comparison to identify non-conformances, we believe it to be novel in this regard.

The approach *DOMICO* by Zhong et al. [20] also detects non-conformances between system representations of different stages in the development process, however, they compare the intended design (UML) against the actual implementation (static model). This approach is partly based on the approach introduced by Murphy et al. [12]. The proposed approach by Murphy et al. computes a reflexion model by finding differences between the architectural model extracted from the source code and the mental (architectural) model constructed by a system developer. Both approaches detect non-conformances between design and implementation, whereas CATMA detects non-conformances between the system's implementation and deployment.

5.6 Conclusion & Future Work

We present CATMA, a tool for automatically conducting conformance analysis of MSAs. It detects possible non-conformances by computing differences between a statically and a dynamically obtained architectural model of the MSA. Found non-conformances are visualized in an easily accessible way. Further, a set of possible interpretations is generated, showing the non-conformances' potential causes. In a preliminary evaluation, CATMA showed promising results in terms of performance as well as usability. In our evaluation, CATMA identified a non-conformance in an open-source MSA on GitHub. A misconfiguration in the Hystrix ⁵ monitoring dashboard prevented stream data from being visualized as intended in the implementation. This is a good example of a non-conformance between the intended and observed behaviors of the MSA. We notified the developers and our fix was accepted ⁶. Hence, CATMA has already shown its first –albeit small– impact on MSA.

As future work, we will extend CATMA with a more intelligent technique for selecting suitable interpretations for found non-conformances. Further, the approach would benefit from additional validation activities concerning its usefulness and possible enhancements. We plan a user study with developers in which they identify non-conformances with the help of CATMA. Finally, we will investigate the feasibility of using other types of models as input and the detection capabilities of other non-conformances.

⁵<https://github.com/Netflix/Hystrix>

⁶<https://github.com/ewolff/microservice/pull/30>

References

- [1] D2.5 dev-time vs run-time model comparison techniques - report and prototype.
- [2] Nuha Alshuqayran, Nour Ali, and Roger Evans. Towards micro service architecture recovery: An empirical study. In *2018 IEEE International Conference on Software Architecture (ICSA)*, pages 47–4709, 2018.
- [3] E. Arisholm, L.C. Briand, S.E. Hove, and Y. Labiche. The impact of uml documentation on software maintenance: an experimental evaluation. *IEEE Transactions on Software Engineering*, 32(6):365–381, 2006.
- [4] Clinton Cao, Agathe Blaise, Sicco Verwer, and Filippo Rebecchi. Learning state machines to monitor and detect anomalies on a kubernetes cluster. In *Proceedings of the 17th International Conference on Availability, Reliability and Security, ARES '22*, New York, NY, USA, 2022. Association for Computing Machinery.
- [5] Clinton Cao, Simon Schneider, Nicolás E. Ferreyra Diaz, Sicco Verwer, A. (Annibale) Panichella, and Riccardo Scandariato. Appendix for 'CATMA: Conformance Analysis Tool for Microservice Applications'. 10 2023.
- [6] Giona Granchelli, Mario Cardarelli, Paolo Di Francesco, Ivano Malavolta, Ludovico Iovino, and Amleto Di Salle. Microart: A software architecture recovery tool for maintaining microservice-based systems. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 298–302, 2017.
- [7] Carmine Gravino, Giuseppe Scanniello, and Genoveffa Tortora. Source-code comprehension tasks supported by uml design models: Results from a controlled experiment and a differentiated replication. *Journal of Visual Languages & Computing*, 28:23–38, 2015.
- [8] Xiaofeng Guo, Xin Peng, Hanzhang Wang, Wanxue Li, Huai Jiang, Dan Ding, Tao Xie, and Liangfei Su. Graph-based trace analysis for microservice architecture understanding and problem diagnosis. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2020*, page 1387–1397, New York, NY, USA, 2020. Association for Computing Machinery.
- [9] Abdelhakim Hannousse and Salima Yahiouche. Securing microservices and microservice architectures: A systematic mapping study. *Computer Science Review*, 41:100415, 2021.
- [10] Valentina Lenarduzzi and Annibale Panichella. Serverless testing: Tool vendors' and experts' points of view. *IEEE Software*, 38(1):54–60, 2021.
- [11] Shang-Pin Ma, I-Hsiu Liu, Chun-Yu Chen, Jiun-Ting Lin, and Nien-Lin Hsueh. Version-based microservice analysis, monitoring, and visualization. In *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*, pages 165–172, 2019.

- [12] G.C. Murphy, D. Notkin, and K.J. Sullivan. Software reflexion models: bridging the gap between design and implementation. *IEEE Transactions on Software Engineering*, 27(4):364–380, 2001.
- [13] Simon Schneider, Nicolas E. Diaz Ferreyra, Pierre-Jean Queval, Georg Simhandl, Uwe Zdun, and Riccardo Scandariato. How dataflow diagrams impact software security analysis: an empirical experiment. In *SANER*, 2024.
- [14] Simon Schneider and Riccardo Scandariato. Automatic extraction of security-rich dataflow diagrams for microservice applications written in java. *Journal of Systems and Software*, 202:111722, 2023.
- [15] Simon Schneider, Tufan Özen, Michael Chen, and Riccardo Scandariato. microsecond: A dataset of security-enriched dataflow diagrams for microservice applications. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, pages 125–129, 2023.
- [16] Jacopo Soldani, Giuseppe Muntoni, Davide Neri, and Antonio Brogi. The mtosca toolchain: Mining, analyzing, and refactoring microservice-based architectures. *Software: Practice and Experience*, 51(7):1591–1621, 2021.
- [17] Sicco Verwer and Christian A. Hammerschmidt. flexfringe: A passive automaton learning package. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 638–642, 2017.
- [18] Neil Walkinshaw, Ramsay Taylor, and John Derrick. Inferring extended finite state machine models from software executions. *Empirical Software Engineering*, 21(3):811–853, jun 2016.
- [19] Muhammad Waseem, Peng Liang, Mojtaba Shahin, Aakash Ahmad, and Ali Rezaei Nassab. On the nature of issues in five open source microservices systems: An empirical study. In *Proceedings of the 25th International Conference on Evaluation and Assessment in Software Engineering*, EASE '21, page 201–210, New York, NY, USA, 2021. Association for Computing Machinery.
- [20] Chenxing Zhong, He Zhang, Huang Huang, Zhikun Chen, Chao Li, Xiaodong Liu, and Shanshan Li. Domico: Checking conformance between domain models and implementations. *Software: Practice and Experience*, 2023.
- [21] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding. Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study. *IEEE Transactions on Software Engineering*, 47(2):243–260, 2018.
- [22] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Dewei Liu, Qilin Xiang, and Chuan He. Latent error prediction and fault localization for microservice applications by learning from system trace logs. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 683–694, New York, NY, USA, 2019. Association for Computing Machinery.

Appendix A: Interview Transcript of CATMA's Pilot Study

Below is the transcript of the interview done for the qualitative evaluation of the tool. The interview took place on the 26th of May 2023.

Interviewer: There are a few questions that I would like to ask you regarding how intuitive the non-conformances are presented to you in the model-based visualization.

Interviewer: The first question is how many different types of non-conformances are presented to you in the model-based visualization?

Participant 1: Two

Participant 2: Two

Interviewer: How many non-conformances were detected for each type of non-conformance?

Participant 1: Two static non-conformances and one dynamic non-conformance.

Participant 2: Two static and one dynamic.

Interviewer: How many non-conformances were detected for each type?

Participant 1: Two static non-conformances and one dynamic non-conformance.

Participant 2: Two static and one dynamic.

Interviewer: As mentioned before the start of the interview, you are a developer of a microservice application and using our tool to find discrepancies between the design and the deployment of your application. Based on two different types of non-conformances that were detected and presented in the model-based visualization, which type of non-conformance would you think would be more important (which one has more weight)?

Participant 1: I think static non-conformance would be the most important type of non-conformance for me because it is suspicious that there is runtime behavior occurring that I did not intend to have in the design of my system. I gave less importance to dynamic non-conformance as it might be the case that sometimes you did not run the system long enough to cover every part of the system. Intuitively, I think that dynamic non-conformances might return more false positives than static non-conformances. I think that any occurred runtime behavior that is not specified in the static model is by definition weird to me.

Interviewer: To follow up on your answer, you gave higher importance to static non-conformances. Did you also give a higher importance to this type of non-conformance as it might reveal potential security issues in your system?

Participant 2: Yes, this type of non-conformance provides a bit more insight into the security implications of my system. I would put a higher priority on looking at this type of non-conformance first.

Participant 2: For me, the answer would be it depends. Static is important if you have a configuration set up for the system and there is behavior that is occurring that you have

never intended within the design, that would be weird. But dynamic is important because if you wanted a particular behavior to happen and it is not happening e.g. communication with the authentication service, then that is also important to know. I would actually weigh them both as equally important.

Interviewer: If you look at the model-based visualization that is presented to you, what kind of information is missing or could be added to the model in order to understand the visualization of the detected non-conformances even better?

Participant 1: What I was wondering when I was looking at the model-based visualization is what is the meaning of the user node in the model-based visualization? Is this node used to represent an actual user or is this an actual microservice implemented in the system? Because I would be interested in how a user actually enters the system or at least see where a user can enter the system.

Interviewer: So you mean some information that explains the nodes in the model-based visualization?

Participant 1: Yes, that is for sure. So some information on the context would be nice.

participant 1: Also, I think it would be nice to see the frequency of the communication links that occurred between the services.

Participant 2: Adding frequency information on the links in the model-based visualization would indeed be nice to see what is happening quite often in the system. Besides the frequency, I think that adding the direction of the communication between the services would also be nice, together with its corresponding frequency. This would help understand which direction of communication is much more common. Finally, highlighting the entry node in the model would be nice as you would have an idea of where the initial interaction starts between the user and the system, and what possible paths can be taken within the system.

Interviewer: Regarding the last part of your answer, don't you think it better to first show a global overview of the system and then do a deep dive into the non-conformances to understand the sequential behavior of the system?

Participant 2: Yes, so from what I know of global visualization dashboards for analysts that are trying to triage events, is that they would like to see the global picture first and then they would like to drill deeper into the details. So just as what I see now from the model-based visualization, you get sort of a global overview, and then you provide a manner to do a more local inspection of non-conformances.

Interviewer: Now I would like to move on to the interpretations that are generated by our tool and I would like to ask a few questions on how intuitive the interpretations are presented to you.

Interviewer: Which parts of the generated interpretations are most useful for you to understand/interpret a detected non-conformance?

Participant 1: For me, the most intuitive parts, or actually the parts that I understood well are the first three parts of the HTML pages. So like the block that says "Potential interpretations for the non-conformance". This block provides helpful suggestions, if I have to start to debug this, then these suggestions are very helpful starting points for the debugging process. The state machine model that is shown was a bit harder to understand for me. The top ten calls would actually be useful for me but I think this information is more relevant when I actually dive into the code during the debugging process.

Interviewer: Do you think that the top 10 calls would resolve what was mentioned earlier on providing different levels of overview to the user?

Participant 1: Yes, that would be nice because the potential interpretations are very high-level. I would need to see the code before I could get more from the other parts that are shown on the HTML pages.

Interviewer: And regarding the dynamic models, what was not intuitive for you?

Participant 1: I am missing a bit of context in order to read and understand the dynamic model. There is information presented in the nodes and edges of the model but I am not completely sure how to interpret that part of the model. I recognize terms that are shown on the edges of the dynamic model but I do not know how it relates back to the terms shown in the model-based visualization.

Participant 2: First of all, the interpretations are nice because if this is a holistic list of interpretations, then it is nice for the developer as they could be thinking in one single perspective to debug a detected non-conformance. The list that is generated by the tool would provide an inventory of perspectives that also could be used to debug the non-conformance. However, the interpretations are a bit too high level to be actionable. One way to improve upon this point is by adding instances directly after the high-level interpretation. This way you link the high-level interpretation to starting points at which the developer could look to debug the detected non-conformance.

Second of all, the dynamic models that are visualized on the HTML pages are a bit hard to interpret because there are a lot of symbols shown in the models that might be clear what it means. For example, you have symbols such as “fin”, “path”, or the list of accepting/rejecting traces shown in the models but it is not clear what these symbols mean. Maybe these might be useful for the developer later on when they actually dive into the code but currently, these types of information seem to be irrelevant and clutter the models. I think you could include a post-processing step in our tool to remove or filter out irrelevant information from the dynamic models.

Third of all, the top ten calls that are shown on the HTML pages might be a bit confusing when you are dealing with non-conformances. My intuition is that non-conformances should be rare and having the words “top ten” could create confusion.

Interviewer: Do you think that it would have been better if it was rephrased to “unique calls” instead of “top ten calls”?

Participant 2: Yes, and I think that if you look from the perspective of the developer, the top ten call information might be helpful indeed when they are actually diving into the code to debug the non-conformance. For example, if there is a repeating wrong port number that occurs very often for a non-conformance, then as a developer, I would understand where the mistake might be located. So this type of information would definitely help narrow down where the developer could look during the debugging process.

Interviewer: How useful is a link from a detected non-conformance back to the line of source code? To clarify a bit more, if you have a look at the dynamic non-conformance, you were able to click on the edges of the model or click on the arrows shown in the sequences, to dive back into the line of source code that is responsible for generating the runtime behavior.

Participant 1: I think it is helpful for the debugging process but we didn’t do any actual debugging during this experiment so I cannot provide suggestions on what might be miss-

ing from the information that is presented. I do realize when I click on the transitions or the arrows, it would bring me back to a line that shows an import of a library. I am not sure whether an import would provide me with enough information but generally, the link back to the source code would be useful as it gives you starting points for the debugging process.

Interviewer: To explain a bit more about your observation, if a runtime behavior is linked back to an import statement, then the behavior is produced by a call within the library and not really code that you have written yourself.

Participant 2: So in my case, the link back to code seems to always lead me back to the class instead of a line of code but if it is redirected to a more interesting line of code, after your extra explanation, then I think that it would be useful for the developer to have this link for the debugging process. But for this experiment, we did not do any debugging ourselves so I cannot provide any suggestions on how it can be improved.

Interviewer: Based on an industrial study that I presented at the start of the experiment, developers usually take days, sometimes even weeks, to find the cause for a fault that occurred in an MSA. If you have the information that is presented by our tool, do you think it would help you understand and resolve a non-conformance?

Participant 1: I personally do not have too much experience with this but I think it will be handy to get some starting points that could help resolve the non-conformance. However, there is a risk that the starting points could throw you off track.

Interviewer: Do you think that if we provide a metric (like confidence level) might help reduce the risk that a developer might be thrown off track?

Participant 1: Yes showing confidence will be very useful and I also think that it would be having this measure that shows how often a starting point, provided by the interpretations, was a good starting point for debugging a non-conformance. So maybe something like how often a starting point was used by the developer and the starting point helped resolve the non-conformance.

Interviewer: So you are suggesting that there should be another user study to gather these statistics?

Participant 1: Yes, that would be indeed very useful.

Participant 2: I think the tool should be put in context what the debugger is doing right now and where in the debugging pipeline the tool fits. I do not have much experience being a developer so I cannot give you a definitive answer to your question. However, I do think that the information provided by your tool does provide valuable insight that could speed up the process to resolve a non-conformance. The question is whether the information is correctly interpreted by the developer. From the answer that I mentioned earlier, there are symbols presented in the dynamic models that do not seem to be clear or hard to interpret. Removing the irrelevant information would probably help with the interpretation of the models. Furthermore, I also think that another user study might be helpful for gathering data on how quickly developers can resolve non-conformances using your tool.

Interviewer: So as the last question for this interview, do you have any other improvement suggestions that you would like to provide to our tool?

Participant 1: The dynamic models are a bit hard to interpret. Doing what was mentioned by participant 2 earlier on removing irrelevant information would already help a

lot with the interpretations of the dynamic models. Also, the transitions could be clarified a bit more as I am missing some context to understand/interpret the sequential behavior. Furthermore, a bit more context could be given to better understand the sequences shown on the HTML pages.

Participant 2: So about the sequences shown on the HTML pages, something that you could add is whether these sequences are an exhaustive list of a sample of all sequences. Finally, I will repeat the suggestion that I already gave before: the interpretations are very high-level and not actionable. Providing instances directly after these interpretations could make them more actionable.

Part III

Automated Testing with Model Inference

6

Automated Test-Case Generation for REST APIs Using Model Inference Search Heuristic

6

The rising popularity of the microservice architectural style has led to a growing demand for automated testing approaches tailored to these systems. EvoMaster is a state-of-the-art tool that uses Evolutionary Algorithms (EAs) to automatically generate test cases for microservices' REST APIs. One limitation of these EAs is the use of unit-level search heuristics, such as branch distances, which focus on fine-grained code coverage and may not effectively capture the complex, interconnected behaviors characteristic of system-level testing. To address this limitation, we propose a new search heuristic (MISH) that uses real-time automaton learning to guide the test case generation process. We capture the sequential call patterns exhibited by a test case by learning an automaton from the stream of log events outputted by different microservices within the same system. Therefore, MISH learns a representation of the system-wide behavior, allowing us to define the fitness of a test case based on the path it traverses within the inferred automaton. We empirically evaluate MISH's effectiveness on six real-world benchmark microservice applications and compare it against a state-of-the-art technique, MOSA, for testing REST APIs. Our evaluation shows promising results for using MISH to guide the automated test case generation within EvoMaster.

6.1 Introduction

In recent years, the microservice architectural style has gained significant traction among companies developing modern (web) applications. This shift is largely driven by the benefits provided by this architectural style: it enables developers to develop, maintain, and deploy the components (known as microservices) of their applications independently. Microservices interact with each other via REST APIs, and a request by a client may involve a sequence of REST API calls between microservices to get the corresponding response. Despite the benefits, microservices are notoriously known to be hard to debug once a fault has occurred due to their distributed nature. Different studies have shown that developers usually spend days debugging a fault, making it a time-consuming process [34, 57].

As with any software system, writing system-level tests could mitigate potential faults after deployment. However, manually crafting these tests for microservices is not only labor-intensive but also prone to incomplete coverage as developers might not cover all possible cases that could occur in the system due to their complexity [34, 42, 51, 52]. These challenges have led to the growing need of tools to generate test cases for microservices automatically. EvoMaster is a state-of-the-art tool that uses Evolutionary algorithms (EAs) [7] to create test cases for microservice applications with web APIs. Specifically, the EA continuously generates and evolves a pool of test cases that can be executed against the system under test (SUT). Each test case represents a sequence of REST API calls for a particular type of resource. Test cases are evolved based on a set of objectives (e.g., line coverage, branch distances, etc.) used to define the fitness of the test cases. Typically, randomized operations such as mutation and crossover are used to evolve the test cases. Furthermore, different (randomized) operations are used to select test cases to keep within the pool.

Despite their success, existing EA-based approaches rely heavily on unit-level search heuristics, such as branch distances, to guide the generation of test cases. Branch distance, for example, measures how close a test case is to cover a specific branch in the code using Korel's rules [31] for branching conditions. While effective for maximizing fine-grained code coverage within individual units or services, these heuristics provide no guidance on system-level behaviors. Specifically, they lack awareness of how microservices interact and how execution flows across services. This narrow focus on unit-level objectives can result in test cases that fail to explore distributed systems' intricate, interconnected behaviors, thereby missing faults that arise from complex inter-service interactions [48].

To address this limitation, we introduce the Model Inference Search Heuristic (MISH), a novel grey-box approach to guide the automatic test case generation process. Unlike white-box techniques that rely on code-level metrics such as branch distance, MISH leverages log statements printed by the SUT during test execution, which developers insert into the code. MISH continuously collects log statements output by the SUT during test execution and uses them to learn a finite state automaton (or simply a state machine) in real-time. Each test case is converted into a sequence of events derived from these logs, and the inferred state machine serves as a representation of the system's behavior. We learn a state machine using the online learning heuristic proposed by Baumgartner and Verwer for FlexFringe [12], a framework for learning state machines. Our approach uses a single objective to define the fitness of a test case, the path traversed within the state machine. To the best of our knowledge, we are the first to apply real-time state machine

learning to the task of automated test-case generation for microservices' REST APIs.

As state machines capture sequential behaviors exhibited by a SUT, we can leverage this information to capture the required sequential pattern of REST API calls and create the corresponding sequential behavior within the SUT. This way, we can preserve sequential patterns that can be used to generate test cases for complex interconnected behaviors between microservices. We believe these sequential patterns will lead to better code coverage and fault detection.

To assess MISH's feasibility and effectiveness, we implemented a prototype of MISH within EvoMaster and tested it against six different real-world microservice applications provided by the EvoMaster Benchmark (EMB) dataset [11]. Additionally, we compared MISH against the state-of-the-art many-objective algorithm, MOSA, for testing REST APIs. An implementation of MOSA has already been provided within EvoMaster. Our evaluation results indicate that MISH performs comparably to MOSA and can even surpass MOSA in specific scenarios while utilizing only a single objective to evaluate fitness for test cases. Moreover, our results indicate that MISH is more efficient than MOSA in covering test targets at the start of the search. This suggests that MISH captures essential patterns that could lead to quicker discovery of unseen targets within the SUT.

Our main contributions are summarized as follows:

- A novel approach, MISH, that uses real-time automaton learning as a search heuristic to generate system-level test cases for REST APIs automatically.
- A prototype of MISH implemented within EvoMaster, which can be used for replication studies.
- A comparison of MISH against the state-of-the-art algorithm (MOSA) for system-level test-case generation on six different microservice applications.

6.2 Background & Related Work

This Section describes the concepts used in this work, this would help the reader understand the methods utilized in MISH. Additionally, we list related works for each concept.

6.2.1 Automated Test Case Generation using EAs

EAs are widely applied within the field of search-based software testing to automate different software testing tasks such as test case generation [26, 40], minimizing test suites [54], and prioritizing test cases [13]. Prior studies have shown that automating these tasks has significantly reduced the time that developers spend on testing and debugging their system [2, 47]. Several automated software testing tools also incorporate EAs to generate test cases for various level of testing, such as EvoSuite [26], EvoMaster [6], SUSHI [15], EXSYST [28], and Sapienz [1].

Inspired by the biological evolution process in nature, an EA starts with an initial population of randomly sampled test cases. In the context of software testing, a test case represents an individual in the population, and the statements in a test represent its genes. EA runs the test cases to determine their fitness values based on a predetermined fitness function (e.g. line coverage). The EA then generates offspring from the population of test cases using genetic operations such as crossover or mutation. The offspring are also run

to determine their fitness values. Finally, EA selects the best candidates from the parent and offspring population to form the new population of test cases that survive to the next iteration of the EA. This process (evaluation, generation of offspring, and selection) is repeated until the stopping criterion is met (e.g., search budget time is exhausted).

6.2.2 Automated Testing of REST APIs

Representation State Transfer (REST) APIs are predominantly used for communication or requesting resources from a (micro) service. These are typically done by executing (Hypertext Transfer Protocol) HTTP calls towards pre-defined endpoints. With the increasing adoption of the microservice architectural style, testing REST APIs automatically has also become an important topic of research. Testing REST APIs comes in two flavors: black-box or white-box. The former does not have access to the source/byte code of the system, meaning no information on the SUT's internal structure can be used to guide the test case generation. The latter approach has access to the internal structure and can use this information to generate test cases to trigger specific parts of the SUT.

RESTTESTGEN [50], RESTest [38], and RapiTest [24] are methods developed to automatically test REST APIs using black-box heuristics. EvoMaster also provides the option to generate test cases following in black-box mode [8]. These typically use the API specifications of the SUT to generate test cases. For the white-box approach, many state-of-the-art techniques are built as search algorithms within EvoMaster. The Many Independent Objective (MIO) algorithm is a state-of-the-art technique built within EvoMaster [5]. This algorithm treats each test target as an independent objective and tries to find a test case that covers the objective. The aim is to find a test suite that covers as many (independent) objectives as possible. MIO keeps track of a population of test cases for each objective and creates the final test suite by taking the union of the best test cases from all populations. The Many Objective Sorting Algorithm (MOSA) is another state-of-the-art technique built within EvoMaster. Proposed by Panichella et al. [40], this algorithm treats test case generation as a multi-objective optimization problem. Instead of optimizing each objective separately, as with the MIO algorithm, all objectives are optimized simultaneously in MOSA. Unlike MIO, MOSA keeps track of a single population of test cases. The final result is a minimal test suite containing the test cases covering as many objectives as possible.

6.2.3 State Machine & Testing REST APIs

A state machine, also known as a finite state automaton, is a mathematical model for capturing the sequential behavior exhibited by a SUT. Formally, it is defined as a quintuple $(\Sigma, Q, q_0, \delta, F)$ [45], where:

- Σ is a finite alphabet.
- Q is a finite set of states.
- $q_0 \in Q$ is a unique start state.
- $\delta : Q \times \Sigma \rightarrow Q$ is the transition function, denoting which state the state machine will transition to based on the current state that it is in and the next input symbol $a \in \Sigma$.

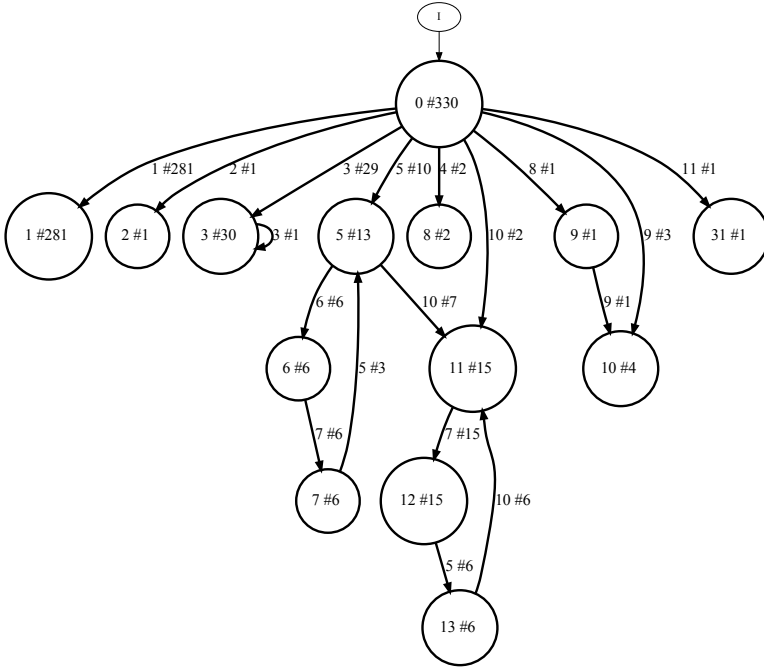


Figure 6.1: Example state machine learned for a SUT S used in the experiments. Running the trace $t = [10, 7, 5, 10]$, S will traverse states 0, 11, 12, 13, and end in state 11. This denotes the sequential behavior S exhibited for trace t .

- $F \subseteq Q$ is the set of final states.

Given an arbitrary trace t containing an arbitrary number of input symbols and a SUT S , the state machine can be used to understand the sequential behavior that S exhibited by looking at the sequence of traversed states. In this work, we use a slightly different definition of a state machine: we do not explicitly have final states, any state within the state machine is considered a final state. Figure 6.1 provides an example of a state machine learned from a microservice application used in our evaluation experiments. The number after the #-symbol denotes the visit frequency of a state/transition, and the number before the symbol denotes the state ID number or the transition symbol (i.e., the symbol for triggering the transition).

There are two primary approaches to learning state machines from data: *active* and *passive* learning. The former uses the well-known L^* algorithm proposed by Angluin [4]. The algorithm continuously queries the SUT for outputs by providing different inputs. The learning process starts by collecting input-output pairs to construct a state machine. The algorithm then tries to find counterexamples to verify the correctness of the state machine. If a counterexample is found, the state machine is updated. These two steps are repeated until no more counterexamples can be found.

In contrast to the active learning approach, the passive learning approach does not need to query the SUT actively to collect data for learning. It solely collects output data

of the SUT (i.e., only listening to the output) and constructs a model from the collected data. The passive learning approach uses state-merging algorithms, which were shown to be most effective for learning a model within this approach [32].

Several works have shown the effectiveness of state machines in testing and verification of the behavior of different systems [23, 25, 29, 35, 39, 56]. However, the focus of these works was not on automated test-case generation for REST-APIs. Cao et al. have applied state machines on REST-APIs [16, 17, 20] for different purposes. However, the focus of these two works was also not on automated test-case generation. Several works have specifically used state machines to generate test cases automatically for REST APIs. These approaches typically extract a state machine from the UML diagrams generated for the microservices [3, 41, 55]. These approaches assume that such diagrams are available for the SUT and are up to date with the latest version of the SUT.

6.3 Approach

Our approach continuously learns a state machine model from event logs outputted by the SUT and uses the model to define fitness for the test cases that have been executed. This section details the internal workings of our novel algorithm, MISH, including (1) the transformation of log events outputted by the SUT into traces, (2) the interaction between EvoMaster and the model inference framework to update a state machine with these traces continuously, (3) the computation of the fitness for each test case, and (4) the evolution of test cases for future iterations/generations. Figure 6.2 depicts a high-level overview of MISH's workflow, and Algorithm 2 outlines the pseudo-code.

6

6.3.1 Transforming Log Events to Traces

As described in Section 6.2, a state machine is learned from traces of symbols. In the context of this work, each symbol in a trace represents an arbitrary log event outputted by the SUT, and each trace represents a sequential behavior exhibited by the SUT (based on the sequence of log events). We use a state machine to model these sequential behaviors within the SUT.

To transform event logs into traces, we first group the log events based on the start and end timestamps of the test cases. This is done to capture the sequential behavior exhibited by the SUT after running each test case. MISH requests the test cases' start and end timestamps from EvoMaster and temporarily stores them in memory (see line 3 in Algorithm 2). We compare each log event's timestamp against each test case's start and end timestamp. If the log event occurred within this period, we add it to the group created for this test case. We keep track of the order in which the log events are added to the group so as not to disrupt the sequential behavior captured in these log events. Note that each test case is executed sequentially; therefore, an arbitrary log event (occurring at a given timestamp) cannot be added to two groups.

Once the groups are formed, we proceed to generate the traces (line 4 in Algorithm 2). We generate one trace for each test case and its corresponding log group. As each log event is a string containing an arbitrary number of characters, we need to transform it into a symbol for efficient learning of a state machine. We use the Drain algorithm proposed by He et al. [30] to learn templates of log events and use these templates to map each log

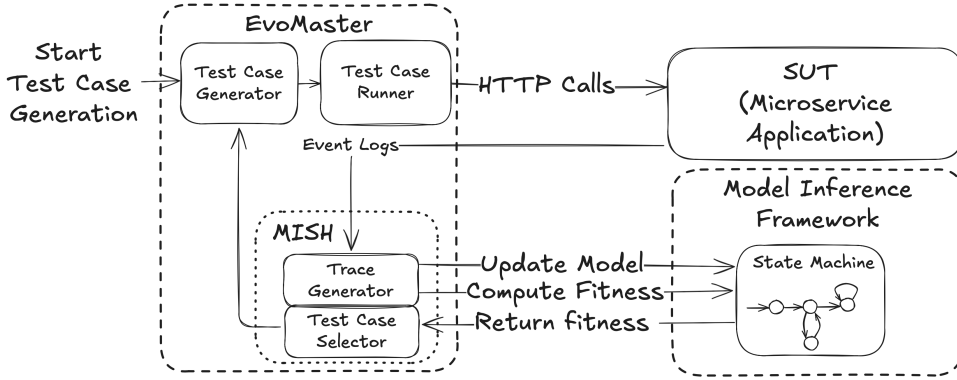


Figure 6.2: High-level overview of MISH's workflow.

Algorithm 2: MISH**Input:** Population size S , Fitness function F **Output:** A test suite TS

```

1  $P \leftarrow \text{randomPopulation}(S)$ 
2  $\text{runPopulation}(P)$ 
3  $C \leftarrow \text{collectTimeStamps}(P)$ 
4  $T \leftarrow \text{generateTraces}(C, P)$ 
5  $TS \leftarrow \text{updateArchive}(P)$ 
6  $\text{updateModel}(T)$ 
7  $\text{computeFitness}(T, P, F)$ 
8 while search budget not exhausted do
9    $O \leftarrow \text{generateOffspring}(P, S)$ 
10   $\text{runPopulation}(O)$ 
11   $C \leftarrow \text{collectTimeStamps}(O)$ 
12   $T_O \leftarrow \text{generateTraces}(C_O, O)$ 
13   $\text{updateModel}(T_O)$ 
14   $\text{computeFitness}(T_O, O, F)$ 
15   $\text{computeFitness}(T, P, F)$ 
16   $P, T \leftarrow \text{generateNewPopulation}(P, O, T, T_O)$ 
17   $TS \leftarrow \text{updateArchive}(P)$ 
18 end
19 return  $TS$ 

```

event to a single symbol. Specifically, we opted to use Drain 3 from LogPai [36] (version 0.9.11) for learning log templates as this version enables us to learn log templates on the fly and thus fits very well with the continuous learning of a state machine from log events. Each log statement in the group is mapped (in sequential order) to its corresponding log template symbol to form the trace produced by a test case. If the group for an arbitrary test case is empty, we use a single “None” word to create a trace of size one. This is done to make sure that the test case is not skipped and that a fitness value will still be assigned for this test case. The “None” word is mapped to its symbol using Drain3. Timestamps collected for the test cases and groups of log statements are cleared from memory once we have finished generating the corresponding traces.

6.3.2 Continuous Learning of State Machine

MISH employs the streaming heuristic proposed by Baumgartner and Verwer [12] to continuously learn/update a state machine from the stream of traces produced by the SUT. For each generation of N test cases, N corresponding traces are generated and forwarded to the model inference framework, which updates the state machine incrementally (line 6 in Algorithm 2). Once the state machine has been updated, the framework sends a response to MISH, which then uses the updated model to compute the fitness values for the test cases. This ensures that the state machine evolves dynamically alongside the test generation process. The fitness computation process is described in the next Subsection.

6

6.3.3 Defining Fitness using State Machine

Once the state machine has been updated, MISH forwards the traces (generated from the test cases) to the model inference framework and requests it to compute their fitness (line 7 in Algorithm 2). The fitness value computed for a particular trace is assigned as the fitness value for the corresponding test case. Fitness is determined based on the path (the sequence of states) traversed by the trace within the state machine. The primary objective of MISH is to generate system-level test cases that trigger deep and interconnected interactions between microservices. Each state in the state machine represents a unique state of the SUT, and its visit frequency is computed from the traces used to update the model. We assume that test cases that lead the system into less frequently visited states (or traverse paths containing more of such states) are more likely to reveal complex and interconnected behaviors, thus contributing to higher system-level coverage. Consequently, MISH assigns higher fitness values to test cases that visit (or traverse paths through) infrequent states.

There are several ways to compute the fitness of a trace (and thus a test case) based on the state visit frequencies. We implemented two distinct fitness functions within the model inference framework. Below, we briefly describe each fitness function and provide an example of how the fitness value is calculated for an arbitrary trace using the corresponding function. For the example, we refer to the state machine and trace illustrated in Figure 6.1. Note that all traces pass through the start state (e.g., state 0); therefore, this state is excluded from fitness computations.

Lower Than Median (LM): This fitness function first computes the median state visit frequency for the path taken by an arbitrary trace t in the state machine. Then, it computes how many states in the path have a lower visit frequency than the median. To mitigate

the tendency of MISH to favor longer traces, this count is divided by the total number of states visited by the trace. If the trace only visits one state, its fitness value is the inverse of its state visit frequency, mitigating the potential bias toward short traces that visit a single, highly frequent state. Using the example state machine and trace t from Figure 6.1, the states visited by t are 11, 12, 13, and 11. Their visit frequencies are 15, 15, 6, and 15, respectively. The median visit frequency for this trace is 15. Only one state (13) has a lower visit frequency than 15. The fitness of trace t in this example is then $\frac{1}{4}=0.25$.

Weighted State Visit Frequencies (WS): this function first collects the visit frequency of the states visited by an arbitrary trace t . These frequencies are sorted in ascending order. A weighted sum is then calculated by multiplying each state's visit frequency by its index in the sorted list (starting from 1). The fitness value of the trace is defined as the inverse of this weighted sum. Using the same trace t from Figure 6.1, the sorted state visit frequencies are 6, 15, 15, and 15. The weighted sum of these frequencies is then $(6 \times 1) + (15 \times 2) + (15 \times 3) + (15 \times 4) = 141$. The fitness of t is then $\frac{1}{141} \approx 0.007$.

6.3.4 Evolving and Selecting Test Cases

In each iteration, MISH generates an offspring population based on the current population of test cases (see line 9 in Algorithm 2). The offspring population has the same size as the current population. Specifically, MISH uses tournament selection to select candidate test cases for the offspring population. MISH then creates a copy of the selected candidate and introduces a mutation in the copied test case. The copied test case is then added to the offspring population. MISH selects 90% of the offspring through tournament selection, and the remainder of the offspring population is filled by sampling test cases to create more diverse individuals for better exploration. We employ the same sampling strategy as MOSA. MISH then computes the fitness of the offspring population following the procedure described previously. Finally, MISH selects the new population (see line 16 of in Algorithm 2) of test cases for the next iteration by taking the top S test cases from the union of the current population and the offspring population, where S denotes the population size.

6.4 Empirical Study

This section details the empirical study we carried out to evaluate MISH's feasibility and effectiveness for generating system-level test cases. We first outline the research questions addressed in the study. Next, we detail the selection of applications and parameter settings used in EvoMaster for evaluation. Finally, we describe the metrics employed to compare the performance of MISH against MOSA.

6.4.1 Research Questions

The study aims to answer the following research questions:

- **RQ1:** *How effective is MISH in generating system-level test cases compared to the state-of-the-art MOSA algorithm?*
- **RQ2:** *In which scenarios does MISH demonstrate a good/poor performance?*

- **RQ3:** *How does MISH's coverage (test targets) over time compare to that of the state-of-the-art MOSA algorithm?*

RQ1 focuses on evaluating the overall effectiveness of MISH in generating test cases that achieve high system-level coverage and reveal complex behaviors in REST APIs compared to MOSA. With **RQ2**, we aim to investigate in detail the scenarios in which MISH generates effective test cases and the scenarios in which MISH struggles with generating effective test cases. Finally, **RQ3** aims to evaluate and compare the efficiency of MISH in achieving test coverage over time, in comparison to MOSA.

6.4.2 Selection of Applications

To evaluate MISH's effectiveness in generating system-level test cases, we benchmark MISH on six real-world microservice applications from the EMB dataset. This dataset is created by the same authors as EvoMaster and was specifically designed to benchmark test case generation approaches of EvoMaster [11]. The version number of the dataset is synced with the version number of EvoMaster. As MISH is developed within EvoMaster, we have selected this dataset for our evaluation. Specifically, we used version 2.0 of the dataset as this was the latest stable release at the time of our implementation. While version 3.2 of EvoMaster was released at the time of writing this work, we found it does not report a specific category of faults (Potential Faults) that we use to evaluate MISH's effectiveness (see Subsection 6.4.5 in the description of the metrics). Additionally, we encountered errors when attempting to write test cases to file. These two points led us to use an earlier version of the dataset for evaluation instead of the latest version. We selected only the applications in which REST APIs are used, as MISH is designed to test REST APIs. The six real-world applications that we have selected are:

- *CatWatch*: a web application that collects, processes, and stores statistics for a GitHub organization. The statistics can be requested via the REST API.
- *Features-Service*: a microservice application for managing product Feature Models. The REST API is used to define and query products and product configurations.
- *Scout-API*: A microservice application that provides a REST API to interact with a web service named "Scout"
- *ProxyPrint*: a web service for comparing and making requests to different print shops
- *LanguageTool*: an application that uses a REST API to do style and grammar checks for a piece of text.
- *OCVN*: a dashboard for conducting visual analytics on public procurement data of Vietnam.

All applications are built with JDK 8 following the instructions provided on the repository of the EMB dataset [10]. Table 6.1 depicts the recent statistics collected for the selected applications from the EMB dataset [9].

Table 6.1: Recent statistics collected by Arcuri et al. for the selected applications from the EMB dataset.

Application	# Source Files	# LOCs	# API Endpoints
CatWatch	106	9636	14
Features-Service	39	2275	18
Scout-API	93	9736	49
ProxyPrint	73	8338	74
LanguageTool	1385	174781	2
OCVN	526	45521	258

6.4.3 Version & Parameter Settings for EvoMaster

We used EvoMaster version 2.0 to implement and benchmark MISH for the reason provided in the previous subsection. We ran EvoMaster using mostly the default parameter settings, with a few modifications to the following parameters: search budget, test-case minimization, tournament selection size, and some other options that are only turned on by default for the MIO algorithm. The default search budget of EvoMaster is set to one minute. We changed the search budget to 60 minutes to give each EA enough time to initialize and explore the search space. This extended search budget aligns with previous studies [9, 27, 44]. We opted to use time as the criterion for the search budget as we deem this to be a fairer comparison between different EAs. Alternative stopping criteria include the number of HTTP calls made by the algorithm, as used by Marculescu et al.[37]. However, MISH would exhaust such a budget quickly because it uses the sequential patterns of REST API calls (captured in the state machine) to define the fitness of a test case. This rapid exhaustion would limit MISH's ability to test deeper sequences of interactions between microservices. Another stopping criterion is the number of iterations or generations an EA has executed[43]. However, this approach does not consider the complexity of the system under test; some systems require more generations to trigger deep and interesting behaviors. Using the number of generations might, therefore, halt the exploration prematurely for complex systems. Moreover, practitioners often allocate a fixed amount of time to run an EA, as argued by Stallenberg et al. [48], making time a practical and equitable stopping criterion.

Test case minimization. We turned off the post-process minimization process of EvoMaster. This process splits larger test cases with multiple HTTP calls into smaller test cases with one single HTTP call at the end of the search. This type of splitting does exactly the opposite of what we are trying to tackle with MISH: it breaks the sequential patterns that MISH tries to capture in the REST API calls. Additionally, we observed that enabling test-case minimization led to a reduction in the coverage achieved by both MISH and MOSA compared to when minimization was disabled.

Tournament selection. We modified the default tournament selection size from ten to four for our experiments. A tournament size of ten is considered too large for a single-objective EA like MISH, potentially hindering its ability to effectively select and evolve promising test cases [33].

Configuration options. We activated several non-default configuration options in EvoMaster, such as smarter sampling of individuals, enhanced mutation operators, and optimized database queries. We found that activating these options led to consistently better

results (i.e., higher coverage) for both MISH and MOSA in our preliminary experiments.

6.4.4 Execution Procedure of Each Algorithm

To answer our research questions, we ran each algorithm (MOSA and MISH) 20 times on each selected microservice application to mitigate the effects of stochastic operations (e.g., mutation) on performance results. Each run is executed independently, i.e., the application and the EA are reset after each run to make sure that they are both started in a clean state for collecting performance statistics. Since MISH implements two distinct fitness functions, each version of MISH was also run 20 times per application. In total, we executed 360 runs, which would take approximately 15 days to complete if run sequentially. To speed up the evaluation process, we created a Docker image that enables us to parallelize the runs. This was inspired by the experimental setup used by Arcuri et al. [9]. Each Docker container executes 20 runs of an algorithm on a particular application. We make our Docker image and source code publicly available[18, 19] to promote the reproducibility of our study. Our Docker image does not include the option to run OCVN, as it spawns a Docker container. We chose not to run Docker containers within another Docker container due to potential performance and compatibility issues. For experiments involving OCVN, we executed all three algorithms directly on the host machine, ensuring a fair comparison by using the same hardware specifications for all runs.

6

6.4.5 Comparison Metrics

We collect and compare the algorithms based on the performance metrics used in the literature [9, 27, 37, 48]:

- Number of covered test targets: aggregated value denoting the (unique) number of search objectives (e.g. line, branch, faults, etc.) covered during the run.
- Number of potential faults: aggregated value denoting the (unique) number of 500 errors detected from the HTTP responses, and the (unique) number of discrepancies detected between API responses and the specifications in the OpenAPI schema.
- Number of potential 500 faults: the (unique) number of 500 error codes detected from HTTP responses.

To determine whether the results between the algorithms are significantly different, we performed unpaired Wilcoxon rank-sum tests with a threshold of 0.05 to compute the p -values. This is a non-parametric pairwise test for determining whether a pair of data distributions are significantly different from each other [21]. Since MOSA is our baseline method and we have two different versions of MISH, we performed a pairwise comparison between each version of MISH and MOSA. Furthermore, to determine the probability that a sample of results from one method will exceed those from the other method, we computed the effect size of the result using the Vargha-Delaney statistic [49]

6.5 Evaluation Results

We now answer our research questions based on the results achieved from our empirical study. Each research question is answered separately in the following subsections.

Table 6.2: Statistics are collected from the model inference framework.

Application	Trace Length		Learning Time (seconds)		Model Size	
	Median	IQR	Median	IQR	Median	IQR
CatWatch	5.00	7.00	0.0187	0.0039	29.00	13.50
Features-Service	2.00	2.00	0.0031	0.0005	22.00	3.00
Scout-API	10.00	4.00	0.0107	0.0025	20.00	8.00
ProxyPrint	3.00	8.00	0.0026	0.0007	44.00	7.00
LanguageTool	1.00	0.00	0.0370	0.0353	93.00	99.00
OCVN	1.00	0.00	0.0046	0.0006	8.00	4.00

Table 6.3: Median number of covered targets.

Application	MOSA		MISH - LM		MISH - WS	
	Median	IQR	Median	IQR	Median	IQR
CatWatch	2014.0	471.25	1703.5	51.25	1725.5	85.5
Features-Service	879.0	6.25	883.0	4.75	885.0	1.5
Scout-API	2223.0	68.25	2073.0	85.0	2089.5	84.5
ProxyPrint	2922.5	308.5	3302.0	127.75	3377.5	81.0
LanguageTool	3106.0	10262.5	2927.0	86.0	2948.5	87.75
OCVN	4464.5	170.0	6358.0	427.75	6083.5	230.75

6.5.1 RQ1: MISH's Performance In Generating Test Cases

Table 6.2 presents various learning statistics gathered from the model inference framework for each application. These represent the averages (medians and interquartile ranges) for one run on the respective applications. We observe that the time MISH requires to learn or update the state machine is in the order of milliseconds, which is negligible compared to the overall search budget of 60 minutes. This demonstrates the feasibility of integrating real-time state machine learning into an EA for system-level test case generation.

Tables 6.3, 6.4, and 6.5 present the performance of each algorithm for every application according to the respective comparison metric. The best-performing algorithm for each corresponding application is highlighted in bold. To address the outliers caused by random processes in EAs, we report the medians along with their corresponding interquartile ranges (IQR) instead of the mean.

From Table 6.3, both versions of MISH achieve higher coverage than MOSA on Features-Service, ProxyPrint, and OCVN. On the remaining applications, MISH demonstrates coverage results that are competitive with MOSA. MISH achieved, on average, a minimal increase (0.68%) in coverage on Feature-Service when compared to MOSA. However, MISH achieved a much larger increase in coverage on ProxyPrint and OCVN (15.7% and 42.4%, respectively). Regarding fault detection capabilities (Tables 6.4 and 6.5), MISH again outperforms MOSA on ProxyPrint and OCVN, while detecting a comparable number of faults as MOSA on the other applications. Specifically, for the potential faults detected on ProxyPrint and OCVN, MISH detected (on average) 4.5% and 13.5% more faults than MOSA, respectively. For the potential 500 faults, MISH detected (on average) 6.3% more faults on ProxyPrint and 76.5% more faults on OCVN. Tables 6.6, 6.7, and 6.5 present the results

Table 6.4: Median number of potential faults.

Application	MOSA		MISH - LM		MISH - WS	
	Median	IQR	Median	IQR	Median	IQR
CatWatch	25.0	1.25	24.0	2.0	24.0	1.0
Features-Service	32.0	0.0	31.0	0.5	31.5	1.0
Scout-API	79.0	9.5	66.0	2.0	69.0	4.25
ProxyPrint	66.0	10.25	68.0	3.0	69.0	2.25
LanguageTool	7.0	2.0	6.5	1.0	6.0	0.0
OCVN	274.5	7.0	311.5	13.0	303.0	10.5

Table 6.5: Median number of potential 500 faults

Application	MOSA		MISH - LM		MISH - WS	
	Median	IQR	Median	IQR	Median	IQR
CatWatch	10.0	1.25	10.0	2.0	10.0	1.25
Features-Service	25.0	0.0	25.0	1.0	25.0	1.0
Scout-API	59.5	9.0	47.0	2.25	50.0	4.25
ProxyPrint	32.0	5.25	33.0	2.0	34.0	2.0
LanguageTool	5.0	1.0	4.5	1.0	4.0	0.0
OCVN	49.0	4.5	86.5	9.75	79.5	10.75

of the statistical tests conducted for each comparison metric. Significant p -values (< 0.05) are highlighted in gray. As expected, there is no statistical significance in MISH's coverage performance on Feature-Service as it achieved only a minimal increase in coverage. However, MISH performed significantly better on ProxyPrint and OCVN across all three metrics, each associated with a *large* effect size. The results of our statistical tests confirm our findings presented in Tables 6.3, 6.4 and 6.5.

Although MISH achieved lower coverage in the latter applications, it is noteworthy that MISH can still detect a similar number of faults in these instances. The lower coverage performance of MISH is expected as it is a single-objective EA and single-objective EAs typically get stuck very quickly in local optima. In contrast, many-objective EAs (e.g., MOSA) can use multiple criteria to generate a wider diversity of individuals, resulting in better exploration [14, 46, 53].

6.5.2 RQ2: Where does MISH exhibit good/poor performance?

We analyzed scenarios where MISH performs well or poorly by examining one application for each case in detail. For strong performance, we selected OCVN, while for poor performance, we analyzed Scout-API. These applications were chosen due to the notable (significant) differences in the number of faults identified by MISH and MOSA.

Performance on OCVN

To understand MISH's superior performance on OCVN, we compared the faults identified by MISH-LM and MISH-WS with those identified by MOSA. Specifically, we analyzed the unique set of fault IDs detected across all runs of this application. Figure 6.3 presents

Table 6.6: Statistical significance (ρ -values and $\hat{A}_{12}$ -values) for the number of covered targets.

Application	MISH - LM vs. MOSA		MISH -WS vs. MOSA	
	ρ -value	$\hat{A}_{12}$	ρ -value	$\hat{A}_{12}$
CatWatch	< 0.01	0.21 (large)	< 0.01	0.27 (medium)
Features-Service	0.47	0.62 (small)	0.47	0.87 (large)
Scout-API	< 0.01	0.01 (large)	< 0.01	0.10 (large)
ProxyPrint	< 0.01	0.89 (large)	< 0.01	0.95 (large)
LanguageTool	< 0.01	0.10 (large)	< 0.01	0.13 (large)
OCVN	< 0.01	1.00 (large)	< 0.01	1.00 (large)

Table 6.7: Statistical significance (ρ -values and $\hat{A}_{12}$ -values) computed for the number of potential faults.

Application	MISH - LM vs. MOSA		MISH -WS vs. MOSA	
	ρ -value	$\hat{A}_{12}$	ρ -value	$\hat{A}_{12}$
CatWatch	0.06	0.37 (small)	0.06	0.37 (small)
Features-Service	0.01	0.23 (large)	0.01	0.40 (small)
Scout-API	< 0.01	0.01 (large)	< 0.01	0.02 (large)
ProxyPrint	0.02	0.69 (medium)	0.02	0.81 (large)
LanguageTool	0.01	0.29 (medium)	0.01	0.18 (large)
OCVN	< 0.01	1.00 (large)	< 0.01	1.00 (large)

the Venn diagram on the faults detected by each algorithm in OCVN. As we can observe, each version of MISH has identified more unique faults than MOSA. Notably, MISH-LM identified more than twice as many faults as MOSA.

Upon analyzing the faults identified by MOSA, we found they were related to seven distinct fault locations in the code. Of these, only two fault locations were associated with test cases involving multiple (and sequential) REST API calls. In contrast, the faults identified by MISH were spread across 38 distinct fault locations. A deeper analysis of these faults revealed a common starting pattern in the REST API calls: the first call logs in as an administrator, saving authentication information in a cookie that is subsequently used for the remaining REST API calls. This indicates a dependency between the initial call and the subsequent calls in the tests.

MISH's design explicitly aims to capture such dependency patterns in REST API calls to trigger interconnected behaviors between microservices. By leveraging model inference to identify and utilize these patterns, MISH was able to generate more effective system-level test cases. Notably, no similar patterns were observed in the test cases associated with faults identified solely by MOSA, highlighting the strength of MISH's approach.

An important observation from our evaluation is that MISH performs particularly well on applications with numerous API endpoints, such as ProxyPrint and OCVN. Additional investigation into the faults identified only by MISH on ProxyPrint revealed a pattern similar to OCVN: the first call logs in as an administrator, followed by a sequence of dependent API calls. We believe that MISH's superior performance on these two applications is due to its ability to capture and exploit the dependencies between REST API calls, which MOSA (and its many-objective design) does not explicitly account for.

Table 6.8: Statistical significance (ρ -values and $\hat{A}_{12}$ -values) computed for the number of potential 500 faults.

Application	MISH - LM vs. MOSA		MISH -WS vs. MOSA	
	ρ -value	$\hat{A}_{12}$	ρ -value	$\hat{A}_{12}$
CatWatch	0.35	0.45 (negligible)	0.35	0.41 (small)
Features-Service	0.36	0.44 (negligible)	0.36	0.63 (small)
Scout-API	< 0.01	0.01 (large)	< 0.01	0.02 (large)
ProxyPrint	0.03	0.66 (small)	0.03	0.81 (large)
LanguageTool	0.18	0.41 (small)	0.18	0.25 (large)
OCVN	< 0.01	1.00 (large)	< 0.01	1.00 (large)

6

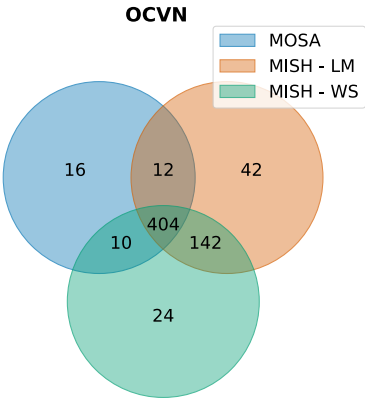


Figure 6.3: Venn diagram for the faults detected by each algorithm on OCVN.

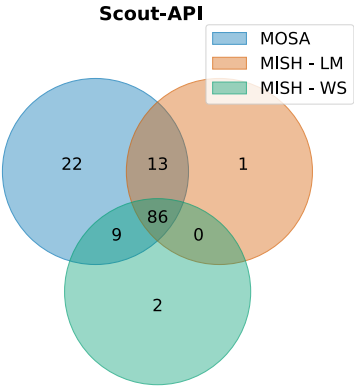


Figure 6.4: Venn diagram for the faults detected by each algorithm on Scout-API.

Performance on Scout-API

To understand MISH’s poor performance on Scout-API, we applied the same methodology used for the analysis of OCVN. Figure 6.4 presents a Venn diagram showing the distribution of fault IDs identified by the three algorithms. From the diagram, we observe that 22

unique fault IDs were identified solely by MOSA, whereas both versions of MISH together only identified three unique fault IDs that MOSA did not. Among the 22 unique faults found by MOSA, all are potential 500 faults, and 17 of these faults are related to the location shown in Listing 6.1. At this location, MOSA executed different actions in the REST API calls to trigger these faults.

For the three unique faults identified by MISH, only MISH-LM managed to find a potential 500 fault at the same location as MOSA. However, MISH-LM executed a different action to trigger it. This suggests that MOSA's superior performance can be attributed to its many-objective nature and access to additional information, such as source code. With source code access, MOSA can compute fine-grained metrics like branch distances, which can guide test-case generation more effectively by enabling diverse actions in REST API calls. MOSA leveraged this capability to discover multiple faults at the same location by varying the actions in the API calls.

In contrast, MISH, as a single-objective EA, does not have access to such information. It relies solely on log statements and the state machine inferred from them to guide test-case generation. As a result, MISH lacks the ability to make fine-grained modifications to test cases, limiting its capacity to discover faults in complex scenarios like those in Scout-API.

```
1 devscout/scoutapi/dao/UserDao_30_readUserByIdentity
```

Example 6.1: Common fault location in the 22 faults identified only by MOSA

Co-factor Analysis. To statistically understand whether there is a significant interaction between the search algorithms (MOSA, MISH-LM, or MISH-WS), the size of the application under tests (measures as the number of endpoints), and the achieved coverage, we conducted a two-way *permutation test* [22]. This test is a non-parametric alternative to the two-way ANOVA test and does not require the data to be normally distributed (unlike ANOVA). It enables us to determine whether the different algorithms achieve significantly different coverage and whether they behave statistically differently depending on the size of the application under test. For this test, we set the threshold $p\text{-value} \leq 0.05$.

The test revealed that the algorithms have a significant impact on the achieved coverage ($p\text{-value}=0.04$), confirming the results of the Wilcoxon test. Besides, there is a significant relation between the coverage achieved by the different algorithms and the size of the applications under test ($p\text{-value}<0.01$). This is expected as it is more challenging to achieve higher coverage for larger applications. The two-way test also reveals that there is a significant interaction between the algorithms and the size of the applications (independent variable) on the achieved coverage (dependent variable), returning $p\text{-value}<0.01$. This confirms our observational results that MISH seems to perform better specifically for applications with a larger number of endpoints.

6.5.3 RQ3: MISH's Coverage Over Time

To evaluate MISH's efficiency in covering test targets, we analyzed the coverage data collected from all 360 runs of the experiments. In each run, EvoMaster reported the number

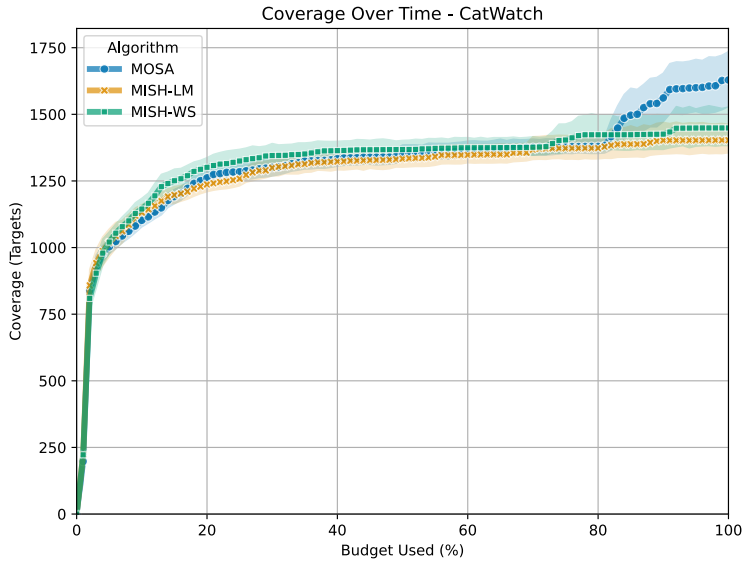


Figure 6.5: Average coverage over time on CatWatch.

6

of test targets covered by the algorithm and the corresponding search budget utilized. We grouped the coverage data by application and computed the average coverage over time for each algorithm on every application. Plotting the coverage over time enables us to assess how quickly each algorithm discovers previously unseen test targets.

Figures 6.6 to 6.10 present the average coverage over time for all three algorithms on the respective applications. Confidence intervals are included to illustrate the variance in performance. These plots show that MISH discovers unseen test targets more quickly than MOSA on almost all applications.

The one exception is *LanguageTool*, where MISH does not outperform MOSA in discovering unseen test targets. On this application, MISH's coverage performance reaches a plateau early in the run and remains stagnant for the rest of the execution. Two key factors contribute to MISH's performance on *LanguageTool*. First, *LanguageTool* has the largest number of source files and lines of code among the applications, which increases the complexity of the system under test. Second, *LanguageTool* has only two API endpoints, limiting MISH's ability to generate diverse traces. MISH relies on log statements produced by these endpoints to create traces and update the state machine. Furthermore, MISH can only adjust the HTTP request parameters and actions directed at these endpoints to identify new test targets. This limitation hinders MISH's ability to uncover more varied behaviors, particularly when the API endpoints function independently and lack dependencies that might reveal sequential patterns.

In the worst-case scenario where the two endpoints are independent (i.e., one endpoint's behavior does not depend on the other), MISH is limited to testing each API endpoint in isolation. Due to the sheer size of *LanguageTool*, MISH, as a single-objective evolutionary algorithm, is more likely to become trapped in a local optimum. Unlike

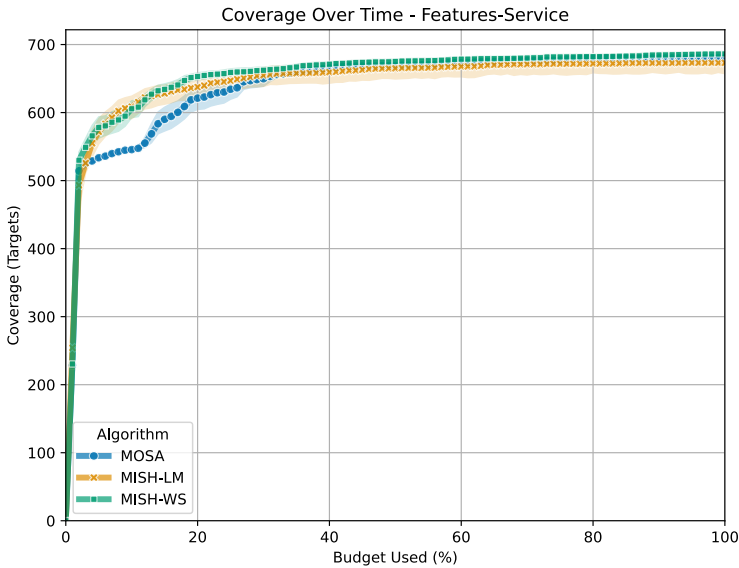


Figure 6.6: Average coverage over time on Features-Service.

MOSA, MISH lacks access to the source code, which MOSA uses to compute metrics such as branch distance to generate a more diverse set of test cases. This lack of diversity hinders MISH’s ability to escape local optima, a limitation supported by the significant variance in MOSA’s coverage performance on *LanguageTool*.

For the other applications, the coverage trends show a consistent pattern: MISH initially discovers test targets faster than MOSA but eventually reaches a plateau for the remainder of the run. This behavior is expected given that MISH is a single-objective EA without access to additional information that could diversify the population of test cases. However, it is worth noting that MISH achieves its initial performance boost solely by leveraging log statements outputted by the application. This demonstrates MISH’s ability to capture essential patterns early in the run, leading to early higher coverage.

These results suggest that MISH could benefit from a hybrid approach. A method that starts with MISH to capture essential patterns and then switches to MOSA to diversify the test cases and improve exploration could combine the strengths of both algorithms, potentially yielding better overall performance.

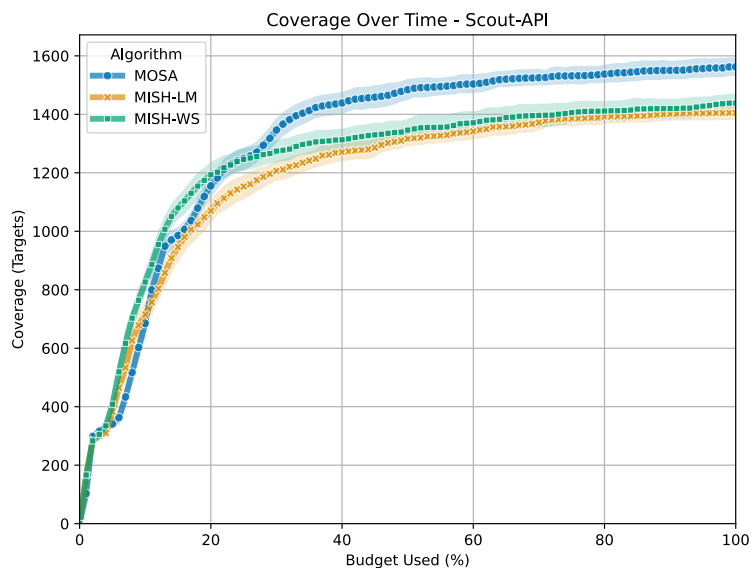


Figure 6.7: Average coverage over time on Scout-API.

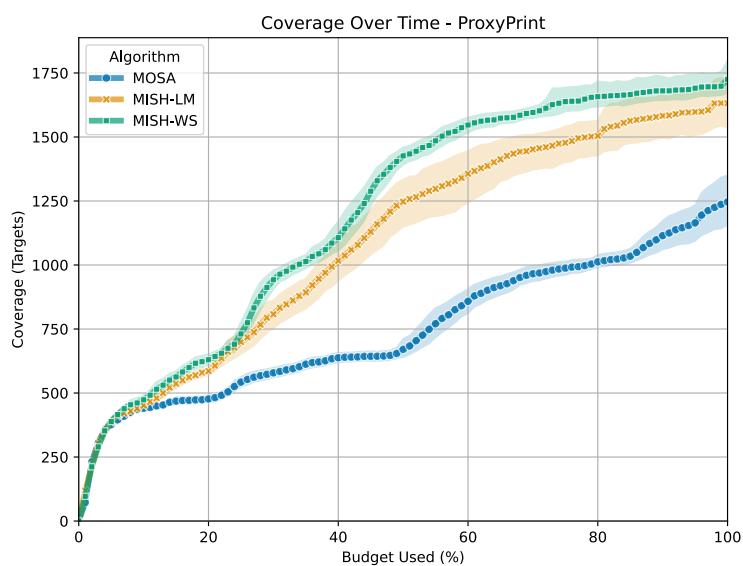


Figure 6.8: Average coverage over time on ProxyPrint.

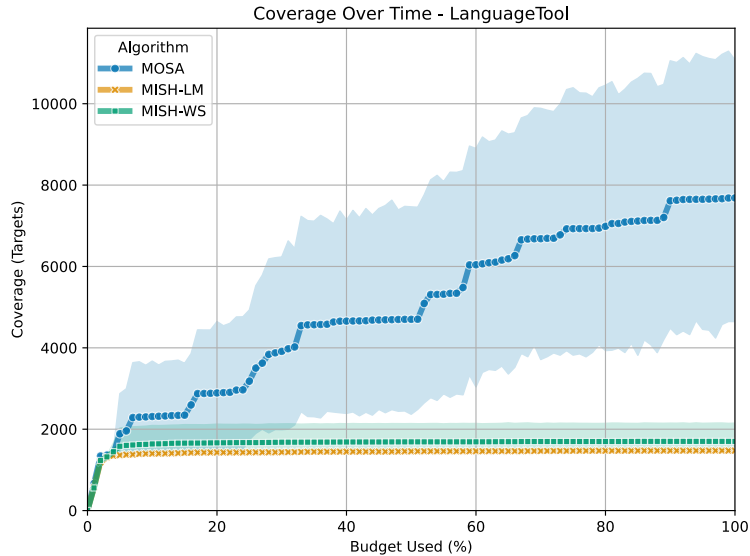


Figure 6.9: Average coverage over time on LanguageTool.

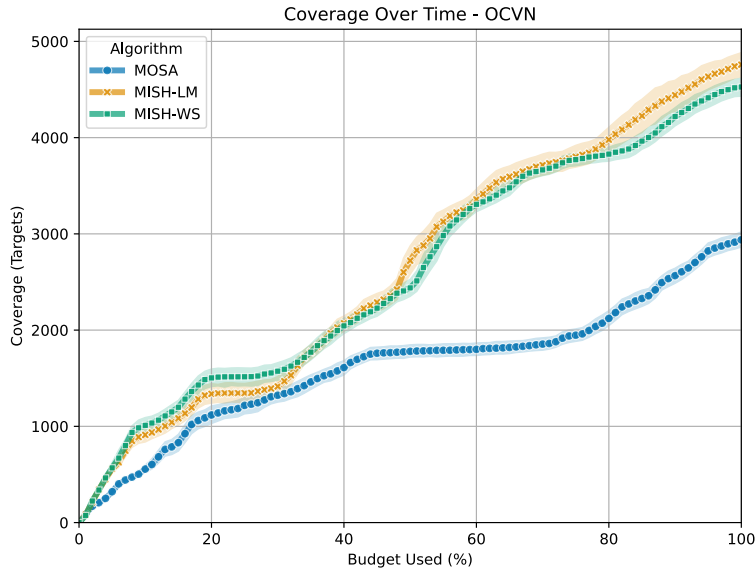


Figure 6.10: Average coverage over time on OCVN.

6.6 Threat to Validity

Concerning threats to *construct validity*, we evaluated all evolutionary algorithms (EAs) using metrics widely employed in prior studies: the number of covered targets, the number of potential faults detected, and the number of potential 500 faults detected. For the stopping criterion, we utilized a commonly used approach based on a fixed search budget measured by runtime. Specifically, we set the search budget to 60 minutes per run, consistent with prior EvoMaster research.

A potential threat to *internal validity* could arise from the integration of state machine learning into the EA for test-case generation. The additional learning time for the state machine could negatively impact the runtime of the EA. However, based on the average statistics collected from the model inference framework, the state machine learning time is in the order of milliseconds. Thus, we consider this impact to be minimal.

For threats to *external validity*, we selected six applications from the well-established EvoMaster Benchmark (EMB), which has been extensively used in prior research. Our study focused on real-world applications, excluding any artificial or synthetic applications. The selected applications vary in size, including differences in the number of source files, lines of code, and API endpoints, which helps assess the generalizability of our algorithm across diverse scenarios.

To address threats to *conclusion validity*, we followed best practices from prior studies on EvoMaster. Each algorithm was run 20 times per application, using different random seeds to account for the stochastic nature of EAs. For statistical analysis, we employed the Wilcoxon rank-sum test to evaluate the significance of differences between algorithms and the Vargha-Delaney effect size measure to quantify their magnitude.

6.7 Conclusion & Future Work

We presented MISH, a novel approach that uses real-time automaton learning as a search heuristic to guide the test case generation within EvoMaster. MISH continuously updates an automaton by gathering logs produced by the SUT after running a generation of test cases. MISH computes fitness for the test cases based on the path traversed by the log statements within the automaton. MISH uses the automaton to capture the sequence of REST API calls needed to trigger deep and connected behaviors between microservices.

We implemented MISH within EvoMaster and empirically evaluated its effectiveness in generating test cases on six microservice applications from the EMB benchmark. Our evaluation results indicate that MISH performs comparably to MOSA, a state-of-the-art technique for generating test cases for microservices, and outperforms MOSA in certain scenarios. MISH achieves this performance while utilizing only a single objective to determine fitness for test cases, whereas MOSA utilizes multiple objectives. MISH provides promising results for the use of model inference as a search heuristic in test case generation. Furthermore, our results suggest that MISH is quicker at discovering unseen targets at the start of the search compared to MOSA.

In future work, we plan to investigate the effect of utilizing MISH as part of a many-objective search algorithm (e.g., combining MOSA and MISH). We speculate that by incorporating MISH within a many-objective search algorithm, we can aid the many-objective search algorithm in discovering unseen targets quicker at the start of the search. Moreover,

we can utilize different information gathered by the many-objective search algorithm to help MISH with its search when its performance reaches a plateau. Preliminary results of the combination between MISH and MOSA is presented in the appendix of this chapter (Appendix 6.7).

In its current implementation, MISH communicates with the model inference framework via file-based interactions, which involve frequent disk I/O operations. Over many generations of test cases, this file-based communication negatively impacts performance. To address this limitation, we plan to replace the file-based communication with a specialized API. This new communication mechanism would not only improve efficiency but also enhance the robustness of MISH by mitigating potential communication failures.

References

- [1] Nadia Alshahwan, Xinbo Gao, Mark Harman, Yue Jia, Ke Mao, Alexander Mols, Taijin Tei, and Ilya Zorin. Deploying search based software engineering with sapienz at facebook. In Thelma Elita Colanzi and Phil McMinn, editors, *Search-Based Software Engineering*, pages 3–45, Cham, 2018. Springer International Publishing.
- [2] Yasaman Amannejad, Vahid Garousi, Rob Irving, and Zahra Sahaf. A search-based approach for cost-effective software test automation decision support and an industrial case study. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*, pages 302–311, 2014.
- [3] Anneliese Amschler Andrews, Jeff Offutt, and Roger T. Alexander. Testing web applications by modeling with fsms. *Software & Systems Modeling*, 4:326–345, 2005.
- [4] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987.
- [5] Andrea Arcuri. Many independent objective (mio) algorithm for test suite generation. In Tim Menzies and Justyna Petke, editors, *Search Based Software Engineering*, pages 3–17, Cham, 2017. Springer International Publishing.
- [6] Andrea Arcuri. Evomaster: Evolutionary multi-context automated system test generation. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 394–397, 2018.
- [7] Andrea Arcuri. Restful api automated test case generation with evomaster. *ACM Trans. Softw. Eng. Methodol.*, 28(1), January 2019.
- [8] Andrea Arcuri. Automated black- and white-box testing of restful apis with evomaster. *IEEE Software*, 38(3):72–78, 2021.
- [9] Andrea Arcuri, Man Zhang, and Juan Galeotti. Advanced white-box heuristics for search-based fuzzing of rest apis. *ACM Trans. Softw. Eng. Methodol.*, 33(6), June 2024.
- [10] Andrea Arcuri, Man Zhang, Amid Golmohammadi, Asma Belhadi, Juan P. Galeotti, Bogdan Marculescu, and Susruthan Seran. WebFuzzing/EMB: EvoMaster benchmark (EMB): A set of web/enterprise applications for experimentation in automated system testing. <https://github.com/WebFuzzing/EMB>.
- [11] Andrea Arcuri, Man Zhang, Amid Golmohammadi, Asma Belhadi, Juan P. Galeotti, Bogdan Marculescu, and Susruthan Seran. Emb: A curated corpus of web/enterprise applications and library support for software testing research. In *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 433–442, 2023.
- [12] Robert Baumgartner and Sicco Verwer. Learning state machines from data streams: A generic strategy and an improved heuristic. In François Coste, Faissal Ouadi, and Guillaume Rabusseau, editors, *Proceedings of 16th edition of the International Conference on Grammatical Inference*, volume 217 of *Proceedings of Machine Learning Research*, pages 117–141. PMLR, 10–13 Jul 2023.

- [13] Christian Birchler, Sajad Khatiri, Pouria Derakhshanfar, Sebastiano Panichella, and Annibale Panichella. Single and multi-objective test cases prioritization for self-driving cars in virtual environments. *ACM Transactions on Software Engineering and Methodology*, 32(2):1–30, 2023.
- [14] J. Black, E. Melachrinoudis, and D. Kaeli. Bi-criteria models for all-uses test suite reduction. In *Proceedings. 26th International Conference on Software Engineering*, pages 106–115, 2004.
- [15] Pietro Braione, Giovanni Denaro, Andrea Mattavelli, and Mauro Pezzè. Combining symbolic execution and search-based testing for programs with complex heap inputs. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2017, page 90–101, New York, NY, USA, 2017. Association for Computing Machinery.
- [16] Clinton Cao, Agathe Blaise, Annibale Panichella, and Sicco Verwer. State frequency estimation for anomaly detection. *arXiv preprint arXiv:2412.03442*, 2024.
- [17] Clinton Cao, Agathe Blaise, Sicco Verwer, and Filippo Rebecchi. Learning state machines to monitor and detect anomalies on a kubernetes cluster. In *Proceedings of the 17th International Conference on Availability, Reliability and Security*, ARES '22, New York, NY, USA, 2022. Association for Computing Machinery.
- [18] Clinton Cao, Annibale Panichella, and Sicco Verwer. Docker image for MISH. <https://figshare.com/s/fffa2873d4593db4e33f>, 2024. [Accessed 07-11-2024].
- [19] Clinton Cao, Annibale Panichella, and Sicco Verwer. MISH's github repository. <https://github.com/ClintonCao/EvoMaster>, 2024. [Accessed 27-1-2025].
- [20] Clinton Cao, Simon Schneider, Nicolas E. Diaz Ferreyra, Sicco Verwer, Annibale Panichella, and Riccardo Scandariato. Catma: Conformance analysis tool for microservice applications. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, ICSE-Companion '24, page 59–63, New York, NY, USA, 2024. Association for Computing Machinery.
- [21] W.J. Conover. *Practical Nonparametric Statistics*. Wiley, 1971.
- [22] Michael J Crawley. *Statistics: an introduction using R*. John Wiley & Sons, 2014.
- [23] Joeri de Ruiter and Erik Poll. Protocol state fuzzing of TLS implementations. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 193–206, Washington, D.C., August 2015. USENIX Association.
- [24] Duarte Felício, José Simão, and Nuno Datia. Rapitest: Continuous black-box testing of restful web apis. *Procedia Computer Science*, 219:537–545, 2023. CENTERIS –International Conference on ENTERprise Information Systems / ProjMAN –International Conference on Project MANagement / HCist –International Conference on Health and Social Care Information Systems and Technologies 2022.

- [25] Paul Fiterau-Brostean, Bengt Jonsson, Robert Merget, Joeri de Ruiter, Konstantinos Sagonas, and Juraj Somorovsky. Analysis of DTLs implementations using protocol state fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2523–2540. USENIX Association, August 2020.
- [26] Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ESEC/FSE '11*, page 416–419, New York, NY, USA, 2011. Association for Computing Machinery.
- [27] Amid Golmohammadi, Man Zhang, and Andrea Arcuri. On the impact of tool evolution and case study size on sbse experiments: A replicated study with evomaster. In Paolo Arcaini, Tao Yue, and Erik M. Fredericks, editors, *Search-Based Software Engineering*, pages 108–122, Cham, 2024. Springer Nature Switzerland.
- [28] Florian Gross, Gordon Fraser, and Andreas Zeller. Search-based system testing: high coverage, no false alarms. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis, ISSTA 2012*, page 67–77, New York, NY, USA, 2012. Association for Computing Machinery.
- [29] Gudmund Grov, Marc Sabate, Wei Chen, and David Aspinall. Towards intelligible robust anomaly detection by learning interpretable behavioural models. *NISK* 7, 32:1–16, 2019.
- [30] Pinjia He, Jieming Zhu, Zibin Zheng, and Michael R. Lyu. Drain: An online log parsing approach with fixed depth tree. In *2017 IEEE International Conference on Web Services (ICWS)*, pages 33–40, 2017.
- [31] Bogdan Korel. Automated software test data generation. *IEEE Transactions on software engineering*, 16(8):870–879, 1990.
- [32] Kevin J. Lang, Barak A. Pearlmutter, and Rodney A. Price. Results of the abbadingo one dfa learning competition and a new evidence-driven state merging algorithm. In Vasant Honavar and Giora Slutzki, editors, *Grammatical Inference*, pages 1–12, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.
- [33] Yuri Lavinas, Claus Aranha, Tetsuya Sakurai, and Marcelo Ladeira. Experimental analysis of the tournament size on genetic algorithms. In *2018 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 3647–3653. IEEE, 2018.
- [34] Valentina Lenarduzzi and Annibale Panichella. Serverless testing: Tool vendors’ and experts’ points of view. *IEEE Software*, 38(1):54–60, 2021.
- [35] Qin Lin, Sridha Adepu, Sicco Verwer, and Aditya Mathur. Tabor: A graphical model-based approach for anomaly detection in industrial control systems. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security, ASIACCS '18*, page 525–536, New York, NY, USA, 2018. Association for Computing Machinery.

- [36] LOGPAI. GitHub - logpai/Drain3: A robust streaming log template miner based on the Drain algorithm — github.com. <https://github.com/logpai/Drain3>. [Accessed 01-10-2024].
- [37] Bogdan Marculescu, Man Zhang, and Andrea Arcuri. On the faults found in rest apis by automated test generation. *ACM Trans. Softw. Eng. Methodol.*, 31(3), March 2022.
- [38] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. Restest: automated black-box testing of restful web apis. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2021*, page 682–685, New York, NY, USA, 2021. Association for Computing Machinery.
- [39] Petr Matoušek, Vojtěch Havlena, and Lukáš Holík. Efficient modelling of ics communication for anomaly detection using probabilistic automata. In *2021 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, pages 81–89. IEEE, 2021.
- [40] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. Reformulating branch coverage as a many-objective optimization problem. In *2015 IEEE 8th international conference on software testing, verification and validation (ICST)*, pages 1–10. IEEE, 2015.
- [41] Pedro Pinheiro, André Takeshi Endo, and Adenilso da Silva Simão. Model-based testing of restful web services using uml protocol state machines. 2013.
- [42] Dudekula Mohammad Rafi, Katam Reddy Kiran Moses, Kai Petersen, and Mika V. Mäntylä. Benefits and limitations of automated software testing: Systematic literature review and practitioner survey. In *2012 7th International Workshop on Automation of Software Test (AST)*, pages 36–42, 2012.
- [43] Miha Ravber, Shih-Hsi Liu, Marjan Mernik, and Matej Črepinšek. Maximum number of generations as a stopping criterion considered harmful. *Applied Soft Computing*, 128:109478, 2022.
- [44] Susruthan Seran, Man Zhang, and Andrea Arcuri. Search-based mock generation of external web service interactions. In Paolo Arcaini, Tao Yue, and Erik M. Fredericks, editors, *Search-Based Software Engineering*, pages 52–66, Cham, 2024. Springer Nature Switzerland.
- [45] Michael Sipser. *Introduction to the Theory of Computation*. Course Technology, Boston, MA, third edition, 2013.
- [46] Mozhan Soltani, Pouria Derakhshanfar, Annibale Panichella, Xavier Devroey, Andy Zaidman, and Arie van Deursen. Single-objective versus multi-objectivized optimization for evolutionary crash reproduction. In Thelma Elita Colanzi and Phil McMinn, editors, *Search-Based Software Engineering*, pages 325–340, Cham, 2018. Springer International Publishing.

- [47] Mozhan Soltani, Annibale Panichella, and Arie van Deursen. Search-based crash reproduction and its impact on debugging. *IEEE Transactions on Software Engineering*, 46(12):1294–1317, 2020.
- [48] Dimitri Stallenberg, Mitchell Olsthoorn, and Annibale Panichella. Improving test case generation for rest apis through hierarchical clustering. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 117–128, United States, 2021. IEEE. IEEE/ACM International Conference on Automated Software Engineering, ASE 2021 ; Conference date: 14-11-2021 Through 20-11-2021.
- [49] András Vargha and Harold D. Delaney. A critique and improvement of the "cl" common language effect size statistics of mcgraw and wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000.
- [50] Emanuele Viglianisi, Michael Dallago, and Mariano Ceccato. Resttestgen: Automated black-box testing of restful apis. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 142–152, 2020.
- [51] Muhammad Waseem, Peng Liang, Gastón Márquez, and Amleto Di Salle. Testing microservices architecture-based applications: A systematic mapping study. In *2020 27th Asia-Pacific Software Engineering Conference (APSEC)*, pages 119–128, 2020.
- [52] Muhammad Waseem, Peng Liang, Mojtaba Shahin, Amleto Di Salle, and Gastón Márquez. Design, monitoring, and testing of microservices systems: The practitioners' perspective. *Journal of Systems and Software*, 182:111061, 2021.
- [53] Shin Yoo and Mark Harman. Pareto efficient multi-objective test case selection. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis, ISSTA '07*, page 140–150, New York, NY, USA, 2007. Association for Computing Machinery.
- [54] Shin Yoo and Mark Harman. Regression testing minimization, selection and prioritization: a survey. *Software testing, verification and reliability*, 22(2):67–120, 2012.
- [55] Tao Yue, Shaukat Ali, and Lionel Briand. Automated transition from use cases to uml state machines to support state-based testing. In Robert B. France, Jochen M. Kuester, Behzad Bordbar, and Richard F. Paige, editors, *Modelling Foundations and Applications*, pages 115–131, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.
- [56] Dongliang Zhao, Jiaxing Guo, Chunxiang Gu, Yonghui Zheng, and Xieli Zhang. Agl-fuzz: Automata-guided fuzzing for detecting logic errors in security protocol implementations. *Computers & Security*, page 103979, 2024.
- [57] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding. Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study. *IEEE Transactions on Software Engineering*, 47(2):243–260, 2018.

Appendix B: Preliminary Results of MISHMOSA

This appendix presents the preliminary results of MISHMOSA, a method combining MISH and MOSA to guide an evolutionary algorithm in generating test cases for REST APIs of microservice applications. A more extensive study is needed to draw definitive conclusions about MISHMOSA's effectiveness.

6.8 Preliminary Analysis of MISHMOSA's Performance

Tables 6.9-6.11 present MISHMOSA's performance results on each respective metric used to evaluate MISH. From Table 6.9, we observe that MISHMOSA achieved a higher coverage on Catwatch and Features-Service, while retaining comparable coverage to the other search algorithms on the remaining applications. However, when directly compared to MOSA, we observe that MISH can enhance MOSA's performance on several applications, namely Catwatch, Features-Service, ProxyPrint, and OCVN. These results suggest that integrating MISH into MOSA's search strategy can enhance its performance, particularly when MOSA alone begins to stagnate. This observation can be further supported by MISHMOSA's convergence plots presented in Figures 6.11-6.16. We observe that MISHMOSA can discover more test targets at the start of the search. However, we also observe that MISHMOSA's performance stagnates in two applications (Scout-API and LanguageTool), ultimately resulting in a lower performance than MOSA on these applications. This may be attributed to the current implementation, where MISH serves as a fixed search objective within MOSA. Since MISHMOSA does not dynamically adapt its search strategy between the two approaches during a run, a smarter search heuristic could potentially yield better performance. Further research is needed to validate this hypothesis and determine how best to optimize MISHMOSA's strategy for sustained improvement.

Table 6.9: Median number of covered targets.

Application	MOSA		MISH - LM		MISH - WS		MISHMOSA - LM		MISHMOSA - WS	
	Median	IQR	Median	IQR	Median	IQR	Median	IQR	Median	IQR
CatWatch	2014.0	471.25	1703.5	51.25	1725.5	85.5	2072.0	94.25	2059.0	123.25
Features-Service	879.0	6.25	883.0	4.75	885.0	1.5	881.0	6.75	886.5	10.25
Scout-API	2223.0	68.25	2073.0	85.0	2089.5	84.5	2031.5	113.0	2078.0	56.5
ProxyPrint	2922.5	308.5	3302.0	127.75	3377.5	81.0	3109.5	170.5	3363.0	92.0
LanguageTool	3106.0	10262.5	2927.0	86.0	2948.5	87.75	2985.5	8610.75	2949.5	7792.0
OCVN	4464.5	170.0	6358.0	427.75	6083.5	230.75	6187.5	252.5	6104.5	149.5

Table 6.10: Median number of potential faults.

Application	MOSA		MISH - LM		MISH - WS		MISHMOSA - LM		MISHMOSA - WS	
	Median	IQR	Median	IQR	Median	IQR	Median	IQR	Median	IQR
CatWatch	25.0	1.25	24.0	2.0	24.0	1.0	19.0	4.0	20.0	3.25
Features-Service	32.0	0.0	31.0	0.5	31.5	1.0	31.0	1.0	31.0	2.0
Scout-API	79.0	9.5	66.0	2.0	69.0	4.25	64.5	3.75	67.5	3.5
ProxyPrint	66.0	10.25	68.0	3.0	69.0	2.25	68.0	2.5	69.0	2.25
LanguageTool	7.0	2.0	6.5	1.0	6.0	0.0	6.0	0.25	6.0	1.0
OCVN	274.5	7.0	311.5	13.0	303.0	10.5	314.5	9.5	314.5	11.25

Table 6.11: Median number of potential 500 faults.

Application	MOSA		MISH - LM		MISH - WS		MISHMOSA - LM		MISHMOSA - WS	
	Median	IQR	Median	IQR	Median	IQR	Median	IQR	Median	IQR
CatWatch	10.0	1.25	10.0	2.0	10.0	1.25	7.0	1.25	7.0	2.0
Features-Service	25.0	0.0	25.0	1.0	25.0	1.0	25.0	1.0	25.0	2.0
Scout-API	59.5	9.0	47.0	2.25	50.0	4.25	45.0	3.75	48.5	4.25
ProxyPrint	32.0	5.25	33.0	2.0	34.0	2.0	33.0	2.25	34.0	1.25
LanguageTool	5.0	1.0	4.5	1.0	4.0	0.0	4.0	0.25	4.0	1.0
OCVN	49.0	4.5	86.5	9.75	79.5	10.75	85.5	9.0	88.0	9.75

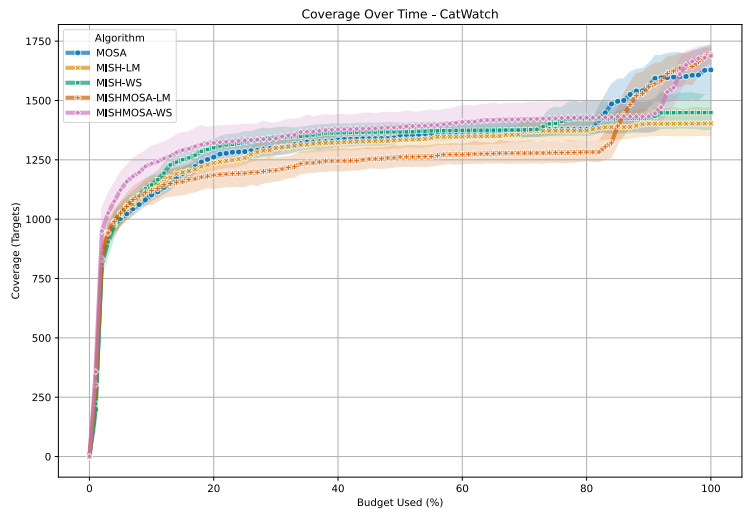


Figure 6.11: Average coverage over time on CatWatch.

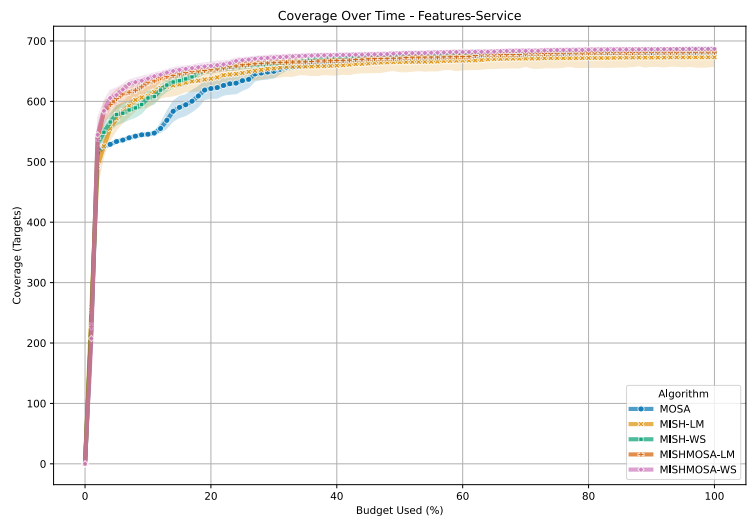


Figure 6.12: Average coverage over time on Features-Service.

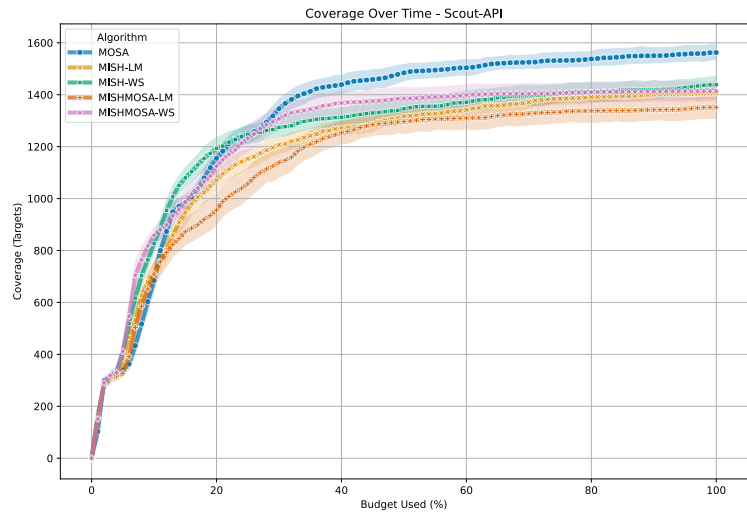


Figure 6.13: Average coverage over time on Scout-API.

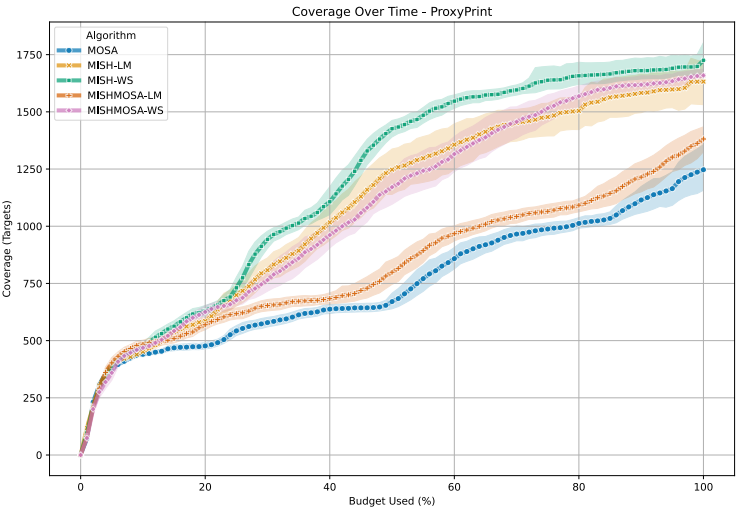


Figure 6.14: Average coverage over time on ProxyPrint.

6

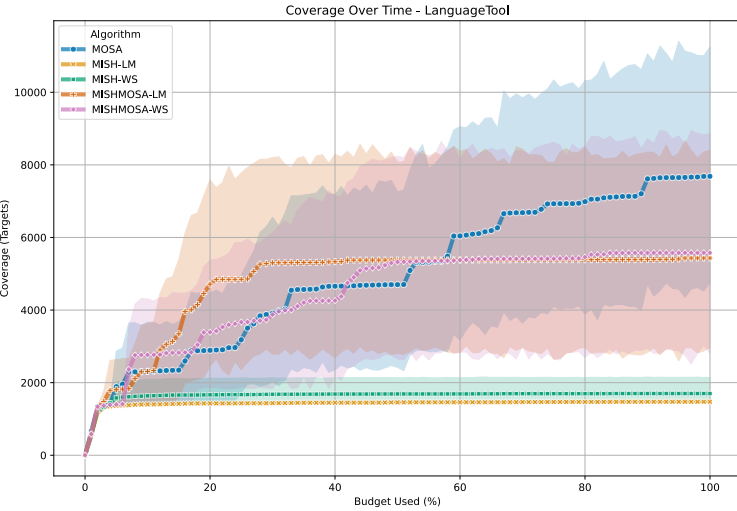


Figure 6.15: Average coverage over time on LanguageTool.

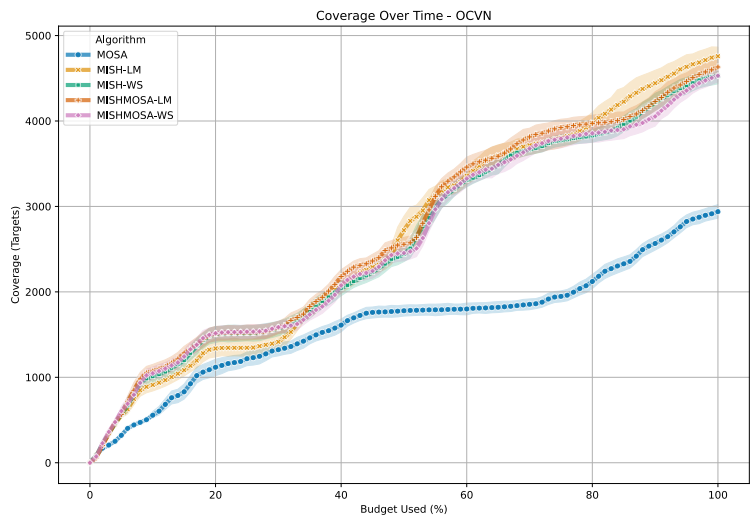


Figure 6.16: Average coverage over time on OCVN.

7

Conclusion

This chapter concludes the research presented in this dissertation. Each research question is revisited along with the answers derived from the corresponding study (or studies). Additionally, potential future works is sketched out for each research question, serving as potential starting points for researchers interested in further applications of state machines on microservices.

7.1 Research Questions Revisited

RQ1: What outputs of microservices enable the learning of accurate state machines?

The studies presented in this dissertation used various forms of outputs collected from microservices to learn state machines.

Part I

Part I of this dissertation explored using NetFlows to learn state machines for microservices. These flows consisted of summarized statistics collected from network connections within a Kubernetes cluster.

- Although essential details or patterns may be lost in the summarized statistics, the experimental results of the corresponding chapters demonstrated that accurate state machine models can be inferred using only a small subset of the NetFlow features.
- These models demonstrated to be highly effective in unsupervised network anomaly detection (e.g., detecting network attacks), outperforming several existing unsupervised ML approaches.
- This indicates that NetFlows collected from microservices are sufficient to learn an accurate state machine model for network anomaly detection.

However, an important consideration must be noted:

- The flows were preprocessed using the ENCODE algorithm (see the answer of RQ2) before they were used to infer a state machine model.
- Therefore, one can learn accurate state machine models from NetFlows, provided that they are first processed with our novel ENCODE algorithm.

Part II

Part II explored a variant of network output, namely HTTP request logs. Unlike NetFlows, these logs also included details on the API Endpoint URL used in the connections and the parameters information

- For the study conducted in Part II, the logs were not preprocessed by the ENCODE algorithm as it currently does not support textual features.
- Instead, the proposed tool CATMA, directly utilizes these textual features to generate explanations for its users.

Despite the lack of preprocessing:

- The experimental results demonstrated that an effective model was learned from the logs.
- In fact, a real-life bug was detected in a microservice application using the inferred state machine.
- The original developer confirmed this bug. A fix was issued using the explanations generated by CATMA, and the original developer accepted it.

This highlights that:

- An alternative output collected from the microservices can also be effectively used to learn a state machine model.

Part III

The study conducted in Part III of this dissertation explored using an alternative output source for learning state machine models: software logs statements inserted in the code by developers.

- Similar to the study conducted in Part II, the log statements are not processed with the ENCODE algorithm due to its previously mentioned limitation on textual features.
- Instead, a different encoding algorithm was applied to process the log statements. This is to reduce the combinatorial state explosion caused by a diverse variety of log statements.
- Experiment results demonstrated that the learned models successfully captured essential behavioral patterns, making it possible to generate more effective system-level tests.

Thus, various outputs collected from microservices can be used to learn an accurate and effective state machine. Whether a specialized encoding algorithm should process these outputs depends on the use case:

- When the state machine is intended as a symbolic representation of system behavior, and the discrete events are not being used for a specific purpose (e.g., as how the discrete events were used in Part II), it is recommended to preprocess the input data before it is used for learning a state machine.
- When the state machine is used to aid the understanding of a system's inner working, it is best to not discretize the event using an encoding algorithm.

RQ2: How can outputs of microservices be transformed into discrete events suitable for learning state machines?

This dissertation explored two main approaches for transforming outputs collected from microservices into discrete events suitable for learning state machines.

Discretizing Events with ENCODE

Part I explored using the ENCODE algorithm to process NetFlows collected from microservices to learn state machines.

- Experiment results demonstrated that ENCODE enhances the performance of state machines for detecting network anomalies.
- Results also demonstrated that ENCODE enhances the performance of several unsupervised ML approaches, primarily due to the contextual information captured in the frequencies of the feature values.

Discretizing Events with Drain3

Part III explored using the Drain3 algorithm from LogPai [4] to transform software log statements into discrete events suitable for learning state machines.

- Drain3 was chosen to address two limitations of ENCODE: (1) ENCODE was not designed for textual features, and (2) ENCODE does not support online learning of the encoding like Drain3 does.
- Experimental results demonstrated that state machines learned through this approach can generate more effective system-level test cases for microservices.

RQ3: How can state machines be applied to model behavioral patterns for anomaly detection in microservices?

Two main approaches were studied in **Part I** regarding using state machines to detect anomalies in microservices. These approaches focused on detecting network anomalies (e.g., network attacks) in an unsupervised manner.

Learning Probabilistic State Machines For Anomaly Detection

The first approach builds upon a well-established anomaly detection method employed by various prior studies involving state machines. In this approach, a model is first trained on benign data collected from the target system or environment. The model is then employed for anomaly detection by assessing how well the test data fit within the learned model. An alert is raised if the test data deviate considerably from the training observations.

Studies conducted in **Chapters 2 and 3** applied this particular approach to detect network anomalies in microservice applications deployed on a Kubernetes cluster. Given the lack of representative runtime monitoring datasets for microservices at the time of the study, a new dataset was created. This dataset features realistic network traffic data, including malicious NetFlows from attacks launched against the microservices and benign NetFlows generated by users interacting with the deployed microservices. Traces derived from these NetFlows were used to train state machine models. Anomalies were then detected by evaluating how well the test traces fit the trained models.

Experimental results demonstrated that these learned models effectively detected network anomalies within Kubernetes-deployed microservices. Furthermore, visualizations showed a clear distinction between benign and malicious data, highlighting the efficacy of the state-machine approach.

Utilizing State Frequency For Anomaly Detection

The second anomaly detection approach is a novel method designed to address two limitations of the first approach: (1) certain benign traces might occur in low frequency, leading to an increase in false alarms, and (2) malicious traces tend to resemble benign traces to avoid detection by the anomaly detection system. These issues pose challenges for traditional approaches that rely solely on how well test data matches training data.

The novel approach, SEQUENT, introduced in **Chapter 4** addresses these two issues by leveraging a state machine's key feature: the state machine's behavioral structure. Specifically, traces replayed on a state machine are used to derive behaviors exhibited by the system for the corresponding traces. Each trace traverses a path within the state machine, with each state corresponding to a distinct type of behavior exhibited by the system.

The sequence of states within the path and their associated visit frequencies can be used to determine whether a particular state is exhibiting anomalous behavior. SEQUENT does this by employing the following intuition: If a particular behavior (represented by a state) occurs considerably more frequently than observed during training, this may indicate anomalous behavior. To achieve this, SEQUENT compares the visit frequencies of the states visited by a test trace to their corresponding expected frequencies observed during training. SEQUENT then computes an anomaly score for the test trace based on the comparison result.

SEQUENT employs a novel scoring method that dynamically adapts its output at test time. Rather than relying solely on static thresholds, SEQUENT incrementally adjusts the scores for each state based on the behaviors observed at test time, making it more flexible to handle sudden changes in the frequency of the behavioral patterns.

Evaluation results of SEQUENT show that state machines perform considerably better in anomaly detection when using SEQUENT's novel scoring approach to determine anomaly scores for arbitrary traces. This approach offers a promising future research direction for state machines or other ML models.

RQ4: How can state machines be utilized to generate root causes or explanations for faults or anomalies detected in microservices?

Providing root causes or explanations to developers or system analysts can greatly help them understand why an unexpected fault or anomalous behavior occurred within the system. With studies showing that developers often spend days identifying the root cause of a fault in microservice applications [3, 5], it has become increasingly important to develop tools that can help developers streamline this process. These tools should not only detect faults or anomalies but also enable users to prioritize the alerts raised by the tools. Without such prioritization, developers or system analysts may encounter the so-called “alert fatigue”, reducing the efficiency. This dissertation explored two approaches to using state machines to generate root causes or explanations for faults or anomalies detected in microservices. The first approach is introduced in **Chapter 4**, while the second approach is introduced in **Chapter 5**.

Using State Frequencies to Generate Root Causes

In the first approach, SEQUENT employs its novel scoring method to determine the root cause of an anomaly for an arbitrary test trace. Specifically, it observes the path traversed by the trace within the state machine and identifies the state with the largest anomaly score. This state is the root cause of an anomaly. It can be used to create groups of anomalies exhibiting a similar behavior (i.e., showing large anomaly scores at the same state). SEQUENT also ranks these groups of anomalies based on their sizes. Anomalies within the groups are also ranked based on their anomaly scores. Additionally, SEQUENT links anomalies back to their corresponding data instances, enabling users (e.g., analysts) to investigate and understand the underlying cause of the anomalies quickly. Experimental results of SEQUENT show that analysts can efficiently determine whether a group of anomalies represents a false alarm by analyzing the most anomalous instances of each group.

Comparison with Dataflow Diagrams to Find Discrepancies

In the second approach, CATMA generates human-readable explanations that help developers quickly identify the potential causes of discrepancies detected between behaviors specified in the logic of the code and the behaviors observed at runtime. Specifically, these explanations are derived by comparing the behavioral patterns captured in the state machine against a Dataflow Diagram (DFD) that models the intended behaviors defined in the code logic. CATMA identifies differences between the models and extracts additional insights to assist developers in understanding the potential root causes behind the detected differences. As the state machine captures behavioral patterns exhibited during runtime, it can be a stack trace for the detected differences. This enables developers to gather the sequence of HTTP requests and their parameter values and replay the behavior causing the corresponding detected difference.

RQ5: How can state machines be utilized to detect non-conformances between code and deployment of microservices?

Chapter 5 defines “non-conformances” as discrepancies identified between the behaviors observed at runtime and the expected behaviors specified in the logic of the source code,

i.e., runtime behavior that does not conform to the intended behavior implemented in the source code. There are various reasons why such non-conformances are detected within microservice applications. Misconfiguration of services is one of the most common root causes of the faults encountered in microservice applications.

CATMA

The study conducted in **Chapter 5** led to the development of CATMA, a lightweight tool for detecting and analyzing non-conformances in microservices applications.

- CATMA aims to pinpoint discrepancies between the implemented code and its actual behavior while providing helpful interpretations of the identified discrepancies.
- Discrepancies are detected by comparing the events captured within the state machine model against the expected events written in the source of the microservices. Any event that occurs at runtime but is not specified in the source code (or vice versa) is flagged as a non-conformance.
- CATMA offers potential interpretations that can provide various insights to help the developer quickly narrow down the potential starting points for debugging a detected non-conformance. These insights are derived from the behavioral patterns captured in the state machine model. Specifically, the state machine models the HTTP request sequence that leads to a particular behavior in the microservice application (each state represents a distinct type of behavior).
- When a non-conformance is detected, information such as HTTP request sequences are derived from the state machine and presented to the user as stack traces (together with the used parameter values) for the non-conformance. The user can essentially replay these sequences to reproduce the detected non-conformance.

7

RQ6: How can state machines be utilized to generate test cases for microservices automatically?

The final part of this dissertation (**Part III**) focused on utilizing state machines for the automated generation of test cases targeting the REST APIs of microservice applications.

Challenges with Testing Microservice Applications:

- Microservices communicate with each other via REST APIs, and a request made by a client may involve a sequence of REST API calls to obtain the desired resource.
- Due to the distributed nature of microservices, manually crafting system-level test cases for the REST APIs is labor-intensive and prone to incomplete coverage of the whole system.

Prior studies have investigated using EAs to generate test cases for REST APIs of microservices automatically. However, the drawback of these approaches are:

- Existing approaches often rely heavily on unit-level heuristics, such as branch distances, to guide the generation of test cases.

- These heuristics are effective at maximizing coverage for individual units or services, but they often fall short of capturing intricate system-level behaviors.
- As a result, the generated test cases may overlook faults that only surface during complex interservice interactions.

Part III introduces a novel search heuristic, MISH, that leverages state machines to guide an EA in the automated generation of test cases for REST APIs of microservices:

- MISH continuously learns a state machine by observing complex interservice interactions exhibited by a microservice application and employs the model to define fitness for the generated test cases.
- MISH assigns higher fitness values to test cases that provoke new behaviors within the microservice application since the aim is to discover faults that arise due to complex interservice interactions.
- Evaluation results of MISH show that MISH can outperform a state-of-the-art search heuristic, MOSA, in generating system-level test cases for microservice applications with complex interservice interactions.
- Additionally, the study provided promising results, suggesting that employing MISH as part of the MOSA algorithm may yield more effective system-level test cases for REST APIs of microservices.

Overall Conclusion of Dissertation

Overall, this dissertation has shown that state machines are powerful and accurate models for modeling sequential runtime behaviors of microservice applications:

- State machines, with the help of ENCODE, can effectively detect network anomalies within microservice applications, outperforming existing unsupervised anomaly detection methods.
- State machines capture essential behavioral information that can help developers or analysts understand the root causes of detected failure within their microservice applications.
- State machines can be learned online and used to derive essential sequential behaviors for generating effective test cases for microservice applications.

State machines are lightweight models and do not require special hardware for training. Although these models are not the newest in computer science, their applications in modeling sequential runtime behaviors of the latest system architectures are worth investigating.

7.2 Limitations & Potential Future Works

RQ1: What outputs of microservices enable the learning of accurate state machines?

While **Part I** of this dissertation has focused on utilizing outputs collected from microservices to learn a state machine model, the emphasis was predominantly on NetFlow data. NetFlow data consists of summarized statistics collected from the network connections within the microservice application. Although the summarized statistics were shown to be sufficient to learn an effective state machine model, this dissertation did not explore the use of more granular network data, such as packet data. Learning a state machine model from this output type may lead to a more accurate model due to its richer details. However, it is not trivial how packet data should be processed to learn a state machine model. Recall that state machines model behavioral patterns captured in discrete events. Having an excessive number of distinct events may lead to the combinatorial explosion of states during learning, thereby slowing down the learning process. Given the detailed nature of packet data, careful preprocessing and transformation would be essential to effectively and efficiently learn a state machine model.

Another type of microservice output data that may produce more accurate state machine models is trace data generated by tools such as DynaTrace or Jaeger. These tools enable users to trace the complex interactions within the distributed nature of microservices and represent them as a sequence of API requests. This kind of trace data essentially models the sequential behavior exhibited by the microservice application, providing valuable insights to developers or analysts when debugging faults or failures. This dissertation explored the approach of passively collecting outputs from microservices and learning a state machine from the collected data without taking the distributed nature of the microservices into account. As a result, the traces (traces of events) extracted from the collected output may not accurately represent the actual traces occurring within the system since collected events may have occurred asynchronously. Trace data generated by DynaTrace [1] or Jaeger [2] may capture sequential behaviors of microservices more accurately, potentially leading to more accurate state machine models.

RQ2: How can outputs of microservices be transformed into discrete events suitable for learning state machines?

One key contribution of this dissertation is ENCODE, an encoding algorithm to process NetFlows for learning state machines. Currently, ENCODE handles numerical features but does not support textual features. An interesting future work for ENCODE is extending its capabilities to handle textual features, enabling ENCODE to process software log data. This extension also enables us to compare ENCODE and the Drain3 algorithm for processing software logs for learning state machines. This comparison allows us to determine which symbolic representation best supports the learning state machine from software log data. Additionally, ENCODE learns a fixed mapping between feature values and their corresponding encoded counterparts (cluster symbols) once from the provided training data. As a result, the clusters remain static and cannot be adapted to new data over time. Another interesting future work for ENCODE is to develop its support for the online learning of the encoded values. This enables the ability to reform the clusters when

needed and enables ENCODE to deal with unseen data observed at test time. Currently, ENCODE tries to find the “closest” feature value observed during training when unseen data is provided as input. The ability to reform the clusters may improve the accuracy of the resulting encodings.

As mentioned in the future works of **RQ1**, trace data generated by DynaTrace or Jaeger may lead to more accurate state machines learned for microservices. It is not trivial how these trace data should be transformed into discrete events suitable for learning state machines. An interesting future work is to investigate this matter to learn more accurate state machines for capturing behavioral patterns of microservices.

RQ3: How can state machines be applied to model behavioral patterns for anomaly detection in microservices?

Chapter 4 introduced SEQUENT, a novel approach that utilizes state machines to compute anomaly scores for arbitrary traces observed at test time. SEQUENT’s approach of using state visit frequency has demonstrated a better performance in detecting network anomalies than the traditional approach of aligning a trace within the learned model to determine its anomaly score. The study conducted in this dissertation only investigated the application of SEQUENT for network anomalies. An interesting future work of SEQUENT would be assessing this approach for detecting anomalies captured in software log data. This enables us to assess whether SEQUENT’s approach can be applied to anomalies beyond the network domain.

Additionally, SEQUENT provides an interesting research direction for applying its strategy within other types of ML models. For instance, it is not trivial how SEQUENT’s approach can be applied to models such as NNs to determine anomaly scores for arbitrary (test) data points. Employing a similar approach within NNs may create a new alternative to define anomaly scores for data points. Furthermore, it enables us to assess whether SEQUENT’s approach provides performance benefits to a broader range of ML models.

RQ4: How can state machines be utilized to generate root causes or explanations for faults or anomalies detected in microservices?

This dissertation explored two approaches for generating explanations or root causes for faults or anomalies using a state machine. **Chapter 5** introduced CATMA, a lightweight tool for conducting conformance analysis on microservice applications, and generates potential root causes to explain the detected non-conformances. However, CATMA’s evaluation was limited to a small pilot study, which may not have provided a representative set of feedback for meaningful improvements. Therefore, CATMA would benefit from an extensive study involving various developers of diverse skill levels to assess its usefulness and gather detailed feedback for refinement. Additionally, CATMA only provides a fixed list of potential interpretations generated for the detected non-conformances. However, it is possible to develop smarter techniques for selecting more suitable explanations for the developer by filtering them using information (as evidence) captured within the state machine and the DFD model.

Chapter 4 introduced SEQUENT, a novel anomaly detection approach that uses state visit frequency to determine anomaly scores. The anomaly scoring method allows analysts to group and rank anomalies, helping them quickly filter out many alerts if a group

of anomalies is deemed false alarms. However, the study only investigated one approach that uses the group size to rank anomalies. An interesting future work for SEQUENT would be to develop other approaches to rank groups of anomalies and conduct an extensive study with analysts to assess the efficacy of the approaches. This study would help determine which ranking strategies best aid analysts in improving their workflow of analyzing anomalies. The study's results can also help extend the set of techniques supported by SEQUENT, creating a more extensive framework for conducting anomaly detection and analysis using state machines.

RQ5: How can state machines be utilized to detect non-conformances between code and deployment of microservices?

Part II of this dissertation explored comparing state machines and the DFD model to detect non-conformances within microservice applications. In this approach, non-conformances are detected by comparing edges within the DFD model against the transitions within the state machine. While CATMA's study focused solely on these two models, the tool was designed to be extendable for other model comparisons. An interesting future work is to explore further comparisons between state machines and alternative models derived from source code logic (e.g., control flow graphs or program dependency graphs). This comparison enables us to evaluate which model combinations are most effective for detecting non-conformances and generating potential interpretations as root causes for the corresponding detected non-conformances.

RQ6: How can state machines be utilized to generate test cases for microservices automatically?

The final part of this dissertation explored how state machines can guide an EA in the automated generation of test cases for REST APIs in microservice applications. The proposed search heuristic, MISH, demonstrated promising results by incorporating model inference as a search heuristic for an EA. However, the study primarily focused on two specific fitness functions to evaluate test cases based on a state machine learned from software log statements. An interesting future work for MISH is to investigate other fitness functions and compare these to those used in the current study. This enables us to determine which information extracted from the state machine works better in generating more effective system-level test cases for REST APIs of microservices. Furthermore, the preliminary findings of MISHMOSA (a combination of MISH and MOSA) indicate that using MISH as part of a multi-objective algorithm may enhance its performance in generating more effective system-level test cases. A more comprehensive evaluation of MISHMOSA is necessary to confirm these initial results.

References

- [1] Dynatrace. Dynatrace: Understand your business like never before. <https://www.dynatrace.com/>. [Accessed 22-6-2025].
- [2] Jaeger. Jaeger: open source, distributed tracing platform. <https://www.jaegertracing.io/>. [Accessed 22-6-2025].
- [3] Valentina Lenarduzzi and Annibale Panichella. Serverless testing: Tool vendors' and experts' points of view. *IEEE Software*, 38(1):54–60, 2021.
- [4] LOGPAI. GitHub - logpai/Drain3: A robust streaming log template miner based on the Drain algorithm — github.com. <https://github.com/logpai/Drain3>. [Accessed 01-10-2024].
- [5] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding. Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study. *IEEE Transactions on Software Engineering*, 47(2):243–260, 2018.

Curriculum Vitæ

Clinton Shen Cao

October 26, 1994	Born in Oranjestad, Aruba
2007 - 2013	VWO Diploma Colegio Arubano
2014 - 2017	BSc. Computer Science Delft University of Technology Information & Knowledge Engineering Track Minor Robotics at Delft University of Technology Thesis at @Large Research Group
April 2017 - June 2017	Project Intern - Software Engineering ING Nederland, Amsterdam
2017 - 2020	MSc Computer Science Delft University of Technology Data Science & Technology Track Specialization in Cybersecurity Thesis at APTA Technologies
2019 - 2020	Research Intern APTA Technologies
2020 - 2024	PhD in Computer Science Delft University of Technology Participated in the EU H2020 AssureMOSS project Cyber Security & Algorithmics research groups
2017 - 2024	Teaching Assistant Delft University of Technology
2024 - present	Data Scientist / A.I. Engineer Nationale Nederlanden

List of Publications

1. **Cao, C.**, Panichella, A., & Verwer, S. E.. (2025). Automated Test-Case Generation for REST APIs Using Model Inference Search Heuristic. In The 6th ACM/IEEE International Conference on Automation of Software Test IEEE / ACM.
2. **Cao, C.**, Schneider, S., Ferreyra, N. E. D., Verwer, S., Panichella, A., & Scandariato, R. (2024). CATMA: Conformance analysis tool for microservice applications. In Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings (pp. 59-63).
3. Nadeem, A., Vos, D., **Cao, C.**, Pajola, L., Dieck, S., Baumgartner, R., & Verwer, S. (2023). Sok: Explainable machine learning for computer security applications. In 2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P) (pp. 221-240). IEEE.
4. **Cao, C.**, Blaise, A., Verwer, S., & Rebecchi, F. (2022, August). Learning state machines to monitor and detect anomalies on a kubernetes cluster. In Proceedings of the 17th International Conference on Availability, Reliability and Security (pp. 1-9).
5. Buckers, T., **Cao, C.**, Doesburg, M., Gong, B., Wang, S., Beller, M., & Zaidman, A. (2017). UAV: Warnings from multiple automated static analysis tools at a glance. In 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER) (pp. 472-476). IEEE.

Preprints

6. **Cao, C.**, Blaise, A., Panichella, A., & Verwer, S. (2024). State Frequency Estimation for Anomaly Detection.
 7. **Cao, C.**, Panichella, A., Verwer, S., Blaise, A., & Rebecchi, F. (2022). ENCODE: Encoding NetFlows for Network Anomaly Detection.
- Included in this dissertation.
- Won a best paper, tool demonstration, poster, or proposal award.

Titles in the SIKS Dissertation Series Since 2016

- 2016 01 Syed Saiden Abbas (RUN), Recognition of Shapes by Humans and Machines
- 02 Michiel Christiaan Meulendijk (UU), Optimizing medication reviews through decision support: prescribing a better pill to swallow
- 03 Maya Sappelli (RUN), Knowledge Work in Context: User Centered Knowledge Worker Support
- 04 Laurens Rietveld (VUA), Publishing and Consuming Linked Data
- 05 Evgeny Sherkhonov (UvA), Expanded Acyclic Queries: Containment and an Application in Explaining Missing Answers
- 06 Michel Wilson (TUD), Robust scheduling in an uncertain environment
- 07 Jeroen de Man (VUA), Measuring and modeling negative emotions for virtual training
- 08 Matje van de Camp (TiU), A Link to the Past: Constructing Historical Social Networks from Unstructured Data
- 09 Archana Nottamkandath (VUA), Trusting Crowdsourced Information on Cultural Artefacts
- 10 George Karafotias (VUA), Parameter Control for Evolutionary Algorithms
- 11 Anne Schuth (UvA), Search Engines that Learn from Their Users
- 12 Max Knobbout (UU), Logics for Modelling and Verifying Normative Multi-Agent Systems
- 13 Nana Baah Gyan (VUA), The Web, Speech Technologies and Rural Development in West Africa - An ICT4D Approach
- 14 Ravi Khadka (UU), Revisiting Legacy Software System Modernization
- 15 Steffen Michels (RUN), Hybrid Probabilistic Logics - Theoretical Aspects, Algorithms and Experiments
- 16 Guangliang Li (UvA), Socially Intelligent Autonomous Agents that Learn from Human Reward
- 17 Berend Weel (VUA), Towards Embodied Evolution of Robot Organisms
- 18 Albert Meroño Peñuela (VUA), Refining Statistical Data on the Web
- 19 Julia Efremova (TU/e), Mining Social Structures from Genealogical Data
- 20 Daan Odijk (UvA), Context & Semantics in News & Web Search
- 21 Alejandro Moreno Céleri (UT), From Traditional to Interactive Playspaces: Automatic Analysis of Player Behavior in the Interactive Tag Playground
- 22 Grace Lewis (VUA), Software Architecture Strategies for Cyber-Foraging Systems
- 23 Fei Cai (UvA), Query Auto Completion in Information Retrieval

- 24 Brend Wanders (UT), Repurposing and Probabilistic Integration of Data; An Iterative and data model independent approach
- 25 Julia Kiseleva (TU/e), Using Contextual Information to Understand Searching and Browsing Behavior
- 26 Dilhan Thilakarathne (VUA), In or Out of Control: Exploring Computational Models to Study the Role of Human Awareness and Control in Behavioural Choices, with Applications in Aviation and Energy Management Domains
- 27 Wen Li (TUD), Understanding Geo-spatial Information on Social Media
- 28 Mingxin Zhang (TUD), Large-scale Agent-based Social Simulation - A study on epidemic prediction and control
- 29 Nicolas Höning (TUD), Peak reduction in decentralised electricity systems - Markets and prices for flexible planning
- 30 Ruud Mattheij (TiU), The Eyes Have It
- 31 Mohammad Khelghati (UT), Deep web content monitoring
- 32 Eelco Vriezekolk (UT), Assessing Telecommunication Service Availability Risks for Crisis Organisations
- 33 Peter Bloem (UvA), Single Sample Statistics, exercises in learning from just one example
- 34 Dennis Schunselaar (TU/e), Configurable Process Trees: Elicitation, Analysis, and Enactment
- 35 Zhaochun Ren (UvA), Monitoring Social Media: Summarization, Classification and Recommendation
- 36 Daphne Karreman (UT), Beyond R2D2: The design of nonverbal interaction behavior optimized for robot-specific morphologies
- 37 Giovanni Sileno (UvA), Aligning Law and Action - a conceptual and computational inquiry
- 38 Andrea Minuto (UT), Materials that Matter - Smart Materials meet Art & Interaction Design
- 39 Merijn Bruijnes (UT), Believable Suspect Agents; Response and Interpersonal Style Selection for an Artificial Suspect
- 40 Christian Detweiler (TUD), Accounting for Values in Design
- 41 Thomas King (TUD), Governing Governance: A Formal Framework for Analysing Institutional Design and Enactment Governance
- 42 Spyros Martzoukos (UvA), Combinatorial and Compositional Aspects of Bilingual Aligned Corpora
- 43 Saskia Koldijk (RUN), Context-Aware Support for Stress Self-Management: From Theory to Practice
- 44 Thibault Sellam (UvA), Automatic Assistants for Database Exploration
- 45 Bram van de Laar (UT), Experiencing Brain-Computer Interface Control
- 46 Jorge Gallego Perez (UT), Robots to Make you Happy
- 47 Christina Weber (UL), Real-time foresight - Preparedness for dynamic innovation networks
- 48 Tanja Buttler (TUD), Collecting Lessons Learned
- 49 Gleb Polevoy (TUD), Participation and Interaction in Projects. A Game-Theoretic Analysis

-
- 50 Yan Wang (TiU), The Bridge of Dreams: Towards a Method for Operational Performance Alignment in IT-enabled Service Supply Chains
-
- 2017 01 Jan-Jaap Oerlemans (UL), Investigating Cybercrime
 02 Sjoerd Timmer (UU), Designing and Understanding Forensic Bayesian Networks using Argumentation
 03 Daniël Harold Telgen (UU), Grid Manufacturing; A Cyber-Physical Approach with Autonomous Products and Reconfigurable Manufacturing Machines
 04 Mrunal Gawade (CWI), Multi-core Parallelism in a Column-store
 05 Mahdiah Shadi (UvA), Collaboration Behavior
 06 Damir Vandic (EUR), Intelligent Information Systems for Web Product Search
 07 Roel Bertens (UU), Insight in Information: from Abstract to Anomaly
 08 Rob Konijn (VUA), Detecting Interesting Differences: Data Mining in Health Insurance Data using Outlier Detection and Subgroup Discovery
 09 Dong Nguyen (UT), Text as Social and Cultural Data: A Computational Perspective on Variation in Text
 10 Robby van Delden (UT), (Steering) Interactive Play Behavior
 11 Florian Kunneman (RUN), Modelling patterns of time and emotion in Twitter #anticipointment
 12 Sander Leemans (TU/e), Robust Process Mining with Guarantees
 13 Gijs Huisman (UT), Social Touch Technology - Extending the reach of social touch through haptic technology
 14 Shoshannah Tekofsky (TiU), You Are Who You Play You Are: Modelling Player Traits from Video Game Behavior
 15 Peter Berck (RUN), Memory-Based Text Correction
 16 Aleksandr Chuklin (UvA), Understanding and Modeling Users of Modern Search Engines
 17 Daniel Dimov (UL), Crowdsourced Online Dispute Resolution
 18 Ridho Reinanda (UvA), Entity Associations for Search
 19 Jeroen Vuurens (UT), Proximity of Terms, Texts and Semantic Vectors in Information Retrieval
 20 Mohammadbashir Sedighi (TUD), Fostering Engagement in Knowledge Sharing: The Role of Perceived Benefits, Costs and Visibility
 21 Jeroen Linssen (UT), Meta Matters in Interactive Storytelling and Serious Gaming (A Play on Worlds)
 22 Sara Magliacane (VUA), Logics for causal inference under uncertainty
 23 David Graus (UvA), Entities of Interest — Discovery in Digital Traces
 24 Chang Wang (TUD), Use of Affordances for Efficient Robot Learning
 25 Veruska Zamborlini (VUA), Knowledge Representation for Clinical Guidelines, with applications to Multimorbidity Analysis and Literature Search
 26 Merel Jung (UT), Socially intelligent robots that understand and respond to human touch
 27 Michiel Jooze (UT), Investigating Positioning and Gaze Behaviors of Social Robots: People's Preferences, Perceptions and Behaviors
 28 John Klein (VUA), Architecture Practices for Complex Contexts

-
- 29 Adel Alhuraibi (TiU), From IT-BusinessStrategic Alignment to Performance: A Moderated Mediation Model of Social Innovation, and Enterprise Governance of IT"
 - 30 Wilma Latuny (TiU), The Power of Facial Expressions
 - 31 Ben Ruijl (UL), Advances in computational methods for QFT calculations
 - 32 Thaer Samar (RUN), Access to and Retrieval of Content in Web Archives
 - 33 Brigit van Loggem (OU), Towards a Design Rationale for Software Documentation: A Model of Computer-Mediated Activity
 - 34 Maren Scheffel (OU), The Evaluation Framework for Learning Analytics
 - 35 Martine de Vos (VUA), Interpreting natural science spreadsheets
 - 36 Yuanhao Guo (UL), Shape Analysis for Phenotype Characterisation from High-throughput Imaging
 - 37 Alejandro Montes Garcia (TU/e), WiBAF: A Within Browser Adaptation Framework that Enables Control over Privacy
 - 38 Alex Kayal (TUD), Normative Social Applications
 - 39 Sara Ahmadi (RUN), Exploiting properties of the human auditory system and compressive sensing methods to increase noise robustness in ASR
 - 40 Altaf Hussain Abro (VUA), Steer your Mind: Computational Exploration of Human Control in Relation to Emotions, Desires and Social Support For applications in human-aware support systems
 - 41 Adnan Manzoor (VUA), Minding a Healthy Lifestyle: An Exploration of Mental Processes and a Smart Environment to Provide Support for a Healthy Lifestyle
 - 42 Elena Sokolova (RUN), Causal discovery from mixed and missing data with applications on ADHD datasets
 - 43 Maaïke de Boer (RUN), Semantic Mapping in Video Retrieval
 - 44 Garm Lucassen (UU), Understanding User Stories - Computational Linguistics in Agile Requirements Engineering
 - 45 Bas Testerink (UU), Decentralized Runtime Norm Enforcement
 - 46 Jan Schneider (OU), Sensor-based Learning Support
 - 47 Jie Yang (TUD), Crowd Knowledge Creation Acceleration
 - 48 Angel Suarez (OU), Collaborative inquiry-based learning
-
- 2018 01 Han van der Aa (VUA), Comparing and Aligning Process Representations
 - 02 Felix Mannhardt (TU/e), Multi-perspective Process Mining
 - 03 Steven Bosems (UT), Causal Models For Well-Being: Knowledge Modeling, Model-Driven Development of Context-Aware Applications, and Behavior Prediction
 - 04 Jordan Janeiro (TUD), Flexible Coordination Support for Diagnosis Teams in Data-Centric Engineering Tasks
 - 05 Hugo Huurdeman (UvA), Supporting the Complex Dynamics of the Information Seeking Process
 - 06 Dan Ionita (UT), Model-Driven Information Security Risk Assessment of Socio-Technical Systems
 - 07 Jieting Luo (UU), A formal account of opportunism in multi-agent systems
 - 08 Rick Smetters (RUN), Advances in Model Learning for Software Systems
 - 09 Xu Xie (TUD), Data Assimilation in Discrete Event Simulations

- 10 Julienka Mollee (VUA), Moving forward: supporting physical activity behavior change through intelligent technology
 - 11 Mahdi Sargolzaei (UvA), Enabling Framework for Service-oriented Collaborative Networks
 - 12 Xixi Lu (TU/e), Using behavioral context in process mining
 - 13 Seyed Amin Tabatabaei (VUA), Computing a Sustainable Future
 - 14 Bart Joosten (TiU), Detecting Social Signals with Spatiotemporal Gabor Filters
 - 15 Naser Davarzani (UM), Biomarker discovery in heart failure
 - 16 Jaebok Kim (UT), Automatic recognition of engagement and emotion in a group of children
 - 17 Jianpeng Zhang (TU/e), On Graph Sample Clustering
 - 18 Henriette Nakad (UL), De Notaris en Private Rechtspraak
 - 19 Minh Duc Pham (VUA), Emergent relational schemas for RDF
 - 20 Manxia Liu (RUN), Time and Bayesian Networks
 - 21 Aad Sloomaker (OU), EMERGO: a generic platform for authoring and playing scenario-based serious games
 - 22 Eric Fernandes de Mello Araújo (VUA), Contagious: Modeling the Spread of Behaviours, Perceptions and Emotions in Social Networks
 - 23 Kim Schouten (EUR), Semantics-driven Aspect-Based Sentiment Analysis
 - 24 Jered Vroon (UT), Responsive Social Positioning Behaviour for Semi-Autonomous Telepresence Robots
 - 25 Riste Gligorov (VUA), Serious Games in Audio-Visual Collections
 - 26 Roelof Anne Jelle de Vries (UT), Theory-Based and Tailor-Made: Motivational Messages for Behavior Change Technology
 - 27 Maikel Leemans (TU/e), Hierarchical Process Mining for Scalable Software Analysis
 - 28 Christian Willemse (UT), Social Touch Technologies: How they feel and how they make you feel
 - 29 Yu Gu (TiU), Emotion Recognition from Mandarin Speech
 - 30 Wouter Beek (VUA), The "K" in "semantic web" stands for "knowledge": scaling semantics to the web
-
- 2019 01 Rob van Eijk (UL), Web privacy measurement in real-time bidding systems. A graph-based approach to RTB system classification
 - 02 Emmanuelle Beauxis Aussalet (CWI, UU), Statistics and Visualizations for Assessing Class Size Uncertainty
 - 03 Eduardo Gonzalez Lopez de Murillas (TU/e), Process Mining on Databases: Extracting Event Data from Real Life Data Sources
 - 04 Ridho Rahmadi (RUN), Finding stable causal structures from clinical data
 - 05 Sebastiaan van Zelst (TU/e), Process Mining with Streaming Data
 - 06 Chris Dijkshoorn (VUA), Nichesourcing for Improving Access to Linked Cultural Heritage Datasets
 - 07 Soude Fazeli (TUD), Recommender Systems in Social Learning Platforms
 - 08 Frits de Nijs (TUD), Resource-constrained Multi-agent Markov Decision Processes

- 09 Fahimeh Alizadeh Moghaddam (UvA), Self-adaptation for energy efficiency in software systems
- 10 Qing Chuan Ye (EUR), Multi-objective Optimization Methods for Allocation and Prediction
- 11 Yue Zhao (TUD), Learning Analytics Technology to Understand Learner Behavioral Engagement in MOOCs
- 12 Jacqueline Heinerman (VUA), Better Together
- 13 Guanliang Chen (TUD), MOOC Analytics: Learner Modeling and Content Generation
- 14 Daniel Davis (TUD), Large-Scale Learning Analytics: Modeling Learner Behavior & Improving Learning Outcomes in Massive Open Online Courses
- 15 Erwin Walraven (TUD), Planning under Uncertainty in Constrained and Partially Observable Environments
- 16 Guangming Li (TU/e), Process Mining based on Object-Centric Behavioral Constraint (OCBC) Models
- 17 Ali Hurriyetoglu (RUN), Extracting actionable information from microtexts
- 18 Gerard Wagenaar (UU), Artefacts in Agile Team Communication
- 19 Vincent Koeman (TUD), Tools for Developing Cognitive Agents
- 20 Chide Groenouwe (UU), Fostering technically augmented human collective intelligence
- 21 Cong Liu (TU/e), Software Data Analytics: Architectural Model Discovery and Design Pattern Detection
- 22 Martin van den Berg (VUA), Improving IT Decisions with Enterprise Architecture
- 23 Qin Liu (TUD), Intelligent Control Systems: Learning, Interpreting, Verification
- 24 Anca Dumitrache (VUA), Truth in Disagreement - Crowdsourcing Labeled Data for Natural Language Processing
- 25 Emiel van Miltenburg (VUA), Pragmatic factors in (automatic) image description
- 26 Prince Singh (UT), An Integration Platform for Synchromodal Transport
- 27 Alessandra Antonaci (OU), The Gamification Design Process applied to (Massive) Open Online Courses
- 28 Esther Kuindersma (UL), Cleared for take-off: Game-based learning to prepare airline pilots for critical situations
- 29 Daniel Formolo (VUA), Using virtual agents for simulation and training of social skills in safety-critical circumstances
- 30 Vahid Yazdanpanah (UT), Multiagent Industrial Symbiosis Systems
- 31 Milan Jelisavcic (VUA), Alive and Kicking: Baby Steps in Robotics
- 32 Chiara Sironi (UM), Monte-Carlo Tree Search for Artificial General Intelligence in Games
- 33 Anil Yaman (TU/e), Evolution of Biologically Inspired Learning in Artificial Neural Networks
- 34 Negar Ahmadi (TU/e), EEG Microstate and Functional Brain Network Features for Classification of Epilepsy and PNES

-
- 35 Lisa Facey-Shaw (OU), Gamification with digital badges in learning programming
 - 36 Kevin Ackermans (OU), Designing Video-Enhanced Rubrics to Master Complex Skills
 - 37 Jian Fang (TUD), Database Acceleration on FPGAs
 - 38 Akos Kadar (OU), Learning visually grounded and multilingual representations
-
- 2020 01 Armon Toubman (UL), Calculated Moves: Generating Air Combat Behaviour
 - 02 Marcos de Paula Bueno (UL), Unraveling Temporal Processes using Probabilistic Graphical Models
 - 03 Mostafa Deghani (UvA), Learning with Imperfect Supervision for Language Understanding
 - 04 Maarten van Gompel (RUN), Context as Linguistic Bridges
 - 05 Yulong Pei (TU/e), On local and global structure mining
 - 06 Preethu Rose Anish (UT), Stimulation Architectural Thinking during Requirements Elicitation - An Approach and Tool Support
 - 07 Wim van der Vegt (OU), Towards a software architecture for reusable game components
 - 08 Ali Mirsoleimani (UL), Structured Parallel Programming for Monte Carlo Tree Search
 - 09 Myriam Traub (UU), Measuring Tool Bias and Improving Data Quality for Digital Humanities Research
 - 10 Alifah Syamsiyah (TU/e), In-database Preprocessing for Process Mining
 - 11 Sepideh Mesbah (TUD), Semantic-Enhanced Training Data Augmentation Methods for Long-Tail Entity Recognition Models
 - 12 Ward van Breda (VUA), Predictive Modeling in E-Mental Health: Exploring Applicability in Personalised Depression Treatment
 - 13 Marco Virgolin (CWI), Design and Application of Gene-pool Optimal Mixing Evolutionary Algorithms for Genetic Programming
 - 14 Mark Raasveldt (CWI/UL), Integrating Analytics with Relational Databases
 - 15 Konstantinos Georgiadis (OU), Smart CAT: Machine Learning for Configurable Assessments in Serious Games
 - 16 Ilona Wilmont (RUN), Cognitive Aspects of Conceptual Modelling
 - 17 Daniele Di Mitri (OU), The Multimodal Tutor: Adaptive Feedback from Multimodal Experiences
 - 18 Georgios Methenitis (TUD), Agent Interactions & Mechanisms in Markets with Uncertainties: Electricity Markets in Renewable Energy Systems
 - 19 Guido van Capelleveen (UT), Industrial Symbiosis Recommender Systems
 - 20 Albert Hankel (VUA), Embedding Green ICT Maturity in Organisations
 - 21 Karine da Silva Miras de Araujo (VUA), Where is the robot?: Life as it could be
 - 22 Maryam Masoud Khamis (RUN), Understanding complex systems implementation through a modeling approach: the case of e-government in Zanzibar
 - 23 Rianne Conijn (UT), The Keys to Writing: A writing analytics approach to studying writing processes using keystroke logging
 - 24 Lenin da Nóbrega Medeiros (VUA/RUN), How are you feeling, human? Towards emotionally supportive chatbots

-
- 25 Xin Du (TU/e), The Uncertainty in Exceptional Model Mining
 - 26 Krzysztof Leszek Sadowski (UU), GAMBIT: Genetic Algorithm for Model-Based mixed-Integer opTimization
 - 27 Ekaterina Muravyeva (TUD), Personal data and informed consent in an educational context
 - 28 Bibeg Limbu (TUD), Multimodal interaction for deliberate practice: Training complex skills with augmented reality
 - 29 Ioan Gabriel Bucur (RUN), Being Bayesian about Causal Inference
 - 30 Bob Zadok Blok (UL), Creatief, Creatiever, Creatiefst
 - 31 Gongjin Lan (VUA), Learning better – From Baby to Better
 - 32 Jason Rhuggenaath (TU/e), Revenue management in online markets: pricing and online advertising
 - 33 Rick Gilsing (TU/e), Supporting service-dominant business model evaluation in the context of business model innovation
 - 34 Anna Bon (UM), Intervention or Collaboration? Redesigning Information and Communication Technologies for Development
 - 35 Siamak Farshidi (UU), Multi-Criteria Decision-Making in Software Production
-
- 2021 01 Francisco Xavier Dos Santos Fonseca (TUD), Location-based Games for Social Interaction in Public Space
 - 02 Rijk Mercur (TUD), Simulating Human Routines: Integrating Social Practice Theory in Agent-Based Models
 - 03 Seyyed Hadi Hashemi (UvA), Modeling Users Interacting with Smart Devices
 - 04 Ioana Jivet (OU), The Dashboard That Loved Me: Designing adaptive learning analytics for self-regulated learning
 - 05 Davide Dell'Anna (UU), Data-Driven Supervision of Autonomous Systems
 - 06 Daniel Davison (UT), "Hey robot, what do you think?" How children learn with a social robot
 - 07 Armel Lefebvre (UU), Research data management for open science
 - 08 Nardie Fanchamps (OU), The Influence of Sense-Reason-Act Programming on Computational Thinking
 - 09 Cristina Zaga (UT), The Design of Robothings. Non-Anthropomorphic and Non-Verbal Robots to Promote Children's Collaboration Through Play
 - 10 Quinten Meertens (UvA), Misclassification Bias in Statistical Learning
 - 11 Anne van Rossum (UL), Nonparametric Bayesian Methods in Robotic Vision
 - 12 Lei Pi (UL), External Knowledge Absorption in Chinese SMEs
 - 13 Bob R. Schadenberg (UT), Robots for Autistic Children: Understanding and Facilitating Predictability for Engagement in Learning
 - 14 Negin Samaeemofrad (UL), Business Incubators: The Impact of Their Support
 - 15 Onat Ege Adali (TU/e), Transformation of Value Propositions into Resource Re-Configurations through the Business Services Paradigm
 - 16 Esam A. H. Ghaleb (UM), Bimodal emotion recognition from audio-visual cues
 - 17 Dario Dotti (UM), Human Behavior Understanding from motion and bodily cues using deep neural networks

-
- 18 Remi Wieten (UU), Bridging the Gap Between Informal Sense-Making Tools and Formal Systems - Facilitating the Construction of Bayesian Networks and Argumentation Frameworks
 - 19 Roberto Verdecchia (VUA), Architectural Technical Debt: Identification and Management
 - 20 Masoud Mansoury (TU/e), Understanding and Mitigating Multi-Sided Exposure Bias in Recommender Systems
 - 21 Pedro Thiago Timbó Holanda (CWI), Progressive Indexes
 - 22 Sihang Qiu (TUD), Conversational Crowdsourcing
 - 23 Hugo Manuel Proença (UL), Robust rules for prediction and description
 - 24 Kaijie Zhu (TU/e), On Efficient Temporal Subgraph Query Processing
 - 25 Eoin Martino Grua (VUA), The Future of E-Health is Mobile: Combining AI and Self-Adaptation to Create Adaptive E-Health Mobile Applications
 - 26 Benno Kruit (CWI/VUA), Reading the Grid: Extending Knowledge Bases from Human-readable Tables
 - 27 Jelte van Waterschoot (UT), Personalized and Personal Conversations: Designing Agents Who Want to Connect With You
 - 28 Christoph Selig (UL), Understanding the Heterogeneity of Corporate Entrepreneurship Programs
-
- 2022 01 Judith van Stegeren (UT), Flavor text generation for role-playing video games
 - 02 Paulo da Costa (TU/e), Data-driven Prognostics and Logistics Optimisation: A Deep Learning Journey
 - 03 Ali el Hassouni (VUA), A Model A Day Keeps The Doctor Away: Reinforcement Learning For Personalized Healthcare
 - 04 Ünal Aksu (UU), A Cross-Organizational Process Mining Framework
 - 05 Shiwei Liu (TU/e), Sparse Neural Network Training with In-Time Over-Parameterization
 - 06 Reza Refaei Afshar (TU/e), Machine Learning for Ad Publishers in Real Time Bidding
 - 07 Sambit Praharaj (OU), Measuring the Unmeasurable? Towards Automatic Co-located Collaboration Analytics
 - 08 Maikel L. van Eck (TU/e), Process Mining for Smart Product Design
 - 09 Oana Andreea Inel (VUA), Understanding Events: A Diversity-driven Human-Machine Approach
 - 10 Felipe Moraes Gomes (TUD), Examining the Effectiveness of Collaborative Search Engines
 - 11 Mirjam de Haas (UT), Staying engaged in child-robot interaction, a quantitative approach to studying preschoolers' engagement with robots and tasks during second-language tutoring
 - 12 Guanyi Chen (UU), Computational Generation of Chinese Noun Phrases
 - 13 Xander Wilcke (VUA), Machine Learning on Multimodal Knowledge Graphs: Opportunities, Challenges, and Methods for Learning on Real-World Heterogeneous and Spatially-Oriented Knowledge
 - 14 Michiel Overeem (UU), Evolution of Low-Code Platforms

- 15 Jelmer Jan Koorn (UU), Work in Process: Unearthing Meaning using Process Mining
 - 16 Pieter Gijbbers (TU/e), Systems for AutoML Research
 - 17 Laura van der Lubbe (VUA), Empowering vulnerable people with serious games and gamification
 - 18 Paris Mavromoustakos Blom (TiU), Player Affect Modelling and Video Game Personalisation
 - 19 Bilge Yigit Ozkan (UU), Cybersecurity Maturity Assessment and Standardisation
 - 20 Fakhra Jabeen (VUA), Dark Side of the Digital Media - Computational Analysis of Negative Human Behaviors on Social Media
 - 21 Seethu Mariyam Christopher (UM), Intelligent Toys for Physical and Cognitive Assessments
 - 22 Alexandra Sierra Rativa (TiU), Virtual Character Design and its potential to foster Empathy, Immersion, and Collaboration Skills in Video Games and Virtual Reality Simulations
 - 23 Ilir Kola (TUD), Enabling Social Situation Awareness in Support Agents
 - 24 Samaneh Heidari (UU), Agents with Social Norms and Values - A framework for agent based social simulations with social norms and personal values
 - 25 Anna L.D. Latour (UL), Optimal decision-making under constraints and uncertainty
 - 26 Anne Dirkson (UL), Knowledge Discovery from Patient Forums: Gaining novel medical insights from patient experiences
 - 27 Christos Athanasiadis (UM), Emotion-aware cross-modal domain adaptation in video sequences
 - 28 Onuralp Ulusoy (UU), Privacy in Collaborative Systems
 - 29 Jan Kolkmeier (UT), From Head Transform to Mind Transplant: Social Interactions in Mixed Reality
 - 30 Dean De Leo (CWI), Analysis of Dynamic Graphs on Sparse Arrays
 - 31 Konstantinos Traganos (TU/e), Tackling Complexity in Smart Manufacturing with Advanced Manufacturing Process Management
 - 32 Cezara Pastrav (UU), Social simulation for socio-ecological systems
 - 33 Brinn Hekkelman (CWI/TUD), Fair Mechanisms for Smart Grid Congestion Management
 - 34 Nimat Ullah (VUA), Mind Your Behaviour: Computational Modelling of Emotion & Desire Regulation for Behaviour Change
 - 35 Mike E.U. Ligthart (VUA), Shaping the Child-Robot Relationship: Interaction Design Patterns for a Sustainable Interaction
-
- 2023 01 Bojan Simoski (VUA), Untangling the Puzzle of Digital Health Interventions
 - 02 Mariana Rachel Dias da Silva (TiU), Grounded or in flight? What our bodies can tell us about the whereabouts of our thoughts
 - 03 Shabnam Najafian (TUD), User Modeling for Privacy-preserving Explanations in Group Recommendations
 - 04 Gineke Wiggers (UL), The Relevance of Impact: bibliometric-enhanced legal information retrieval

-
- 05 Anton Bouter (CWI), Optimal Mixing Evolutionary Algorithms for Large-Scale Real-Valued Optimization, Including Real-World Medical Applications
 - 06 António Pereira Barata (UL), Reliable and Fair Machine Learning for Risk Assessment
 - 07 Tianjin Huang (TU/e), The Roles of Adversarial Examples on Trustworthiness of Deep Learning
 - 08 Lu Yin (TU/e), Knowledge Elicitation using Psychometric Learning
 - 09 Xu Wang (VUA), Scientific Dataset Recommendation with Semantic Techniques
 - 10 Dennis J.N.J. Soemers (UM), Learning State-Action Features for General Game Playing
 - 11 Fawad Taj (VUA), Towards Motivating Machines: Computational Modeling of the Mechanism of Actions for Effective Digital Health Behavior Change Applications
 - 12 Tessel Bogaard (VUA), Using Metadata to Understand Search Behavior in Digital Libraries
 - 13 Injy Sarhan (UU), Open Information Extraction for Knowledge Representation
 - 14 Selma Čaušević (TUD), Energy resilience through self-organization
 - 15 Alvaro Henrique Chaim Correia (TU/e), Insights on Learning Tractable Probabilistic Graphical Models
 - 16 Peter Blomsma (TiU), Building Embodied Conversational Agents: Observations on human nonverbal behaviour as a resource for the development of artificial characters
 - 17 Meike Nauta (UT), Explainable AI and Interpretable Computer Vision –From Oversight to Insight
 - 18 Gustavo Penha (TUD), Designing and Diagnosing Models for Conversational Search and Recommendation
 - 19 George Aalbers (TiU), Digital Traces of the Mind: Using Smartphones to Capture Signals of Well-Being in Individuals
 - 20 Arkadiy Dushatskiy (TUD), Expensive Optimization with Model-Based Evolutionary Algorithms applied to Medical Image Segmentation using Deep Learning
 - 21 Gerrit Jan de Bruin (UL), Network Analysis Methods for Smart Inspection in the Transport Domain
 - 22 Alireza Shojaiifar (UU), Volitional Cybersecurity
 - 23 Theo Theunissen (UU), Documentation in Continuous Software Development
 - 24 Agathe Balayn (TUD), Practices Towards Hazardous Failure Diagnosis in Machine Learning
 - 25 Jurian Baas (UU), Entity Resolution on Historical Knowledge Graphs
 - 26 Loek Tonnaer (TU/e), Linearly Symmetry-Based Disentangled Representations and their Out-of-Distribution Behaviour
 - 27 Ghada Sokar (TU/e), Learning Continually Under Changing Data Distributions
 - 28 Floris den Hengst (VUA), Learning to Behave: Reinforcement Learning in Human Contexts
 - 29 Tim Draws (TUD), Understanding Viewpoint Biases in Web Search Results
-

- 2024 01 Daphne Miedema (TU/e), On Learning SQL: Disentangling concepts in data systems education
- 02 Emile van Krieken (VUA), Optimisation in Neurosymbolic Learning Systems
- 03 Feri Wijayanto (RUN), Automated Model Selection for Rasch and Mediation Analysis
- 04 Mike Huisman (UL), Understanding Deep Meta-Learning
- 05 Yiyong Gou (UM), Aerial Robotic Operations: Multi-environment Cooperative Inspection & Construction Crack Autonomous Repair
- 06 Azqa Nadeem (TUD), Understanding Adversary Behavior via XAI: Leveraging Sequence Clustering to Extract Threat Intelligence
- 07 Parisa Shayan (TiU), Modeling User Behavior in Learning Management Systems
- 08 Xin Zhou (UvA), From Empowering to Motivating: Enhancing Policy Enforcement through Process Design and Incentive Implementation
- 09 Giso Dal (UT), Probabilistic Inference Using Partitioned Bayesian Networks
- 10 Cristina-Iulia Bucur (VUA), Linkflows: Towards Genuine Semantic Publishing in Science
- 11 withdrawn
- 12 Peide Zhu (TUD), Towards Robust Automatic Question Generation For Learning
- 13 Enrico Liscio (TUD), Context-Specific Value Inference via Hybrid Intelligence
- 14 Larissa Capobianco Shimomura (TU/e), On Graph Generating Dependencies and their Applications in Data Profiling
- 15 Ting Liu (VUA), A Gut Feeling: Biomedical Knowledge Graphs for Interrelating the Gut Microbiome and Mental Health
- 16 Arthur Barbosa Câmara (TUD), Designing Search-as-Learning Systems
- 17 Razieh Alidoosti (VUA), Ethics-aware Software Architecture Design
- 18 Laurens Stoop (UU), Data Driven Understanding of Energy-Meteorological Variability and its Impact on Energy System Operations
- 19 Azadeh Mozafari Mehr (TU/e), Multi-perspective Conformance Checking: Identifying and Understanding Patterns of Anomalous Behavior
- 20 Ritsart Anne Plantenga (UL), Omgang met Regels
- 21 Federica Vinella (UU), Crowdsourcing User-Centered Teams
- 22 Zeynep Ozturk Yurt (TU/e), Beyond Routine: Extending BPM for Knowledge-Intensive Processes with Controllable Dynamic Contexts
- 23 Jie Luo (VUA), Lamarck's Revenge: Inheritance of Learned Traits Improves Robot Evolution
- 24 Nirmal Roy (TUD), Exploring the effects of interactive interfaces on user search behaviour
- 25 Alisa Rieger (TUD), Striving for Responsible Opinion Formation in Web Search on Debated Topics
- 26 Tim Gubner (CWI), Adaptively Generating Heterogeneous Execution Strategies using the VOILA Framework
- 27 Lincen Yang (UL), Information-theoretic Partition-based Models for Interpretable Machine Learning

- 28 Leon Helwerda (UL), Grip on Software: Understanding development progress of Scrum sprints and backlogs
 - 29 David Wilson Romero Guzman (VUA), The Good, the Efficient and the Inductive Biases: Exploring Efficiency in Deep Learning Through the Use of Inductive Biases
 - 30 Vijanti Ramautar (UU), Model-Driven Sustainability Accounting
 - 31 Ziyu Li (TUD), On the Utility of Metadata to Optimize Machine Learning Workflows
 - 32 Vinicius Stein Dani (UU), The Alpha and Omega of Process Mining
 - 33 Siddharth Mehrotra (TUD), Designing for Appropriate Trust in Human-AI interaction
 - 34 Robert Deckers (VUA), From Smallest Software Particle to System Specification - MuDForM: Multi-Domain Formalization Method
 - 35 Sicui Zhang (TU/e), Methods of Detecting Clinical Deviations with Process Mining: a fuzzy set approach
 - 36 Thomas Mulder (TU/e), Optimization of Recursive Queries on Graphs
 - 37 James Graham Nevin (UvA), The Ramifications of Data Handling for Computational Models
 - 38 Christos Koutras (TUD), Tabular Schema Matching for Modern Settings
 - 39 Paola Lara Machado (TU/e), The Nexus between Business Models and Operating Models: From Conceptual Understanding to Actionable Guidance
 - 40 Montijn van de Ven (TU/e), Guiding the Definition of Key Performance Indicators for Business Models
 - 41 Georgios Siachamis (TUD), Adaptivity for Streaming Dataflow Engines
 - 42 Emmeke Veltmeijer (VUA), Small Groups, Big Insights: Understanding the Crowd through Expressive Subgroup Analysis
 - 43 Cedric Waterschoot (KNAW Meertens Instituut), The Constructive Conundrum: Computational Approaches to Facilitate Constructive Commenting on Online News Platforms
 - 44 Marcel Schmitz (OU), Towards learning analytics-supported learning design
 - 45 Sara Salimzadeh (TUD), Living in the Age of AI: Understanding Contextual Factors that Shape Human-AI Decision-Making
 - 46 Georgios Stathis (Leiden University), Preventing Disputes: Preventive Logic, Law & Technology
 - 47 Daniel Daza (VUA), Exploiting Subgraphs and Attributes for Representation Learning on Knowledge Graphs
 - 48 Ioannis Petros Samiotis (TUD), Crowd-Assisted Annotation of Classical Music Compositions
-
- 2025 01 Max van Haastrecht (UL), Transdisciplinary Perspectives on Validity: Bridging the Gap Between Design and Implementation for Technology-Enhanced Learning Systems
 - 02 Jurgen van den Hoogen (JADS), Time Series Analysis Using Convolutional Neural Networks
 - 03 Andra-Denis Ionescu (TUD), Feature Discovery for Data-Centric AI
 - 04 Rianne Schouten (TU/e), Exceptional Model Mining for Hierarchical Data

- 05 Nele Albers (TUD), Psychology-Informed Reinforcement Learning for Situated Virtual Coaching in Smoking Cessation
- 06 Daniël Vos (TUD), Decision Tree Learning: Algorithms for Robust Prediction and Policy Optimization
- 07 Ricky Maulana Fajri (TU/e), Towards Safer Active Learning: Dealing with Unwanted Biases, Graph-Structured Data, Adversary, and Data Imbalance
- 08 Stefan Bloemheuvel (TiU), Spatio-Temporal Analysis Through Graphs: Predictive Modeling and Graph Construction
- 09 Fadime Kaya (VUA), Decentralized Governance Design - A Model-Based Approach
- 10 Zhao Yang (UL), Enhancing Autonomy and Efficiency in Goal-Conditioned Reinforcement Learning
- 11 Shahin Sharifi Noorian (TUD), From Recognition to Understanding: Enriching Visual Models Through Multi-Modal Semantic Integration
- 12 Lijun Lyu (TUD), Interpretability in Neural Information Retrieval
- 13 Fuda van Diggelen (VUA), Robots Need Some Education: on the complexity of learning in evolutionary robotics
- 14 Gennaro Gala (TU/e), Probabilistic Generative Modeling with Latent Variable Hierarchies
- 15 Michiel van der Meer (UL), Opinion Diversity through Hybrid Intelligence
- 16 Monika Grewal (TU Delft), Deep Learning for Landmark Detection, Segmentation, and Multi-Objective Deformable Registration in Medical Imaging
- 17 Matteo De Carlo (VUA), Real Robot Reproduction: Towards Evolving Robotic Ecosystems
- 18 Anouk Neerinx (UU), Robots That Care: How Social Robots Can Boost Children's Mental Wellbeing
- 19 Fang Hou (UU), Trust in Software Ecosystems
- 20 Alexander Melchior (UU), Modelling for Policy is More Than Policy Modelling (The Useful Application of Agent-Based Modelling in Complex Policy Processes)
- 21 Mandani Ntekouli (UM), Bridging Individual and Group Perspectives in Psychopathology: Computational Modeling Approaches using Ecological Momentary Assessment Data
- 22 Hilde Weerts (TU/e), Decoding Algorithmic Fairness: Towards Interdisciplinary Understanding of Fairness and Discrimination in Algorithmic Decision-Making
- 23 Roderick van der Weerd (VUA), IoT Measurement Knowledge Graphs: Constructing, Working and Learning with IoT Measurement Data as a Knowledge Graph
- 24 Zhong Li (UL), Trustworthy Anomaly Detection for Smart Manufacturing
- 25 Kyana van Eijndhoven (TiU), A Breakdown of Breakdowns: Multi-Level Team Coordination Dynamics under Stressful Conditions
- 26 Tom Pepels (UM), Monte-Carlo Tree Search is Work in Progress
- 27 Danil Provodin (JADS, TU/e), Sequential Decision Making Under Complex Feedback

- 28 Jinke He (TU Delft), Exploring Learned Abstract Models for Efficient Planning and Learning
- 29 Erik van Haeringen (VUA), Mixed Feelings: Simulating Emotion Contagion in Groups
- 30 Myrthe Reuver (VUA), A Puzzle of Perspectives: Interdisciplinary Language Technology for Responsible News Recommendation
- 31 Gebrekirstos Gebreselassie Gebremeskel (RUN), Spotlight on Recommender Systems: Contributions to Selected Components in the Recommendation Pipeline
- 32 Ryan Brate (UU), Words Matter: A Computational Toolkit for Charged Terms
- 33 Merle Reimann (VUA), Speaking the Same Language: Spoken Capability Communication in Human-Agent and Human-Robot Interaction
- 34 Eduard C. Groen (UU), Crowd-Based Requirements Engineering
- 35 Urja Khurana (VUA), From Concept To Impact: Toward More Robust Language Model Deployment
- 36 Anna Maria Wegmann (UU), Say the Same but Differently: Computational Approaches to Stylistic Variation and Paraphrasing
- 37 Chris Kamphuis (RUN), Exploring Relations and Graphs for Information Retrieval
- 38 Valentina Maccatrozzo (VUA), Break the Bubble: Semantic Patterns for Serendipity
- 39 Dimitrios Alivanistos (VUA), Knowledge Graphs & Transformers for Hypothesis Generation: Accelerating Scientific Discovery in the Era of Artificial Intelligence
- 40 Stefan Grafberger (UvA), Declarative Machine Learning Pipeline Management via Logical Query Plans
- 41 Mozghan Vazifehdoostirani (TU/e), Leveraging Process Flexibility to Improve Process Outcome - From Descriptive Analytics to Actionable Insights
- 42 Margherita Martorana (VUA), Semantic Interpretation of Dataless Tables: a metadata-driven approach for findable, accessible, interoperable and reusable restricted access data
- 43 Krist Shingjergji (OU), Sense the Classroom - Using AI to Detect and Respond to Learning-Centered Affective States in Online Education
- 44 Robbert Reijnen (TU/e), Dynamic Algorithm Configuration for Machine Scheduling Using Deep Reinforcement Learning
- 45 Anjana Mohandas Sheeladevi (VUA), Occupant-Centric Energy Management: Balancing Privacy, Well-being and Sustainability in Smart Buildings
- 46 Ya Song (TU/e), Graph Neural Networks for Modeling Temporal and Spatial Dimensions in Industrial Decision-making
- 47 Tom Kouwenhoven (UL), Collaborative Meaning-Making. The Emergence of Novel Languages in Humans, Machines, and Human-Machine Interactions
- 48 Evy van Weelden (TiU), Integrating Virtual Reality and Neurophysiology in Flight Training
- 49 Selene Báez Santamaría (VUA), Knowledge-centered conversational agents with a drive to learn

-
- 50 Lea Krause (VUA), Contextualising Conversational AI
 - 51 Jiaxu Zhao (TU/e), Understanding and Mitigating Unwanted Biases in Generative Language Models
 - 52 Qiao Xiao (TU/e), Model, Data and Communication Sparsity for Efficient Training of Neural Networks
 - 53 Gaole He (TUD), Towards Effective Human-AI Collaboration: Promoting Appropriate Reliance on AI Systems
 - 54 Go Sugimoto (VUA), MISSING LINKS Investigating the Quality of Linked Data and its Tools in Cultural Heritage and Digital Humanities
 - 55 Sietze Kai Kuilman (TUD), AI that Glitters is Not Gold: Requirements for Meaningful Control of AI Systems
 - 56 Wijnand van Woerkom (UU), A Fortiori Case-Based Reasoning: Formal Studies with Applications in Artificial Intelligence and Law
 - 57 Syeda Amna Sohail (UT), Privacy-Utility Trade-Off in Healthcare Metadata Sharing and Beyond: A Normative and Empirical Evaluation at Inter and Intra Organizational Levels
 - 58 Junhan Wen (TUD), "From iMage to Market": Machine-Learning-Empowered Fruit Supply
 - 59 Mohsen Abbaspour Onari (TU/e), From Explanation to Trust: Modeling and Measuring Trust in Explainable Decision Support
 - 60 Marcel Jurriaan Robeer (UU), Beyond Trust: A Causal Approach to Explainable AI in Law Enforcement
 - 61 Shuai Wang (VUA), Links in Large Integrated Knowledge Graphs: Analysis, Refinement, and Domain Applications
 - 62 Khaleel Asyraaf Mat Sanusi (OU), Augmenting a learning model within immersive learning environments for psychomotor skills
 - 63 Rashid Zaman (TU/e), Online Conformance Checking on Degraded Data
 - 64 Jens d'Hondt (TU/e), Effective and Efficient Multivariate Similarity Search
 - 65 Aswin Balasubramaniam (UT), Disentangling Runner Drone Interaction Potentialities
-
- 2026 01 Pei-Yu Chen (TUD), Human-Agent Alignment Dialogues: Eliciting User Information at Runtime for Personalized Behavior Support
 - 02 Hezha Hassan Mohammedkhan (TiU), Estimating Body Measurements of Children from 2D Images: Towards the Automatic Detection of Malnutrition
 - 03 Kyriakos Psarakis (TUD), Democratizing Scalable Cloud Applications: Transactional Stateful Functions on Streaming Dataflows
 - 04 Boyu Xu (UU), Exploring Indirect Relations Between Topics in Neuroscience Literature Using Augmented Reality to Inform Experimental Design
 - 05 Koen Minartz (TU/e), Stochastic Simulation with Geometric Deep Generative Models
 - 06 Azim Afroozeh (CWI, VUA), FastLanes: A Next-Gen File Format
 - 07 Inès Blin (VUA), Narrative Understanding with Knowledge Graphs
 - 08 Paul van Vulpen (UU), Debating Digital Dominance: Decentralized Technology Governance For Strategic Autonomy

- 09 Afrizal Doewes (TU/e), Rethinking Automated Essay Scoring: Agreement, Fairness, and Feedback
- 10 Nikolaos Delapaschos Kondylidis (VUA), Establishing Task-Oriented Understanding between Agents
- 11 Işıl Baysal Erez (UT), Handling Missing Data with Meta-Learning and Large Language Models
- 12 Xue Li (UvA), From Fine-tuning to Prompting: A Paradigm Shift in Knowledge Graph Construction
- 13 Isaac da Silva Torres (VUA), Guidelines To Flux Between Conceptual Models: Understanding Complex Digital Business Ecosystems
- 14 Philip Lippmann (TUD), Synthetic Data for Robust Language Modelling
- 15 Rashmi Khazanchi (OU), Artificial Intelligence in Education: Impact of AI-Based Systems on Mathematics Achievement
- 16 Carolina Ferreira Gomes Centeio Jorge (TUD), Modelling Artificial Trust for Effective Human-AI Teamwork
- 17 Maria Tsfasman (TUD), Towards Predicting Memory in Multimodal Group Interactions
- 18 Riccardo Lo Bianco (TU/e), Deep Reinforcement Learning for Automated Decision-Making in Process Management Systems
- 19 Israel Campero Jurado (TU/e), Innovations in Optimization and Applications in Healthcare
- 20 Ifitahu Ni'mah (TU/e), Contrastive Learning and Evaluation in Low Resource Scenario of Natural Language Processing
- 21 Francisco N.F.Q. Simoes (UU), Causality, Information, and Decision-Making
- 22 Ruben Verhagen (TUD), Transparent and Explainable Agents for Human-Agent Teaming
- 23 Eduardo Calò (UU), Automatically Expressing the Meaning of Logical Formulae in Natural Language
- 24 Shuai Han (UU), Improving Sample Efficiency of Reinforcement Learning: Exploiting Structural Knowledge for Decision Making
- 25 Francis Saa-Dittoh (VUA), From Radio to AI: African Community-Driven Development of Sustainable Information Systems
- 26 Mohammed Al Owayyed (TUD), Interactive Simulation-Based Learning Tools for Training Children's Helpline Counsellors
- 27 Bram Grooten (TU/e), Adaptive Reinforcement Learning: Lean and Dynamic Agents for Robust Generalization
- 28 Anouck Braggaar (TiU), Does This Answer Your question? Evaluating, Replicating, and Improving Task-Oriented Dialogue Systems
- 29 Afsana Khan (UM), Unlocking the Value of Data with Vertical Federated Learning
- 30 Yucheng Yang (TU/e), Generalization in Reinforcement Learning: Theories and Algorithms for Downstream Task and Multi-objective Trade-off Adaptation
- 31 Luis Pedro Silvestrin (VUA), Efficient Machine Learning for Time-Varying Data: Contributions to Time-Series Machine Learning and Transfer Learning

- 32 Giovanni Varricchione (UU), Declarative Specifications for Efficient and Safe Reinforcement Learning
- 33 Markus Funke (VUA), Carving Sustainability into Software Architecture
- 34 Melika Ayoughi (UvA), External and Internal Semantics for Visual Understanding
- 35 Clinton Cao (TUD), Modeling Behavior Patterns in Microservice Applications: Practical Applications Using Finite State Machines

