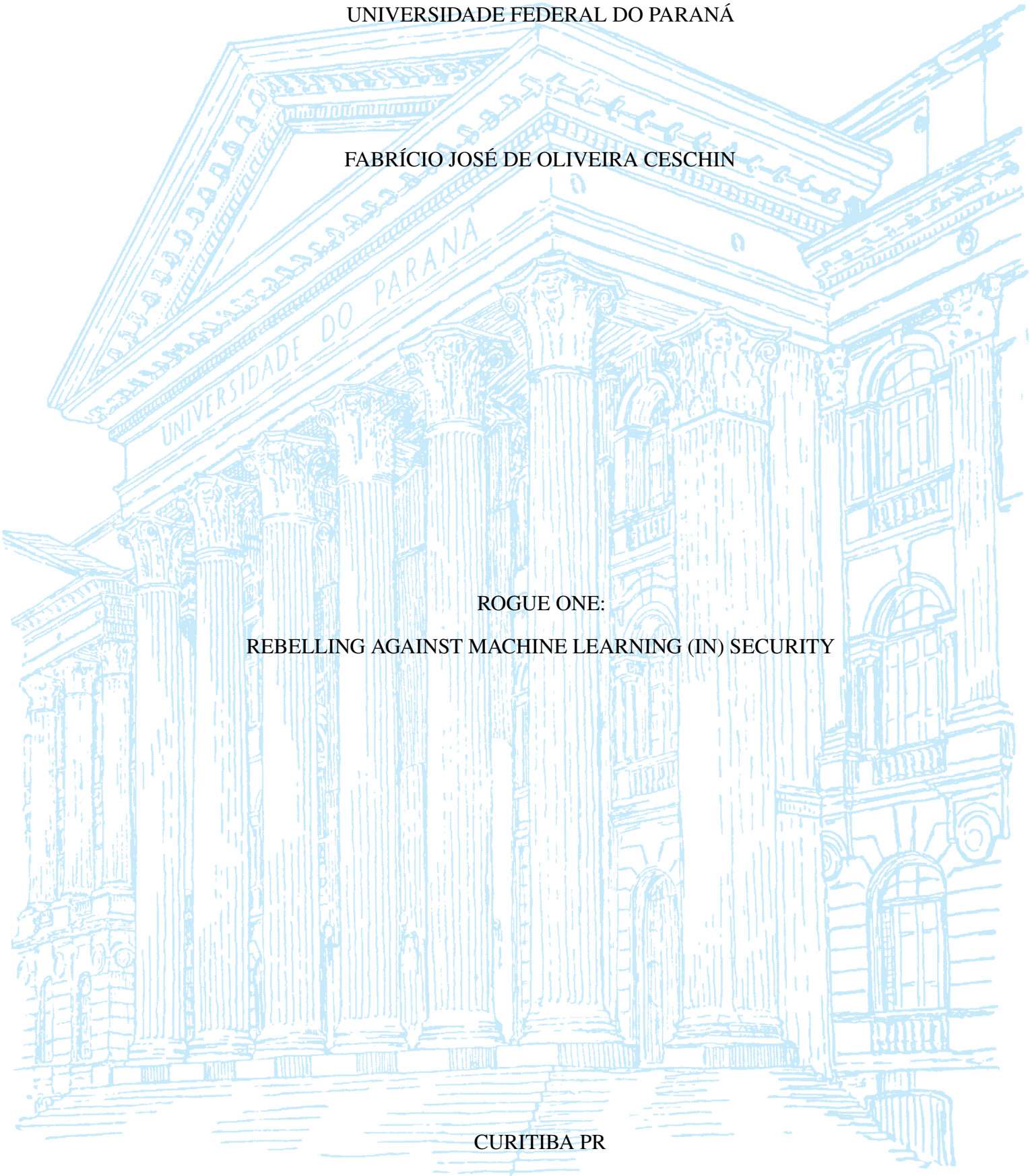
UNIVERSIDADE FEDERAL DO PARANÁ

FABRÍCIO JOSÉ DE OLIVEIRA CESCHIN



ROGUE ONE:

REBELLING AGAINST MACHINE LEARNING (IN) SECURITY

CURITIBA PR

2023

FABRÍCIO JOSÉ DE OLIVEIRA CESCHIN

ROGUE ONE:

REBELLING AGAINST MACHINE LEARNING (IN) SECURITY

Tese apresentada como requisito parcial à obtenção do grau de Doutor em Ciência da Computação no Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Ciência da Computação*.

Orientador: André Grégio.

Coorientador: Heitor Murilo Gomes.

CURITIBA PR

2023

DADOS INTERNACIONAIS DE CATALOGAÇÃO NA PUBLICAÇÃO (CIP)
UNIVERSIDADE FEDERAL DO PARANÁ
SISTEMA DE BIBLIOTECAS – BIBLIOTECA DE CIÊNCIA E TECNOLOGIA

Ceschin, Fabrício José de Oliveira
Rogue One: rebelling against machine learning (in) security / Fabrício José de Oliveira Ceschin. – Curitiba, 2023.
1 recurso on-line : PDF.

Tese (Doutorado) - Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-Graduação em Informática.

Orientador: André Ricardo Abed Grégio
Coorientador: Heitor Murilo Gomes

1. Aprendizado de máquinas. 2. Computadores - Medidas de segurança. 3. Inteligência artificial. 4. Fluxo de dados (Computadores). I. Universidade Federal do Paraná. II. Programa de Pós-Graduação em Informática. III. Grégio, André Ricardo Abed. IV. Gomes, Heitor Murilo. V. Título.

Bibliotecário: Elias Barbosa da Silva CRB-9/1894

TERMO DE APROVAÇÃO

Os membros da Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação INFORMÁTICA da Universidade Federal do Paraná foram convocados para realizar a arguição da tese de Doutorado de **FABRÍCIO JOSÉ DE OLIVEIRA CESCHIN** intitulada: **Rogue One: Rebelling Against Machine Learning (In) Security**, sob orientação do Prof. Dr. ANDRÉ RICARDO ABED GRÉGIO, que após terem inquirido o aluno e realizada a avaliação do trabalho, são de parecer pela sua APROVAÇÃO no rito de defesa.

A outorga do título de doutor está sujeita à homologação pelo colegiado, ao atendimento de todas as indicações e correções solicitadas pela banca e ao pleno atendimento das demandas regimentais do Programa de Pós-Graduação.

CURITIBA, 01 de Fevereiro de 2023.

Assinatura Eletrônica
09/03/2023 11:59:27.0
ANDRÉ RICARDO ABED GRÉGIO
Presidente da Banca Examinadora

Assinatura Eletrônica
28/02/2023 10:01:04.0
DAVID MENOTTI GOMES
Avaliador Interno (UNIVERSIDADE FEDERAL DO PARANÁ)

Assinatura Eletrônica
13/03/2023 21:36:18.0
IAN WELCH
Avaliador Externo (VICTORIA UNIVERSITY OF WELLINGTON)

Assinatura Eletrônica
28/02/2023 09:55:23.0
PAULO RICARDO LISBOA DE ALMEIDA
Avaliador Interno (UNIVERSIDADE FEDERAL DO PARANÁ)

Assinatura Eletrônica
07/03/2023 16:49:27.0
BERNHARD PFAHRINGER
Avaliador Externo (UNIVERSIDADE DE WAIKATO)

"If you want to be successful, you need to have total dedication, seek your last limit and do your best."
Ayrton Senna

ACKNOWLEDGEMENTS

This work would not have been done without the efforts of many people. When I say a lot, it is really **a lot** of people, and I will probably forget someone's name. So, if your name is not here, I am sorry, but if you helped me somehow, you are not less important.

I will start by thanking my parents, Francisco and Elizete Ceschin. I could spend all this text talking about how they supported me in this journey, I know how hard they worked to give me the best they could and I really appreciate that. Also, they were the ones that introduced me to computers when I was around 4 years old. Since then, I never stopped to investigate how it worked – my father had to take courses to fix our computer because I was always messing with the operating system and the BIOS. I'd like to also thank my whole family for supporting me.

I need to thank the Federal University of Paraná (UFPR) for the opportunity I had during all these years since I started my bachelor's in computer science. This university changed my life and I had a lot of opportunities due to its competent professors and staff members. Federal universities in Brazil are **very** important to build excellent professionals and researchers and they are essential to improve the country.

I made great friends during my graduation. Each one of them helped me in their way during this journey, we know how hard it is to graduate, but I was lucky to have great friends with me that made it a little bit less hard. So thanks Renan Burda, the first friend I made in the university, Renan Souza, João Risso, Katheryne Graf, Bruno Krinski, Daniel Enzo, Jean Diogo, Rafael Castro, Marco Mendes, Henrique Pacheco, Paulo Vieira, Antony Kossoski, Rafael Faria, Thiago Schmöckel, Reginaldo Santos, Felipe Souza, Lucas Martins, Lucas Ikeda. It is great to still have contact with all of you!

I would like to thank Caroline Hukami for all the support and patience during my masters degree and Ph.D., you helped me a lot! Thanks for understanding the weekends I spent writing papers and experiments, and the travels I had to do to present them, you are an amazing girlfriend – and now an amazing wife. I love the way you always support me when reviewer #2 decides to reject my work, you always find a way to view it in a positive way.

David Menotti, you were my mentor during my graduation and masters transition. I met you in the last semester of my graduation and you were responsible for my first contact with research papers and all the basics of machine learning. I learned a lot with you, not only in your classes but also in our meetings. Thanks for being a great co-advisor in my masters and for being a good friend. Also, thanks for introducing me to André Grégio.

Marcos Castilho, in my graduation you taught me the basics of programming. Years latter you selected me for the masters program and advised me alongside David Menotti in the beginning. Thank you for your support and understanding when I started to work with André Grégio, you gave me all the freedom for me to choose my research area.

André Grégio, you were crazy to give me root access to your server in our first meeting. You really believed in me since the beginning (you said I had a promising future as a researcher, I am not sure if I got there, but I did my best) and I liked the way you work since then. I also learned a lot from you and I am really glad for all the opportunities you gave me and also for having the opportunity to work with you. There is no masters or Ph.D. without a good advisor, you were there for me every time I needed something and I consider you not only an advisor but also a good friend. Thank you for all the advice, for introducing me to a lot of people that became my friends, and for all the moments we had over these years, you inspired me a lot.

Luiz S. Oliveira, I also learned a lot about machine learning with you. I had a lot of great ideas while having classes and meetings with you. My work would not be the same without you, you always gave me great ideas and I also consider you a friend. Thanks for introducing me to your Ph.D. and postdoc students, Paulo Almeida and Felipe Pinagé, they also helped me a lot during these years. Also, thanks for introducing me to Heitor Murilo Gomes.

Daniela Oliveira, you also believed in my work and gave me a lot of opportunities since the beginning. I enjoyed and learned a lot being part of some projects with you and while visiting your laboratory. My writing and presenting (after all, I am a visual person, right?) skills improved a lot while we worked together. Thank you for all the valuable advice and recommendations you gave me, I consider you a great friend. Also, thanks for introducing me to your students Luiz Giovanini, Nikolaos Sapountzis, Xiaoyong Yuan, Aokun Chen, Ruimin Sun, Mirela Silva, and Heng Qiao. You were all important for this thesis.

Heitor Murilo Gomes, you helped me a lot with stream learning, I learned a lot about it with you. Thanks for giving me the opportunity to visit your laboratory and to contribute to scikit-multiflow. I believe that contributing to an open-source project is one way to contribute to society somehow, and sometimes it has even more impact than publishing papers. You were an amazing co-advisor and are a great friend. Thanks for introducing me to manuka honey and great researchers Albert Bifet, Bernhard Pfahringer, Georg Krempl, Guilherme Cassales, Jacob Montiel, Nedeljko Radulovic, and Nuwan Gunasekara.

Marcus Botacin, you became a close friend on this journey. I am proud of what we achieved by using our "ping-pong method". Many papers and projects happened due to it. It was so easy to work with you that we produced a lot together without even noticing sometimes. I also learned a lot from you. Thank you for being an awesome travel mate and for showing me some good music.

Luiz Giovanini, you helped me a lot in the continuous authentication project with Daniela Oliveira. You were a great addition to the team and we did improve a lot because of you. Thank you for your patience with all the experiments. I also consider you a great friend.

Hyrum Anderson, MLSEC would not be the same without you. Thanks for organizing the competition that made me learn and produce a lot of things for my Ph.D. It is great to see how you give space for people to show their work. I remember how you were curious to understand our strategies. Also, thanks for the advice you gave me, I also consider you a great friend.

I need to thank both LarSis and SECRET research groups. I had the opportunity to meet great people there. So, thanks Adi Marcondes, Alexandre Huff, Amanda Viescinski, Anatoli Kalysch, Artur Bizon, David Gomes, Eduardo Barboza, Florian Hantke, Gabriel Lüders, Gabriel Castanhel, Jorge Correia, José Flauzino, Newton Will, Raphael Machnicki, Tiago Heinrich, Thalita Pimenta, and Vinicius Fulber. Thank you for listening to my paper review complaints, for giving me ideas, and for the company during all this time.

Marco Zanata, Carlos Maziero, Luis Bona, and Paulo Licio, I had the opportunity to work with you during the development of this thesis, and you all helped me somehow. Thank you.

Also, thanks to my evaluation committee. Ian Welch, Bernhard Pfahringer, Paulo Almeida, and David Menotti, thank you for the valuable contributions you gave to this work.

Finally, I would like to thank CAPES, Google Latin America Research Awards, and Samsung Research Brazil (SRBR) for helping me with funding to make this research possible. I also would like to thank SBSEG, for fomenting research in Brazil, USENIX, for the opportunity to attend ENIGMA twice (the first opportunity with a Diversity Grant and the second, as a speaker), MLSEC, for being an awesome challenge that helped me to develop my thesis, ROOTS conference, for the opportunity to show my experience with MLSEC, and CAMLIS, for the student travel scholarship.

RESUMO

Aprendizado de Máquina é amplamente utilizado em várias tarefas de segurança computacional hoje em dia e é considerado estado da arte pois ajuda a melhorar a detecção de novos ataques, podendo acompanhar suas evoluções. Entretanto, soluções baseadas em aprendizado de máquina podem ser muito difíceis de se avaliarem em alguns cenários, os tornando propensos a problemas que podem invalidar seus usos na prática. Uma das razões para isso é que dados de segurança seguem uma distribuição não-estacionária devido a sua natureza de sempre estar mudando para evadir a detecção, requerendo uma atenção especial. Por isso, é essencial saber como utilizar corretamente aprendizado de máquina em segurança, considerando todos os desafios que são encontrados durante a proposta ou implantação de mecanismos de defesa. Nesta tese, eu proponho investigar os principais problemas da aplicação de aprendizado de máquina em segurança, mostrando como soluções existentes falham e, em alguns casos, propondo possíveis mitigações. Baseado nisso, eu apresento uma análise crítica do estado da arte e aponto direções para os trabalhos futuros. Os principais objetivos desse trabalho são (i) entender os principais problemas de aplicar aprendizado de máquina em segurança; (ii) detectar o que pode ser melhorado; (iii) qual é o futuro de aprendizado de máquina para segurança; e (iv) reduzir a distância da indústria e academia. Finalmente, as principais contribuições dessa tese são (i) uma análise extensiva da literatura recente a respeito do uso de aprendizado de máquina em segurança de forma comparativa; (ii) direções para pesquisas de segurança considerando suas particularidades e como aplicar corretamente o aprendizado de máquina para melhorar a qualidade das soluções e permitir o uso efetivos em aplicações do mundo real; e (iii) um conjunto de módulos e *frameworks* para apoiar e melhorar futuras soluções de aprendizado de máquina para segurança que podem ser utilizados tanto pela indústria como pela academia.

Palavras-chave: Aprendizado de Máquina. Segurança Computacional. Concept Drift. Adversarial Machine Learning. Data Streams.

ABSTRACT

Machine Learning (ML) is widely used in many cybersecurity tasks nowadays and it is considered state-of-the-art because it helps to improve the detection of new attacks, keeping pace with their evolution. However, ML-based solutions may be too difficult to evaluate in some scenarios, making them prone to gaps and pitfalls that could invalidate their use in practice. One of the reasons for that is that cybersecurity data follows a non-stationary distribution due to its constantly changing nature to evade detection, requiring special attention. Thus, it is essential to know how to correctly use Machine Learning (ML) in cybersecurity, considering all the challenges that are faced during the proposal or deployment of defense solutions. In this thesis, I propose to investigate the main challenges of applying Machine Learning to cybersecurity, showing how existing solutions fail and, in some cases, proposing possible mitigations to them. Based on that, I present a critical analysis of the state-of-the-art literature and point directions toward adequate ways for future research. The main objectives of this work are to (i) understand the main problems of applying Machine Learning in cybersecurity; (ii) detect what can be improved; (iii) what is the future of Machine Learning for security; and (iv) reduce the gap between industry and academy. Finally, the main contributions of this thesis are (i) an extensive analysis of the recent literature regarding ML applied to cybersecurity in a comparative way; (ii) directions for cybersecurity research considering its particularities and how to correctly apply ML to improve quality and allow their effective use in real-world applications; and (iii) a set of modules or frameworks to support and improve further ML solutions for cybersecurity that can be used by both industry and academy.

Keywords: Machine Learning. Cybersecurity. Concept Drift. Adversarial Machine Learning. Data Streams.

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

DINF	Departamento de Informática
PPGINF	Programa de Pós-Graduação em Informática
UFPR	Universidade Federal do Paraná
ML	Machine Learning
AUC-ROC	Area Under the ROC Curve
AUC-PR	Area Under the Precision-Recall Curve
IDS	Intrusion Detection System

CONTENTS

1 INTRODUCTION

Today, a massive amount of data is produced, and, as a consequence, new emerging threats are constantly being developed by attackers, increasing the losses in the past years (Crane, 2020). Thus, Machine Learning (ML) is considered state-of-the-art and the ideal ally to build automated methods to detect attack vectors (Goh, 2018), being widely adopted by security solutions. On the one hand, it is important to have ML research in cybersecurity, once it can help to improve the detection of new attacks, keeping pace with their evolution (Gibert et al., 2020). On the other hand, ML solutions for cybersecurity have many particularities that are not inherited from other fields and may be too complex or difficult to evaluate in some scenarios, which makes them prone to gaps and pitfalls that, most of the time, are not covered in the literature and could invalidate their use in practice (Pendlebury et al., 2019; Arp et al., 2022).

In other domains, it is common to follow ML best practices when evaluating proposed solutions. For instance, imagine a simple image classification problem where we want to create a model to classify cats and dogs. This model will not change as time goes by: cats will always be cats, and so the dogs. They will always look the same, so we do not need to update a classification model every time a new dog or cat is seen, we just need a population that generalizes both classes. However, in cybersecurity, data is different. For example, according to Google, 68% of phishing emails blocked by Gmail are different from day to day (Bursztein and Oliveira, 2019), which means that their distribution is non-stationary, i.e., they are constantly being changed by attackers to evade detection.

Thus, the behavior of security data and attackers requires special attention from all the stakeholders involved, once that "blindly applying ML evaluation best practices may backfire in security contexts" (Cavallaro, 2019). Moreover, it is essential to know how to correctly use ML in cybersecurity, considering the numerous challenges that are faced before and during the deployment of defense solutions. Finally, recognizing how to evaluate a solution in a realistic scenario is also important, assuming the same circumstances that it would deal with when deployed. Recently, the cybersecurity community started to worry more about the application of machine learning in the real world. Therefore, this work is part of the movement to improve the application of machine learning in cybersecurity, which is gaining attention in the last years. In the STAR WARS universe, Rogue One (the title of this thesis) is a military call sign that means "the first rebels". I started to work in this area in 2016, I am aware that I am not alone in this quest, but I believe that I have great novel contributions to the field and that I am part of the "first rebels" of machine learning in security.

In this thesis, I propose to investigate the main challenges of applying Machine Learning to cybersecurity, showing how existing solutions fail and, in some cases, proposing possible solutions to fix them, especially for adversarial machine learning, concept drift, and evaluation, which have a special focus in papers that I published during the development of this thesis (Section 1.3.1). Thus, this thesis is mainly based on the following six papers that were published and submitted during its development, each of them with its own contribution:

1. Machine Learning (In) Security: A Stream of Problems. **Ceschin, Fabrício**, Gomes, Murilo Heitor, Botacin, Marcus, Bifet, Albert, Pfahringer, Bernhard, Oliveira, Luiz S., Grégio, André. `https://arxiv.org/abs/2010.16045`. October 2020 (Ceschin et al., 2020b): this paper is the core of this thesis, in which I present a survey on the challenges, gaps, and pitfalls that are still a problem in many scenarios of ML solutions applied to cybersecurity, suggesting, in some cases, possible mitigation for

them. The main contribution of this paper is to point directions to future cybersecurity researches that make use of ML, aiming to improve their quality to be used in real applications.

2. Shallow Security: On the Creation of Adversarial Variants to Evade Machine Learning-Based Malware Detectors. **Ceschin, Fabrício**, Botacin, Marcus, Gomes, Heitor Murilo, Oliveira, Luiz S, Grégio, André. Proceedings of the 3rd Reversing and Offensive-Oriented Trends Symposium, Association for Computing Machinery, Vienna, Austria, 2019, ISBN: 9781450377751. https://doi.org/10.1145/3375894.3375898. November 2019 (Ceschin et al., 2019): in this paper I report the experience I had in the Machine Learning Security Evasion Competition (MLSEC) 2019, detailing the methodological approach to overcome the challenge and how I managed to bypass all three malware detection models. With this paper, I expect to contribute to the community and provide a better understanding of ML-based detectors' weaknesses, pinpointing future research directions toward the development of more robust malware detectors against adversarial machine learning.

3. No need to teach new tricks to old malware: Winning an evasion challenge with xor-based adversarial samples. **Ceschin, Fabrício**, Botacin, Marcus, Lüders, Gabriel, Gomes, Heitor Murilo, Oliveira, Luiz S., Grégio, André. Proceedings of the 3rd Reversing and Offensive-Oriented Trends Symposium, Association for Computing Machinery, Vienna, Austria, 2020, ISBN: 9781450377751. https://doi.org/10.1145/3433667.3433669. November 2020 (Ceschin et al., 2020a): this paper continues my quest on investigating adversarial machine learning attacks against ML-based malware detectors, in which I describe my insights of the second edition of the Machine Learning Security Evasion Competition (MLSEC) contest regarding attacking models, as well defending them from adversarial attacks. Moreover, I show how frequency-based models (e.g., TF-IDF) are vulnerable to the addition of dead function imports and how models based on raw bytes are vulnerable to payload-embedding obfuscation (e.g., XOR and base64 encoding), which is considered a known and simple but yet effective technique.

4. Online Binary Models are Promising for Distinguishing Temporally Consistent Computer Usage Profiles. Giovanini, Luiz, **Ceschin, Fabrício**, Silva, Mirela, Chen, Aokun, Kulkarni, Ramchandra, Banda, Sanjay, Lysaght, Madison, Qiao, Heng, Sapountzis, Nikolaos, Sun, Ruimin, Matthews, Brandon, Wu, Dapeng Oliver, Grégio, André, Oliveira, Daniela. IEEE Transactions on Biometrics, Behavior, and Identity Science. https://doi.org/10.1109/TBIOM.2022.3179206. June 2022 (Giovanini et al., 2021): in this study, I investigate whether computer usage profiles are unique and consistent over time in a naturalistic setting, discussing challenges and opportunities of using them in applications of continuous authentication. After performing experiments, I found that profiles were mostly consistent over time, computer usage profiling has the potential to uniquely characterize computer users, network-related events are the most relevant features to accurately recognize profiles, and binary models are the most well-suited for profile recognition.

5. Fast & Furious: Modelling Malware Detection as Evolving Data Streams. Expert Systems with Applications. **Ceschin, Fabrício**, Botacin, Marcus, Gomes, Heitor Murilo, Pinagé, Felipe, Oliveira, Luiz S., Grégio, André. https://doi.org/10.1016/j.eswa.2022.118590. August 2022 (Ceschin et al., 2022): in this paper, I evaluate the impact of concept drift on malware classifiers for two Android datasets (DREBIN

and AndroZoo), comparing some possible approaches to mitigate drifts and proposing a novel data stream pipeline that updates both the classifier and the feature extractor and outperformed literature approaches. As a contribution, aside from our novel data stream pipeline, I show that concept drift is a generalized phenomenon in Android malware, and I made the source code publicly available, including the improved version of the datasets that include samples' timestamps to make it possible to simulate streams.

6. The SPICE Must Flow: Challenges of Modelling Security Data as Streams. **Ceschin, Fabrício**, L. P. Gomes, David, V. L. Barboza, Eduardo, R. L. de Almeida, Paulo, Grégio, André. Under Review, 2022/2023: in this work, I propose to model security data as streams to overcome current classification issues, discuss challenges faced by online approaches and how they differ from offline ones, and apply our proposal to popular Android and MS Windows malware datasets (DREBIN, AndroZoo, and EMBER), critically analyzing the results either to validate the approach. The main contributions of this paper are: identifying problems with the most popular drift detectors in the literature when applied to security data, given that they work solely based on accuracy; showing that in a realistic scenario where labels are provided with delay and may never be available the model performance drops significantly; and implementing the SPICE (Streaming Problems Investigation for Cybersecurity Evaluations) framework, an open-source library for data stream evaluation that includes modules to simulate delayed labels, labeling probability, and efficiently compute the AUC-PR by using a red-black tree.

Based on that, I present a critical analysis of the state-of-the-art literature and suggest points toward adequate ways for future research in the field, showing practical examples regarding user profiling and malware detection that may extend to other security problems, given their similarities. Finally, the main objectives of this thesis involve (i) understanding the main problems of applying Machine Learning in cybersecurity; (ii) detecting what can be improved; (iii) discussing potential challenges and solutions regarding the future of Machine Learning for security; and (iv) reducing the gap between industry and academy using problems that are common in the real world.

1.1 RESEARCH QUESTIONS

In this thesis, I will delve into the following questions:

- **What are the main issues of using Machine Learning for cybersecurity in practice?** There are many challenges related to the use of Machine Learning in cybersecurity and, even with new research being released daily, they receive little attention in the literature and generally are not considered (Arp et al., 2022). Therefore, I reviewed the literature and pinpointed the main problems according to the Machine Learning step performed in the pipeline.

- **What can be improved in the Machine Learning pipeline related to cybersecurity solutions?** According to the problems listed in the previous research question, I analyzed the points that need more attention in the literature, proposing, when possible, mitigations to them.

- **What could be the future of Machine Learning for cybersecurity?** Following the problems found and possible mitigations proposed, I analyzed what are the next steps

for Machine Learning to be used in future cybersecurity solutions in the real world, in a way that they can be feasible and reliable.

1.2 METHODOLOGY

This thesis is mainly organized and presented as a collection of articles. To better understand how I developed the several topics of the thesis, I detail below the employed methodology. Our proposed research methodology was divided into four steps, as shown in Figure 1.1. It is worth noticing that the methodology's process is a cycle, i.e., every step may (and should) be revisited as new research and solutions are developed, similar to the scientific method (Science Buddies, 2020), in which further research questions are elaborated in response to new results.

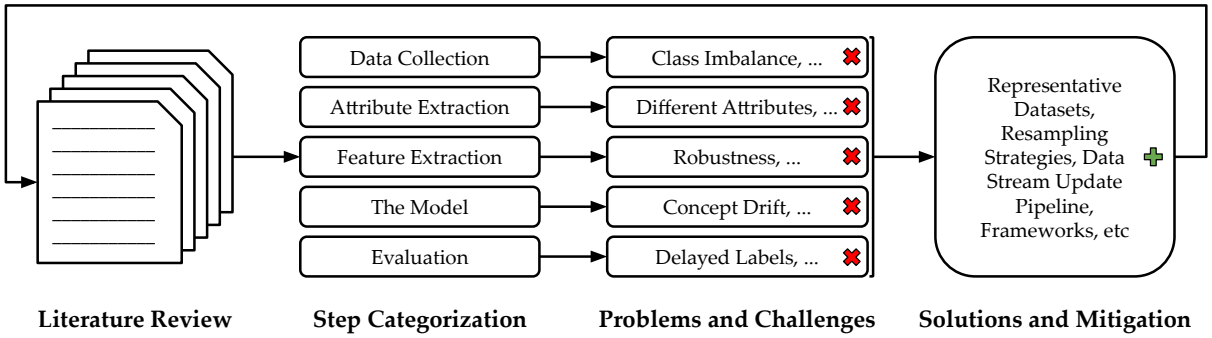


Figure 1.1: **Methodology used in this thesis.** An incremental cycle is used to categorize, list problems and challenges, and propose solutions and mitigations.

Thus, every step of our methodology is explained in detail below:

1. **Literature Review:** collection of related and state-of-the-art works, which will be used as a baseline for our research;

2. **Step Categorization:** categorization of each work according to the ML pipeline steps that it is related to. Note that a single work might be related to many steps;

3. **Problems and Challenges:** list problems and challenges related to each step according to existing solutions in the literature. In some cases, practical experiments are applied to evidence such problems;

4. **Solutions and Mitigation:** propose novel solutions in order to solve or mitigate some problems listed before. Since new works are constantly being released, it is important to return to step 1 to make the literature review updated, which may result in new challenges.

1.3 CONTRIBUTIONS

Our main contributions are threefold:

- Collect, present, and critically analyze the recent literature regarding ML applied to cybersecurity in a comparative way.

- Based on the previous item, propose and provide established directions for cybersecurity research, considering the particularities of security data, and that correctly applies ML so as to improve quality and allow their effective use in real-world applications.

- A set of modules or frameworks to support and improve further ML solutions for cybersecurity, aiming to reduce the gap between industry and academy.

Moreover, all my academic contributions are presented in the subsections below.

1.3.1 Publications

1. **Ceschin, Fabrício**, Botacin, Marcus, Gomes, Heitor Murilo, Pinagé, Felipe, Oliveira, Luiz S., Grégio, André. Fast & Furious: Modelling Malware Detection as Evolving Data Streams. Expert Systems with Applications. https://doi.org/10.1016/j.eswa.2022.118590. August 2022.

2. Giovanini, Luiz, **Ceschin, Fabrício**, Silva, Mirela, Chen, Aokun, Kulkarni, Ramchandra, Banda, Sanjay, Lysaght, Madison, Qiao, Heng, Sapountzis, Nikolaos, Sun, Ruimin, Matthews, Brandon, Wu, Dapeng Oliver, Grégio, André, Oliveira, Daniela. Online Binary Models are Promising for Distinguishing Temporally Consistent Computer Usage Profiles. IEEE Transactions on Biometrics, Behavior, and Identity Science. https://doi.org/10.1109/TBIOM.2022.3179206. June 2022.

3. Botacin, Marcus, Domingues, Felipe Duarte, **Ceschin, Fabrício**, Machnicki, Raphael, Alves, Marco Antonio Zanata, de Geus, Paulo Lício, Grégio, André. AntiViruses under the Microscope: A Hands-On Perspective. Computers & Security, pp. 102500, 2021, ISSN: 0167-4048. https://doi.org/10.1016/j.cose.2021.102500. January 2022.

4. Castanhel, Gabriel R., Heinrich, Tiago, **Ceschin, Fabrício**, Maziero, Carlos. Taking a Peek: An Evaluation of Anomaly Detection Using System calls for Containers. 2021 IEEE Symposium on Computers and Communications (ISCC). https://doi.org/10.1109/ISCC53001.2021.9631251. December 2021.

5. Silva, Mirela, **Ceschin, Fabrício**, Shrestha, Prakash, Brant, Christopher, Gilda, Shlok, Fernandes, Juliana, Silva, Catia S., Grégio, André, Oliveira, Daniela, Giovanini, Luiz. People Still Care About Facts: Twitter Users Engage More with Factual Discourse than Misinformation–A Comparison Between COVID and General Narratives on Twitter. https://arxiv.org/abs/2012.02164. September 2021.

6. Botacin, Marcus, Moia, Vitor Hugo Galhardo, **Ceschin, Fabricio**, Henriques, Marco Amaral A, Grégio, André. Understanding uses and misuses of similarity hashing functions for malware detection and family clustering in actual scenarios. Forensic Science International: Digital Investigation, 38 , pp. 301220, 2021, ISSN: 2666-2817. https://doi.org/10.1016/j.fsidi.2021.301220. September 2021.

7. Botacin, Marcus; **Ceschin, Fabricio**; Sun, Ruimin; Oliveira, Daniela; Grégio, André. Challenges and pitfalls in malware research. Computers & Security, pp. 102287, 2021, ISSN: 0167-4048. https://doi.org/10.1016/j.cose.2021.102287. July 2021.

8. **Ceschin, Fabrício**, Botacin, Marcus, Lüders, Gabriel, Gomes, Heitor Murilo, Oliveira, Luiz S., Grégio, André. No need to teach new tricks to old malware: Winning an evasion challenge with xor-based adversarial samples. Proceedings of the 3rd Reversing and Offensive-Oriented Trends Symposium, Association for Computing Machinery, Vienna,

Austria, 2020, ISBN: 9781450377751. https://doi.org/10.1145/3433667.3433669. November 2020.

9. Castanhel, Gabriel Ruschel, Heinrich, Tiago, **Ceschin, Fabrício**, Maziero, Carlos. The Impact of Trace Size in Anomaly Detection System for Containers Through Machine Learning. ERRC - WRSeg 2020. https://doi.org/10.5753/errc.2020.15203. November 2020.

10. **Ceschin, Fabrício**, Gomes, Murilo Heitor, Botacin, Marcus, Bifet, Albert, Pfahringer, Bernhard, Oliveira, Luiz S., Grégio, André. Machine Learning (In) Security: A Stream of Problems. https://arxiv.org/abs/2010.16045. October 2020.

11. Lüders, Gabriel, Botacin, Marcus, **Ceschin, Fabrício**, Grégio, André. Breaking Good: Injeção de Payloads Legítimos em Binários Maliciosos para Teste de Robustez de Antivírus contra Evasão. IV Salão de Ferramentas - SBSEG 2020. https://www.doi.org/10.5753/sbseg_estendido.2020.19273. October 2020.

12. Castanhel, Gabriel Ruschel, Heinrich, Tiago, **Ceschin, Fabrício**, Maziero, Carlos. Detecção de Anomalias: Um Estudo Voltado na Identificação de Ataques no Ambiente de Contêiner. XIV Workshop de Trabalhos de Iniciação Científica e de Graduação (WTICG) - SBSEG 2020. https://doi.org/10.5753/sbseg_estendido.2020.19283. October 2020.

13. Botacin, Marcus, **Ceschin, Fabricio**, de Geus, Paulo, Grégio, André. We Need to Talk About AntiViruses: Challenges & Pitfalls of AV Evaluations. Computers & Security, pp. 101859, 2020, ISSN: 0167-4048. https://doi.org/10.1016/j.cose.2020.101859. August 2020.

14. **Ceschin, Fabrício**, Botacin, Marcus, Gomes, Heitor Murilo, Oliveira, Luiz S, Grégio, André. Shallow Security: On the Creation of Adversarial Variants to Evade Machine Learning-Based Malware Detectors. Proceedings of the 3rd Reversing and Offensive-Oriented Trends Symposium, Association for Computing Machinery, Vienna, Austria, 2019, ISBN: 9781450377751. https://doi.org/10.1145/3375894.3375898. November 2019.

15. **Ceschin, Fabrício**, S. Oliveira, Luiz, Grégio, André. Aprendizado de Máquina para Segurança: Algoritmos e Aplicações. Mini Cursos - XIX Simposio Brasileiro de Segurança da Informacão ao e de Sistemas Computacionais - SBSeg 2019. https://doi.org/10.5753/sbc.8589.4. September 2019.

16. Beppler, Tamy, Botacin, Marcus, **Ceschin, Fabrício**, Oliveira, Luiz S, Grégio, André. L(a)ying in (Test)Bed: How Biased Datasets Produce Impractical Results for Actual Malware Families' Classification. Information Security, pp. 381–401, Springer International Publishing, Cham, 2019, ISBN: https://www.doi.org/10.1007/978-3-030-30215-3_19. September 2019.

17. Botacin, Marcus, Galante, Lucas, **Ceschin, Fabricio**, Santos, Luigi Carro Paulo Cesar, de Geus, Paulo Licio, Gregio, Andre, Zanata, Marco. The AV says: Your hardware definitions were updated! 14th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC 2019), IEEE, 2019, ISBN: 978-1-7281-4770-3. https://www.doi.org/10.1109/ReCoSoC48741.2019.9034972. July 2019.

18. **Ceschin, Fabrício**, Pinage, Felipe, Castilho, Marcos, Menotti, David, Oliveira, Luis S, Gregio, André. The need for speed: An analysis of brazilian malware classifiers. IEEE Security Privacy, 16 (6), pp. 31-41, 2018, ISSN: 1540-7993. `https://www.doi.org/10.1109/MSEC.2018.2875369`. November 2018.

19. **Ceschin, Fabrício**, Menotti, David, Castilho, Marcos, Grégio, André. Avaliação da Eficácia de Classificadores de Malware ao Longo do Tempo. Workshop de Forense Computacional - SBSEG 2017. Anais do XVII SBSeg, 2017. p. 1-10. November 2018.

1.3.2 Teaching & Presentations

- 2021 – Presenter, Machine Learning seminar, University of Waikato, New Zealand.
- 2020 – Course Assistant, Data Science for Cybersecurity, Federal University of Paraná.
- 2020 – Presenter, CiDWeek - CiDAMO's I Data Science Week, Federal University of Paraná.
- 2020 – Presenter, "No Need to Teach New Tricks to Old Malware: Winning an Evasion Challenge with XOR-based Adversarial Samples", Reversing and Offensive-Oriented Trends Symposium (ROOTS).
- 2019 – Presenter, "Shallow Security: On the Creation of Adversarial Variants to Evade Machine Learning-Based Malware Detectors", Reversing and Offensive-Oriented Trends Symposium (ROOTS).
- 2019 – Course Assistant, Data Science Course, Informatics Academic Week, Federal University of Paraná.
- 2019 – Presenter, Machine Learning applied to Cybersecurity Course, Brazilian Security Symposium (SBSEG).
- 2017 – Monitor, Undergraduate Algorithm Class, Federal University of Paraná.

1.3.3 Awards

- Sep 2021 – Machine Learning Security Evasion Competition (MLSEC) 2021, Attacker's Challenge – 1st Place
- Sep 2021 – Machine Learning Security Evasion Competition (MLSEC) 2021, Defender's Challenge – 1st Place
- Sep 2020 – Machine Learning Security Evasion Competition (MLSEC) 2020, Attacker's Challenge – 1st Place
- Sep 2020 – Machine Learning Security Evasion Competition (MLSEC) 2020, Defender's Challenge – 2nd Place
- Sep 2019 – Machine Learning Security Evasion Competition (MLSEC) 2019 – 2nd Place

1.3.4 Grants

- Oct 2022 – Conference on Applied Machine Learning in Information Security (CAMLIS) 2022 – Student Scholarship
- May 2019 – Secure and Private AI Scholarship from Facebook AI
- Jan 2019 – USENIX Enigma 2019 - Diversity Grant
- Aug 2017 – Google Research Awards for Latin America

1.3.5 Projects

- **Fast & Furious: Malware Detection Data Stream Datasets.** These datasets (DREBIN and AndroZoo) are different from their original versions. The original DREBIN dataset does not contain the samples' timestamps, which I collected using VirusTotal API. My version of the AndroZoo dataset is a subset of reports from their dataset previously available in their APK Analysis API, which was discontinued. With samples' timestamps, we can simulate real world data streams. `https://www.kaggle.com/datasets/fabriciojoc/fast-furious-malware-data-stream`.
- **Scikit-Multiflow.** Scikit-Multiflow is an open-source machine learning package for streaming data, which I was able to contribute during my Ph.D. `https://scikit-multiflow.github.io/`.
- **Corvus_.** Corvus_ is a public malware analysis system, basically the result of multiple developments created by my research group (SECRET). `https://corvus.inf.ufpr.br`.
- **Machine Learning applied to Cyber Security Course.** The course I presented in SBSEG 2019, with all the source codes used. `https://github.com/fabriciojoc/ml-cybersecuritiy-course`.
- **Brazilian Malware Dataset.** The Brazilian Malware Dataset I developed during my masters. `https://github.com/fabriciojoc/brazilian-malware-dataset`.

1.3.6 Visits

- Feb – Mar 2020 – The University of Waikato, School of Computing and Mathematical Sciences. Visitor Ph.D. Student. Hamilton, Waikato, New Zealand. Collaborative research in machine learning applied to cybersecurity, data streams, and development of algorithms for scikit-multiflow library. Hosted by Prof. Heitor Murilo Gomes, Albert Bifet, and Bernhard Pfahringer.
- May 2019 and Aug 2018 – University of Florida, Department of Electrical and Computer Engineering. Visitor Ph.D. Student. Gainesville, Florida, United States. Collaborative research in machine learning applied to cybersecurity. Hosted by Prof. Daniela Oliveira.

1.3.7 Service

- 2017, 2019, 2020 – Reviewer for Brazilian Security Symposium (SBSeg) Conference.
- 2018 – Reviewer for Annual Computer Security Applications Conference (ACSAC).
- 2019 – Reviewer for Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA).
- 2019 – Reviewer for IEEE SecDev Conference.
- 2019 – Reviewer for IEEE Security & Privacy Magazine.
- 2020 – Reviewer for Australasian Conference on Information Security and Privacy (ACISP).
- 2020 – Reviewer for The European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML).
- 2021 – Reviewer for Expert Systems With Applications (ESWA) Journal.
- 2021 – Reviewer for Frontiers in Big Data Journal.
- 2021 – Reviewer for The International Conference on Availability, Reliability and Security (ARES).
- 2021 – Reviewer for USENIX Security 2021 Artifact Evaluation.
- 2022 – Reviewer for USENIX Security & Privacy 2022.
- 2022 – Reviewer for Australasian Data Mining Conference (AUSDM) 2022.

1.4 OUTLINE

This thesis is organized as follows: in Chapter 2 the background is presented, reviewing the literature to understand what the challenges related to each step of ML in security are; in Chapter 3 I show some adversarial machine learning attacks for ML-based malware detectors; the effects of concept drift in two security domains (computer usage profiles and malware detection) are also shown in Chapter 4; in Chapter 5 I present the problems related to the evaluation of ML-based solutions and a framework capable of evaluating any data stream considering delays in their labels, even when they may never be available with a given probability; in Chapter 6, I discuss some of the key aspects I found during the development of my research; finally, in Chapter 7, I conclude this thesis.

2 BACKGROUND

In our preliminary analysis of the literature, we identified that many of the existing cybersecurity researchers are not engaged in using Machine Learning to solve actual security problems, i.e., there is a lack of focus on “machine learning that matters” (Wagstaff, 2012). Hence, to discuss the main issues of ML applied to cybersecurity, I will delve into the more relevant aspects of ML-based solution modelling – data collection, attribute extraction, feature extraction, train/update model, and evaluation. Although the separation of those topics results in a more legible text, all related work is spread within each of their related sections, in our paper "Machine Learning (In) Security: A Stream of Problems".

2.1 MACHINE LEARNING (IN) SECURITY: A STREAM OF PROBLEMS

This paper was submitted for publication to the ACM Digital Threats: Research and Practice (DTRAP) journal.

Fabrício Ceschin[1], Heitor Murilo Gomes[2], Marcus Botacin[1], Albert Bifet[3], Bernhard Pfahringer[3], Luiz S. Oliveira[1], André Grégio[1]
[1]Federal University of Paraná, Brazil
{fjoceschin, mfbotacin, lesoliveira, gregio}@inf.ufpr.br
[2]Victoria University of Wellington, Wellington, New Zealand
heitor.gomes@vuw.ac.nz
[3]University of Waikato, Hamilton, New Zealand
{abifet, bernhard}@waikato.ac.nz

2.1.1 Abstract

Machine Learning has been widely applied to cybersecurity and is currently considered state-of-the-art for solving many of the open issues in that field. However, it is very difficult to evaluate how good the produced solutions are, since the challenges faced in security may not appear in other areas (at least not in the same way). One of these challenges is the concept drift, which increases the existing arms race between attackers and defenders: malicious actors can always create novel, evolved threats to overcome the defense solutions at hand, and these "evolution steps" are often not considered in many security approaches. Due to this type of issue, it is fundamental to know how to properly build and evaluate an ML-based security solution. In this paper, we identify, detail, and critically discuss the main challenges in the correct application of ML techniques to cybersecurity data. We evaluate how concept drift, concept evolution, delayed labels, and adversarial machine learning impact the existing solutions. Moreover, we address how issues related to data collection (e.g., inconsistency, labeling, and imbalance) affect the quality of the results presented in the security literature. Finally, we show how existing solutions may fail under certain circumstances, and propose possible solutions to fix them when appropriate.

2.1.2 Introduction

The massive amount of data produced daily demands automated solutions capable of keeping Machine Learning (ML) models updated and working properly, even with all emerging threats that constantly try to evade these models. This arms race between attackers and defenders moves the cybersecurity research forward: malicious actors are continuously creating new variants of attacks, exploring new vulnerabilities, and crafting adversarial samples, whereas security analysts are trying to counter those threats and improve detection models. For instance, 68% of phishing emails blocked by GMail are different from day to day (Bursztein and Oliveira, 2019), requiring Google to update and adapt its security components regularly. Thus, applying ML on cybersecurity is a challenging endeavour. One of the main challenges is the volatility of the data used for building models as attackers constantly develop adversarial samples to avoid detection. This leads to a situation where the models need to be constantly updated to keep track of new attacks. Another challenge is related to the application of fully supervised methods, since the class labels tend to depict an extreme imbalance, i.e. dozens of attacks diluted in thousands normal examples. Labeling such instances is also problematic as it requires domain knowledge

and it can detain the learning method, i.e. the analyst labeling the data is a bottleneck in the learning process. This motivates the development of semi-supervised and anomaly detection methods (Salehi and Rashidi, 2018).

To improve the process of continuously updating an ML cybersecurity solution, the adoption of stream learning (also incremental learning or online learning) algorithms are recommended so they can operate in real-time using a reasonable amount of resources, considering that we have limited time and memory to process each new sample incrementally, predict samples at any time, and adapt to changes (Bifet et al., 2018). However, the majority of works in the literature do not consider these challenges when proposing a solution, which makes them not feasible in the reality. Thus, many cybersecurity researches are not related to real-world problems and are not focused on "machine learning that matters" (Wagstaff, 2012).

Some previous work reported the relevance of some of these problems and provided research directions. Rieck et al. (Rieck, 2011) stated that only few research has produced practical results, presenting directions and perspectives of how to successfully link cybersecurity and Machine Learning and aiming at fostering research on intelligent security methods, based on a cyclic process that starts discovering new threats, followed by their analysis and the development of prevention measures. Jiang et al. systematically studied some publications that applied ML in security domains, providing a taxonomy on ML paradigms and their applications in cybersecurity (Jiang et al., 2016). Salehi et al. categorized existing strategies to detect anomalies in evolving data using unsupervised approaches, since label information is mostly unavailable in real-world applications (Salehi and Rashidi, 2018). Papernot et al. systematized findings on ML security and privacy, focusing on attacks identified on ML systems and defense solutions, creating a threat model for ML, and categorizing attacks and defenses within an adversarial framework (Papernot et al., 2018b). Maiorca et al. explored adversarial attacks against PDF (Portable Document Format) malware detectors, highlighting how the arms race between attackers and defenders has evolved over the last decade (Maiorca et al., 2019). Gomes et al. highlighted the state-of-the-art of Machine Learning for data streams, presenting possible research opportunities (Gomes et al., 2019). Arnaldo et al. described a set of challenges faced when developing a real cybersecurity ML platform, stating that many researches are not valid in many use cases, with a special focus on label acquisition and model deployment (Arnaldo and Veeramachaneni, 2019). Kaur et al. presented a comparative analysis of some approaches used to deal with imbalanced data (pre-processing methods, algorithmic centered approaches, and hybrid ones), applying them to different data distributions and application areas (Kaur et al., 2019). Gibert et al. listed a set of methods and features used in a traditional ML workflow for malware detection and classification in the literature with emphasis on deep learning approaches, exploring some of their limitations and challenges, such as class imbalance, open benchmarks, concept drift, adversarial learning and interpretability of the models (Gibert et al., 2020). Finally, Arp et al. conducted a study of 30 papers from top-tier security conferences within the past 10 years and showed that many pitfalls are widespread in the current security literature (Arp et al., 2022).

In this work, we present a broad collection of gaps, pitfalls, and challenges that are still a problem in many scenarios of ML solutions applied in cybersecurity, which may overlap with other areas, suggesting, in some cases, possible mitigation for them. As a study case, in most cases we focus in malware detection or classification tasks given that they may contain all the problems listed. We want to acknowledge that we are not pointing fingers at anyone, given that our own work is subject to many problems stated here. Our main contribution is to point directions to future cybersecurty researches that make use of ML, aiming to improve their quality to be used in real applications.

In Figure 2.1, we show a scheme of the process to develop and evaluate ML solutions (both supervised and unsupervised methods) for cybersecurity based on the literature, which is similar to any pattern classification task. It consists of two phases: train, i.e., training a model with the data available at a given time (for supervised methods, using the available labels and features to create decision boundaries, for unsupervised, using only the features to create clusters) and test, i.e., testing it considering the new data collected (for both supervised and unsupervised, using their labels – to check if they were correctly predicted – and features to update decision boundaries and clusters). Note that we defined two steps after the data collection, given that, in many cybersecurity tasks, the raw data collected needs to have their metadata (attributes) extracted before actually being used by the ML model as features (attributes processed by a feature extractor). We understand that these steps may overlap in some cases, but for a more fine-grain discussion, we analyzed them separately. Also, it is important to notice that security data, the input of the process, is available from a data stream that is in constant production, and the model and its metrics, the output of the process, are produced during the execution of the scheme.

Thus, this paper is organized as follows: first, in Section 2.1.3, we discuss how to identify the correct ML task for each cybersecurity problem; Further, it is organized according to each step of the process to make this work easier to follow, with data collection (obtaining data for the ML solution), in Section 2.1.4, attribute extraction (extracting metadata from the data previously obtained) in Section 2.1.5, feature extraction (extracting features from the attributes collected) in Section 2.1.6, model (training and updating the model using the features extracted) in Section 2.1.7, and evaluation (evaluating the proposed solution) in Section 2.1.8. In Section 2.1.9 we discuss how ML applications should be understood, their limitations, and existing open gaps. Finally, we conclude our work in Section 2.1.10.

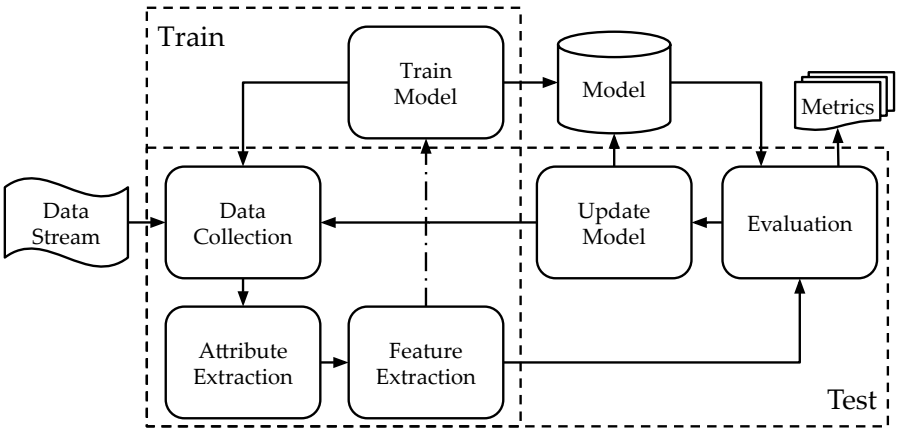


Figure 2.1: **Scheme to develop and evaluate Machine Learning solutions for cybersecurity.** Each step is executed in sequence, considering that (i) the model is first trained and then (ii) tested/evaluated and updated (if needed).

2.1.3 Security Tasks and Machine Learning

ML algorithms have been applied to multiple security contexts, as summarized in Table 2.1. Most detection tasks largely rely on classification algorithms to solve binary problems, i.e., trained to identify "benign" and "malicious" samples (e.g., malware vs. goodware, spam vs. ham, attack vs. non-attack, etc.). Anomaly-based intrusion detection solutions, however, mostly operate by detecting a deviation in comparison to a baseline (e.g., a host that starts to frequently access/be accessed by another unknown host regarding the typical network activity) often using

outlier detection algorithms, but also relying on alternative approaches (Liao and Vemuri, 2002). The usual choice for attacks that need to be either classified **and** detected is to apply both classification and clustering algorithms. These are the cases when malware samples (that were already classified and checked) need to be assigned to a known family. In this case, the unknown sample is compared to known samples in previously generated clusters and then assigned to the most similar one. However, this strategy is prone to failure if a motivated attacker generates adversarial samples that, although exhibit different features than those from which they are derived, maintain the same original malicious behavior. Therefore, these adversarial samples would belong to a certain family, but end up being (mis)classified as from another one (Ceschin et al., 2020a).

A procedure similar to clustering is employed to authorship attribution in forensic procedures (Rocha et al., 2017). In addition to the complex attribution procedures, the cybersecurity field also has space for less complex but faster ML algorithms via their application to the security assessment procedure's triage steps. Risk assessment procedures often operate leveraging linear regression to extrapolate previous events to the future. Malware analysts often verify malware strings to have an initial hypothesis about malware operation. The most relevant strings can be identified via learn-to-rank (LTR) algorithms. In this work, we cover the challenges associated to all these tasks.

Table 2.1: **Applications of ML to cybersecurity**. Distinct approaches are applied according to the specific security need.

Security Task	Specialized Task	ML Task
Detection	Malware Detection	Classification (Arp et al., 2014)
	Intrusion Detection	Outlier Detection (Kumar and Mathur, 2014)
	Spam Filtering	Classification (Panigrahi, 2012)
Analysis & Attribution	Malware Labelling	Clustering (Sebastián et al., 2016)
	Log Analysis	Clustering (El Hadj et al., 2018)
Triage	Malware Analysis	Reinforcement Learning/Ranking (FireEye, 2019)
	Risk Assessment	Regression (Gritzalis et al., 2018)
Forensics	Attribution	Clustering (Rocha et al., 2017)
	Object Recognition	Dimensionality Reduction (Keser and Töreyin, 2019)

Identifying the correct ML task associated to the security problem to be solved is essential to properly plan and evaluate experiments, as well as to understand the limitations of the proposed solutions. However, this identification is far from straightforward, and requires reasoning about multiple corner conditions. For instance, for the malware detection case, it is usual to find in the literature multiple solutions proposing a ML-based engine to be applied by anti viruses (AVs). Most of these solutions are initially modeled as a classification problem (goodware vs. malware), which suffices for detection. In practice, AVs provides more than a detection label (Botacin et al., 2020b), they also attribute the malware sample to a family (e.g., ransomware, banker, so on) to present a family label, which is essential to allow incident response procedures. Therefore, a ML engine for an AV should also be modeled as a family attribution problem, which requires the application of both classifying and clustering algorithms.

2.1.4 Data Collection Challenges

The quality, quantity, and distribution of data inputted to a Machine Learning algorithm (dataset) are the basis of an adequate learning process, since ML algorithms rely on the samples presented to them in the training step and the resulting model will allow further decision making. Thus, data collection, which comprises acquisition, enrichment/augmentation, and labeling (Roh et al., 2019; Gron, 2017), may be one of the most challenging steps of a ML project. Besides real-world problems take these steps into consideration (Gomes et al., 2019), certain research works might miss some of them. This might happen either due to the format of the dataset used or external reasons, such as privacy requirements about the data (might need differential privacy (Dwork and Roth, 2014)), local laws that prevent the distribution of potentially harmful pieces of code, ease of accomplishing reproducibility of the experiments, and so on. Considering the steps performed in ML-based cybersecurity systems (Saxe and Sanders, 2018) and to provide a straightforward discussion, we assume that data can be available in three formats, which may overlap according to the task at hand or the chosen approach:

- **Raw Data:** data available in the same way they were collected, usually used to analyse their original behavior. For example: PE executables (Ceschin et al., 2018), ELF (VirusShare, 2019), APK packages (Allix et al., 2016a), or network traffic data captured in PCAP format (NetResec, 2020; Pang et al., 2005; Hick et al., 2007; Shiravi et al., 2012). Comparing to computer vision, the raw data are the images collected to create an image recognition model (Dalal and Triggs, 2005). In cybersecurity, AndroZoo (Allix et al., 2016a) provides many APK packages collected from several sources that could be used for malware detection systems or forensics analysis;

- **Attributes:** filtered metadata extracted from the raw data with less noise and focus on the data that matters, being very suited for ML. For example: CSV with metadata, execution logs of software or data extracted from its header (Ceschin et al., 2018; Anderson and Roth, 2018), or a summary of information from a subset of network traffic data. In computer vision, the attributes would be the gradient images extracted from the original ones (Dalal and Triggs, 2005). EMBER (Anderson and Roth, 2018) is a good example of a dataset containing attributes, available in JSON, extracted statically from raw data (Windows PE files). DREBIN (Arp et al., 2014) is another example containing static attributes extracted from APK packages.

- **Features:** features extracted from the attributes or raw data that distinguish samples, ready to be used in a classifier. For example the transformation of logs into feature vectors for every software mentioned in the previous item, whose positions of this vector correspond to the features (Ceschin et al., 2018), or a transformation of traffic data into features containing the frequency of pre-determined attributes (i.e., protocols, amount of packages, bytes, etc.). In an image recognition problem, the features would be the histogram of gradients extracted using the gradient images created before (Dalal and Triggs, 2005). This type of data is usually used by researchers to make their experiments faster, given that they extract the features once and can share them to use as input for their models.

The data collection or even any of the dataset formats above are susceptible to problems that may affect the whole process of using ML in any cybersecurity scenario and are frequently seen in the literature, such as data leakage (or temporal inconsistency), data labeling, class imbalance, and anchor bias.

2.1.4.1 Data Leakage (Temporal Inconsistency)

To evaluate a ML system, it is commont to split the dataset into at least two sets: one to train the model and another to test it. The k-fold cross-validation is used to create k partitions and evaluate a given model k times, using one of the partitions as test set and the remaining as training set, taking the average of the metrics as a final result (Michie et al., 1994). This is a common practice for many ML experiments, such as image classification, where temporal information may not be important and are used in batches. However, when considering cybersecurity data, which are obtained from a stream, it is not real to consider data from different or mixed epochs to train and test a model (known as data leakage (Kaufman et al., 2011), data snooping (Arp et al., 2022), or temporal inconsistency), given that it could increase the detection accuracy because the model knows how future threats are (i.e., the model is exposed to test data when it was trained) (Ceschin et al., 2018). For instance, consider a malware detector that works similar to an antivirus, i.e., given a software, our model wants to know if it is a malware or not to block an unwanted behaviour. To create this model we train it using a set of malicious and benign software that are known and were seen in the past. Then, this model is used to detect if files seen in the future are malign, even if they perform totally different attacks than before. These new threats will just be used to update the model after they are known and labeled, which will probably increase the coverage of different unknown attacks, detecting more malware than before.

Due to this temporal inconsistency problem, it is important to collect the timestamp of the samples during the data collection and it is something not considered by some authors in cybersecurity. For instance, the DREBIN dataset (Arp et al., 2014) (malware detection) makes all the malicious APKs available and also their attributes, but does not include the timestamp that they were found in the wild, making all the research that use this dataset exposed to data leakage. We understand that sometimes it is difficult to set a specific release date for a sample, but they are needed to avoid this problem. For malware detection, for example, researchers usually use the date when they were first seen in VirusTotal as a timestamp (Ceschin et al., 2018; Pendlebury et al., 2019; Kantchelian et al., 2015; Total, 2019).

2.1.4.2 Data Labeling

Labeling artifacts is essential to training and evaluating models. However, having artifacts properly labeled is as hard as collecting the malicious artifact themselves, as previously discussed. In many cases, researchers leverage datasets of crawled artifacts and make a strong hypothesis about them, such as “all samples collected by crawling a blacklist are malicious“, or “all samples collected from an App Store are benign“. These hypotheses are strong because previous work has demonstrated that even samples downloaded from known sources might be trojanized with malware (Botacin et al., 2020a). Therefore, a model trained based on this assumption would make the model also learn some malicious behaviors as legitimate. Also, some labels may be inaccurate, unstable, or erroneous, which may affect the overall classification performance of ML-based solutions (Arp et al., 2022).

A common practice to obtain labels and mitigate the aforementioned problems is to rely on the labels assigned by AVs. A common approach is to use the VirusTotal service (Total, 2019), which provides detection results based on more than sixty antivirus engines available on the platform. Unfortunately, AV labels are not uniform (Botacin et al., 2020b), with each vendor reporting a distinct label. Therefore, researchers have either to select a specific AV to label their samples (according to their established criteria) or adopt a committee-based approach to unify the labels. For Windows malware, AVClass (Sebastián et al., 2016) is widely used for this purpose and, for Android malware, Euphony (Hurier et al., 2017) is used. Both of these techniques were

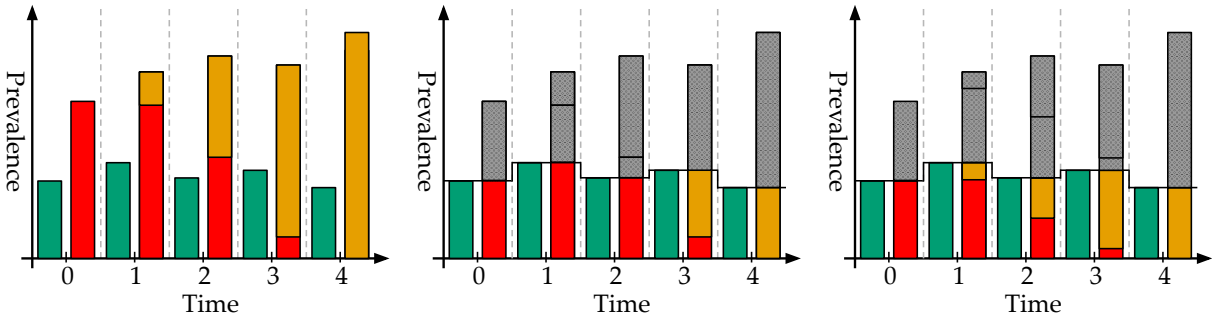
evaluated by the authors using a high number of malware (8.9 million in AVClass and more than 400 thousand samples in Euphony), obtaining a significant F-measure score (bigger than 90% in both cases) and generating consistent results to create real datasets such as AndroZoo (Allix et al., 2016a), which uses Euphony (Hurier et al., 2017) to generate malware family labels. Although AVs can mitigate the labeling problem, their use should consider the drawbacks of AV internals. AVs provide two labels: detection and family attribution. Both the detection label as well as the family attribution label change very often over time: newly-released samples are often assigned a generic and/or heuristic label at the initial time and this is further evolved as new, more specific signatures are added to the AV to detect this threat class. Therefore, the date on which the samples are labeled might significantly affect the ML results. Recent research shows that AVs label might not establish before the 20 or 30 days after a new threat is released (Botacin et al., 2020b). To mitigate this problem, delayed evaluation approaches should be considered, as shown in Section 2.1.8.

2.1.4.3 Class Imbalance

Class imbalance is a problem in which the data distribution between the classes of a dataset differs relatively by a substantial margin, and it is usually present in many research works (Kaur et al., 2019). If we consider the Android landscape, the AndroZoo dataset contains only about 18% of malicious apps (the remaining apps are considered benign (Pendlebury et al., 2019; Allix et al., 2016a)). This makes the Android malware detection problem more challenging due to the presence of imbalanced data. There are several methods in the literature whose aim is to overcome this problem by making use of pre-processing techniques, improving the learning process (cost-sensitive learning), or using ensemble learning methods (Kaur et al., 2019; Gomes et al., 2019). The two latter methods will be discussed in Section 2.1.7, since they are part of the process of training/updating the model. The former method (pre-processing) relies on resampling, i.e., removing instances from the majority classes (undersampling) or creating synthetic instances for the minority classes (oversampling) (Gomes et al., 2019).

In the context of cybersecurity, undersampling may affect the dataset representation, given that removing some samples from a certain class can affect its detection. For instance, considering malware detection, removing malware samples may reduce the detection of some malware families (the ones that had samples removed) and also make their prevalence in a given time (monthly or weekly) less important than reality when creating a dataset, possibly not capturing a concept drift or sub-classes of the majority classes. Figure 2.2 presents a hypothetical dataset distribution with two classes (green and red/orange), one of the classes has two concepts or two sub-classes (red and orange) as time goes by, i.e., consider it as being a dataset with normal and malicious behavior, the malicious one behavior that evolves or has sub-classes of behaviors as time goes by, something that is known to happen in real-world cybersecurity datasets (Ceschin et al., 2018). In Figure 2.2(a) we can see the original distribution of the dataset, with no technique applied. In Figure 2.2(b) we can see the dataset distribution when removing samples (in gray) to keep the same distribution by class, which results in a different scenario than the reality since the orange concept or sub-class is not seen in some periods. In Figure 2.2(c) we see the ideal scenario for undersampling when the proportion of concepts or sub-classes is the same for the same period while keeping the same number of samples for both classes (this may be a mitigation for the undersampling problem, but even with this solution important samples may be discarded).

An example of oversampling technique is SMOTE, which consists in selecting samples that are close in the feature space, drawing a line between them, and generating new samples along with it (Chawla et al., 2002). Although such techniques are interesting, they may generate results that produce data leakage if they do not consider the time when creating synthetic samples.

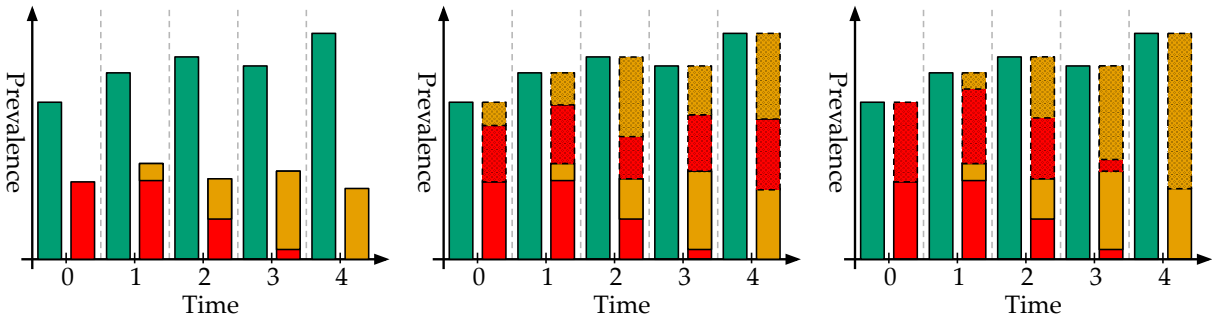


(a) **No undersampling.** Dataset original distribution without undersampling.

(b) **Undersampling with no temporal information.** Undersampling dataset distribution when simply removing samples to keep the same distribution by class.

(c) **Undersampling with temporal information.** Undersampling dataset distribution when keeping the same proportion of sub-classes of a given class.

Figure 2.2: **Undersampling examples in a dataset.** Samples in green and red/orange represent different classes. Red and orange colors represent different sub-classes or concepts of the same class. Gray color with circles represents ignored/removed instances.



(a) **No oversampling.** Dataset original distribution without oversampling.

(b) **Oversampling with no temporal information.** Oversampling dataset distribution without considering temporal information.

(c) **Oversampling with temporal information.** Oversampling dataset distribution considering temporal information.

Figure 2.3: **Oversampling examples in a dataset.** Samples in green and red/orange represent different classes. Red and orange colors represent different sub-classes or concepts of the same class. Samples with dashed line and circles are the synthetic instances created by oversampling.

For instance, consider Figure 2.3, where we present again a hypothetical dataset distribution with two classes, with the same classes and problems as Figure 2.2. In the first case, in Figure 2.3(a), we see the original dataset distribution, where the class red/orange is the minority one. In Figure 2.3(b) the problem of data leakage is shown: the artificial data generated (with dashed lines and circles) is based on all the dataset, without considering any temporal information. Thus, we can see the orange concept/sub-class at time 0 in the synthetic instances, which does not represent the real distribution of this class at that time (this problem happens all the time in this case, with the red concept/sub-class being shown even at time 4). In contrast, in Figure 2.3(c) we can see an oversampling technique that considers the temporal information, generating synthetic data that correspond to the actual concept/sub-class of a given time, resulting in the same distribution of concepts/sub-classes as the original data. Despite also being an interesting approach, for the cybersecurity context, oversampling works at the feature level, which means that it may not generate synthetic raw data. For instance, if we consider these data as being applications, oversampling will only create synthetic feature vectors and not real applications that work.

2.1.4.4 Dataset Size Definition

A major challenge in creating a dataset is to define its size. Although it is a problem that affects all ML domains, it has particularly severe implications in cybersecurity. On the one hand, a small dataset may not be representative of a real scenario, which invalidates the results of possible experiments using it. Also, some models may not generalize enough and not work as intended, presenting bad classification performance (Gron, 2017). Limited evaluations are often seen in the cybersecurity context because collecting real malicious artifacts is a hard task, as most organizations do not share the threats that affect them to not reveal their vulnerabilities. On the other hand, a big dataset may result in long training time, and produce too complex models and decision boundaries (according to the classifier and parameters used) that are not feasible in the reality (e.g., real-time models for resource-constrained devices), such as some deep learning models that usually requires a large amount of data to achieve good results (Jiang et al., 2016). As an analogy, consider that a dataset is a map and, for instance, represents a city. It is almost impossible that this map has the same scale as the city, however, it can be useful by representing what is needed for a given purpose. For example, a map of public transportation routes is enough if our objective is to use it, but if we need to visit touristic places, this very same map will not be useful, i.e., new data or a new map is required (Steele, 2006). These circumstances reflect the ideas of both George Box and Alfred Korzybski. Box said that "essentially, all models are wrong, but some are useful" (Shoesmith et al., 1987), i.e., a model built using a dataset might be useful for a given task, but it will not be perfect – there will be errors, we just need to know how wrong they have to be – and it will not be good for other tasks, which makes necessary to collect more (or new) data and select new parameters for a new model. Korzybski mentioned that "the map is not the territory" (Korzybski, 1931), meaning that it can be seen as a symbol, index, or representation of it, but it is not the place itself and it is prone to errors. In our case, a model or a dataset can represent a real scenario, but it is just a representation of it and may contain errors. These errors may present results that do not reflect reality. For instance, considering a malware detection task that uses grayscale images as representation for windows binaries, when using a dataset that represents the real scenario with no data filtering, i.e., without removing specific malware families, the accuracy ($\approx 35\%$) was much worse than other scenarios where the authors filter the number of malware families, reducing the complexity of the problem and achieving almost 90% of accuracy, as shown in Figure 2.4 (Beppler et al., 2019).

Another point to consider when building a cybersecurity solution is that they may have regional characteristics that are reflected in the dataset and, as a consequence, in the model and

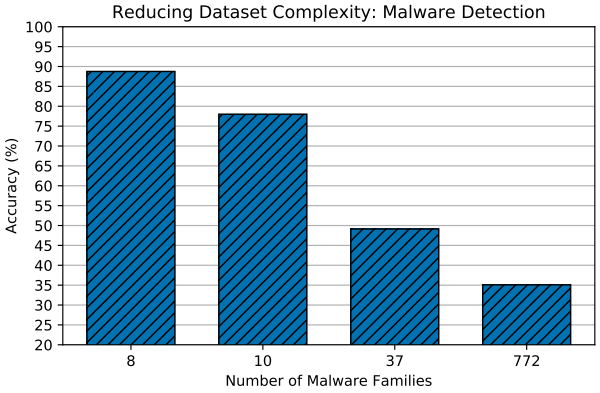


Figure 2.4: **Reducing dataset complexity.** The more the dataset is filtered, the bigger the accuracy achieved, which means that researchers must avoid filtered datasets to not produce misleading results (Beppler et al., 2019).

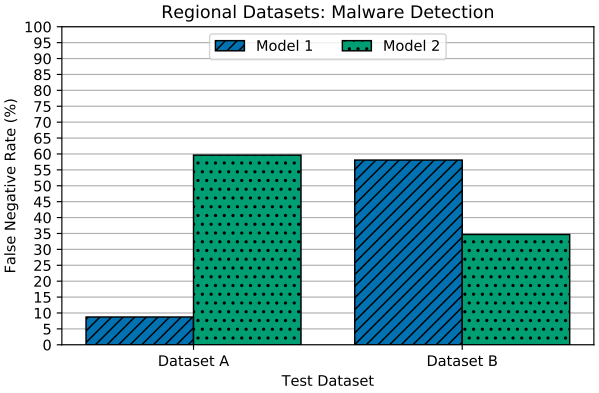
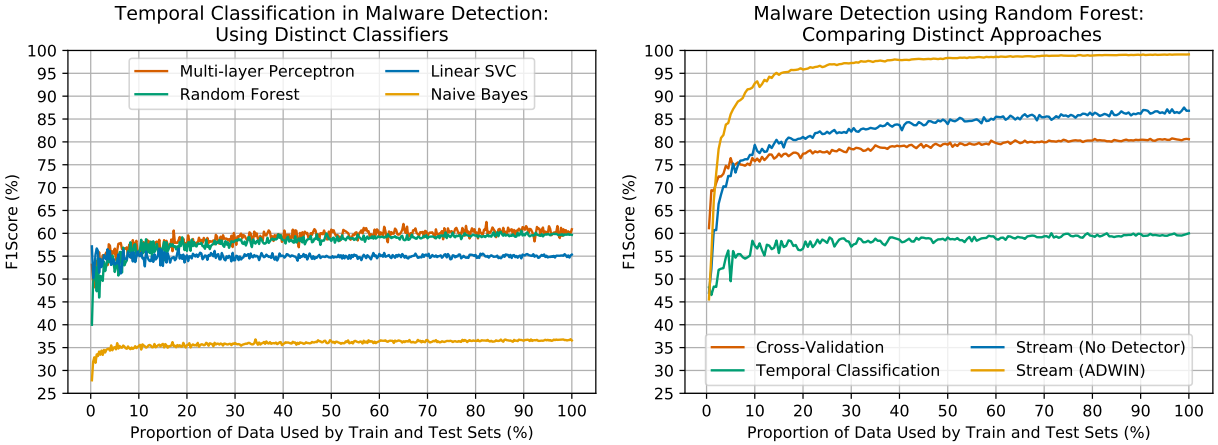


Figure 2.5: **Regional datasets.** Models may have a bias towards the scenario where the dataset used to train them was collected, indicating that they need to be specially crafted in some cases (Ceschin et al., 2020a).

its predictions (Ceschin et al., 2020a), which is related to the generalization problem in machine learning. For instance, considering a malware detection task with two models that are based on classification trees: Model 1 (random forest with PE metadata (Ceschin et al., 2018)) is trained using data from region A (BRMalware dataset (Ceschin et al., 2018)) and Model 2 (LightGBM with PE metadata (Anderson and Roth, 2018)) is trained with data from region B (EMBER dataset (Anderson and Roth, 2018)). Ceschin et al. showed that when testing both models with test data from both regions (test dataset A, from region A, and test dataset B, from region B), they perform better in their respective regions and present a much higher false negative rate (FNR) in the opposite region (allowing malware to be executed if this solution is applied in a different region that it was designed for, for example), as shown in Figure 2.5. Thus, it is important to consider collecting new data when implanting a known solution to a new target scenario (or region).

To elaborate our discussion about dataset size definition, we created two experiments to better understand how much data we need to achieve representative results using a subset of AndroZoo dataset (Ceschin et al., 2022; Allix et al., 2016a) for malware detection, composed of 347, 444 samples (267, 342 benign and 80, 102 malicious applications) spanning from 2016 to 2018. Both experiments consist in understanding how much data we need to achieve a stable classification performance based on the dataset proportion used to train our models. To do so, we divided the dataset by months and reduced the proportion of goodware and malware in both the training and test set (the first half of the dataset, ordered by their first seen date in virus total, is used to train and the second, to test). Thus, in the first experiment, we tested different classifiers using temporal classification (using the training set with “known data” to train the models and ”future data“ to test them), with different proportions of data in the training and test dataset, to see how each one of them was going to perform. Surprisingly, all classifiers (Multi-layer Perceptron, Linear SVC, Random Forest, and Naive Bayes (Pedregosa et al., 2011)) had similar behaviors, also presenting similar curves as consequence and reaching an “optimal” classification performance by using only around 10% to 20% of the original dataset, with multi-layer perceptron and random forest achieving the best overall results, as shown in Figure 2.6(a). It is clear to see that, after these proportions, the f1score was almost stable, improving just a little even when increasing a lot the amount of data used. Furthermore, in the second experiment, we compared different approaches in a unique classifier (Random Forest) to see if they present the same behavior as the first one. To do so, we used only random forest in four different approaches: cross-validation (randomly selecting train and test samples, with data leakage), temporal classification (the same



(a) **Comparing classifiers.** "Optimal" classification performance is achieved by using only around 10% to 20% of the original dataset in all cases.

(b) **Comparing approaches.** All of them stabilize their f1score after 30% to 40% proportions, less than half of the original dataset.

Figure 2.6: **Dataset size definition in terms of f1score.** In both experiments a similar behaviour is seen, with a certain stability in classification performance after using only a given proportion of the original dataset.

approach as the first experiment), and stream with and without drift detector (initializing a stream version of the random forest (Gomes et al., 2017b) – known as adaptive random forest – with the training data and incrementally updating it with testing data, using the ADWIN drift detector (Bifet and Gavaldà, 2007) to detect changes in data distribution when it is enabled). As we can see in Figure 2.6(b), all the approaches also presented a similar behavior and curves, but this time, all of them started to stabilize their f1score after 30% to 40% proportions, which means that they just need at least half of the dataset to present almost the same result as using the entire dataset. Also, it is interesting to note that stream approaches presented better results than cross-validation, which produces data leakage. We believe that this result is due to the fact that the standard test-than-train approaches already have samples' labels right after they are collected, using them to update the model, which is something that does not happen in the real world due to the delays in labeling artifacts, as we show in Section 2.1.8.3. With both experiments, we conclude that, at some point, it may be more important to look for new features or representation strategies than to add more data to the training set, given that the classification performance is not improved so much according to our experiments.

The implication of these findings for cybersecurity is that more observational studies – research that focuses on analyzing ecosystem landscape, platforms, or specific types of attacks to inform the development of future solutions – are required so to allow for the creation of useful datasets and ML solutions. Due to the parsimonious number of this kind of study, one might fall for the Anchor bias (Epley and Gilovich, 2006), a cognitive bias where an individual relies too heavily on an initial piece of information offered (the "anchor") during decision-making to make subsequent judgments. When the value of the anchor is set, future decisions are made using it as a baseline. For instance, if we consider research that uses a one million samples dataset, it is going to become the anchor for future research, even if this dataset is not consistent with the real scenario. Thus, choosing or creating a dataset is not about its size, but its representation of the real world for the task being performed, as a map. In addition, concept drift and evolution are the nature of cybersecurity datasets, which makes it necessary to collect samples from different epochs to correctly evaluate any solution (Ceschin et al., 2018; Gibert et al., 2020). We acknowledge here that some approaches need more data to achieve better results, such as deep learning techniques (Underwood, 2019), and it may be a limitation for them. A good example of building a real dataset is the one built by Pendlebury et al. for malware detection, where the

proportion of samples found in the dataset is the same found in the wild by AndroZoo collection of Android applications (Pendlebury et al., 2019; Allix et al., 2016a).

2.1.5 Attribute Extraction

Extracting attributes, i.e., selecting filtered metadata collected from the raw data, is a key step to create useful features for the ML models. In this section, we pinpoint the impact of different attributes in ML solutions. Note that we have separate definitions for attributes and features (see Section 2.1.4): the former is filtered metadata extracted from the raw data obtained in the data collection step and strictly related to the security task being performed, whereas the latter is the attributes transformation into a distinct set of samples representation ready to serve as input to a model. They usually cannot be directly used by a ML model, given that they need to be preprocessed and/or transformed into features (generally, numerical) to be used as input of a classifier.

2.1.5.1 The Impact of Different Attributes

Different approaches for attribute extraction impose varied costs, and might also lead to distinct ML outcomes. Naturally, executing an artifact to extract feature costs more than statically inspecting it, but the precision of the classification approach might get higher if this data is properly used. Therefore, the selection of the attribute extraction procedure should consider its effect on the outcome. Galante et al. (Galante et al., 2019) show the effect of attribute extraction procedures over three distinct pairs of models to detect Linux malware (SVM with RBF kernel, Multi-layer Perceptron, and Random Forest). These three pairs of models consider the same set of features, one of them considering statical attributes and the other, dynamic attributes. For instance, in statically extracted attributes, the authors consider the presence of the `fork` system call in the import table of the sample's binary to build a feature vector, whereas, in the corresponding dynamic attributes, they consider the frequency of the fork system call invocation during the sample execution in a sandbox to build a feature vector as well. The authors used a balanced dataset of Linux binaries with benign and malign applications as input to all of the aforementioned models and the outcome was that the dynamic extraction approach outperformed the static approach, as shown in the accuracy rates (Figure 2.7). Although the dynamic attributes greatly impacted SVM and Multi-layer Perceptron results, it did not hold for Random Forest – the difference between feature vectors resulting from dynamic and static attributes was not significant, which means that using only static attributes is enough for this particular model and scenario.

Nguyen et al. (Nguyen et al., 2022) compared four different attribute extraction methods for malware detection: raw bytes (Raff et al., 2017), EMBER features (static PE file header attributes) (Anderson and Roth, 2018), CAPA features (Ballenthin and Raabe, 2020), and dynamic analysis. While the raw bytes attributes take 0.002 seconds per file to be classified, the static attributes (EMBER) take 0.09 seconds, the CAPA features take 45.75 seconds, and the dynamic analysis takes 526 seconds per file, i.e., using raw bytes on a file is over 26, 300 times faster than running dynamic analysis. Finally, according to their experiments, the raw bytes model achieved a higher malware detection accuracy than the dynamic analysis ($\approx$ 90% vs $\approx$ 85%), while the ensemble using both of them achieves almost 93%, showing that, by combining different features, it is possible to improve classification performance considering that it would be much more expensive.

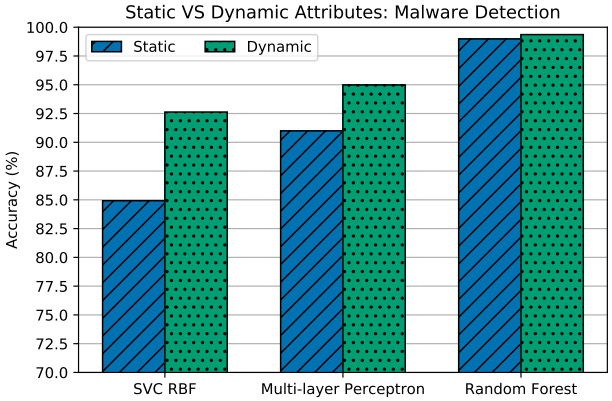


Figure 2.7: **Static VS Dynamic attributes in malware detection.** According to the classifier used, accuracy is highly impacted by the type of attributes used (Galante et al., 2019).

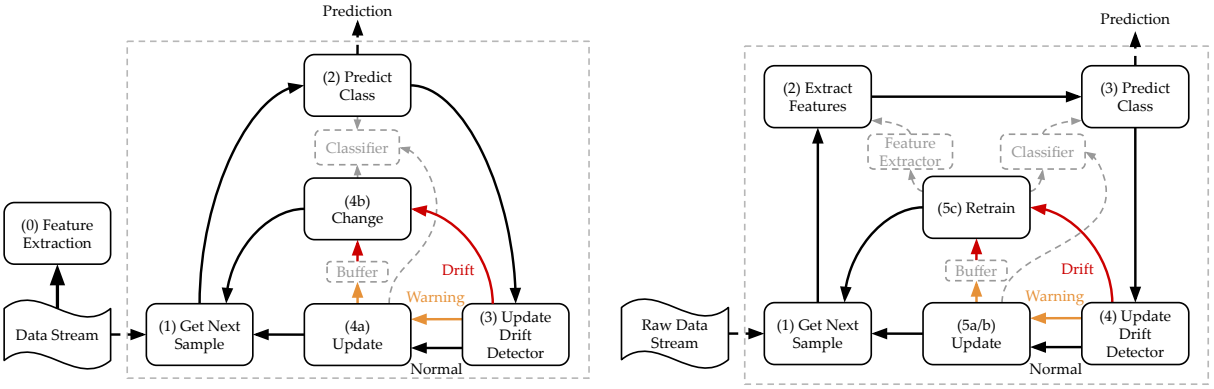
2.1.6 Feature Extraction Pitfalls

It is faster and simpler to use numerical or categorical attributes in any model, either by just encoding the categorical ones, or normalizing both of them. However, these attributes may not be directly used in the ML model, depending on their type after being extracted from raw data. For instance, a list of system calls or libraries used by software must pass through one more processing step before it can serve as input to the ML algorithm. This step is known as feature extraction, and its goal is to transform these attributes into something readable by the classifier and simplify the data while keeping the level of provided information, but reducing the number of resources used to describe them (Gron, 2017; Saxe and Sanders, 2018). Thus, there are several approaches to extracting features from attributes, and they usually rely on well-known techniques from ML literature, such as text classification, image recognition or classification, graph feature learning, and deep learning (Gibert et al., 2020; Bahri et al., 2020). Finally, according to the feature extraction selected, an ML solution may present several challenges and pitfalls.

2.1.6.1 Adapting to Changes

Many of the feature extractors mentioned in the literature need to be created based on a training dataset to, for instance, create a vocabulary (e.g., TF-IDF (Jones, 1972)) or to compute the weights of the neural network used (e.g., Word2Vec (Mikolov et al., 2013a)), similar to an ML model. Thus, as time goes by, it is necessary to update the feature extractor used if a concept drift or evolution happens in the application domain, which is something very common in cybersecurity environments due to new emerging threats (Kantchelian et al., 2013; Ceschin et al., 2018; Gibert et al., 2020). For instance, when using any vocabulary-based feature extractor for malware detection based on static features, such as the list of libraries used by a software, new libraries may be developed as time goes by (and they are not present in the vocabulary), which can result in a concept drift and make the representation created for all the new software outdated. In response to concept drift, the feature extractor may also need to be updated when it is detected and not only the classifier itself, requiring an efficient performance to update both of them.

To illustrate this challenge, Ceschin et al. created an experimental scenario using a proportionally reduced version of the AndroZoo dataset (Allix et al., 2016a) with almost 350K samples (the same we used in Section 2.1.4.4), and the DREBIN dataset (Arp et al., 2014), composed of 129, 013 samples (123, 453 benign and 5, 560 malicious Android applications) (Ceschin et al., 2022). The authors sorted the samples by their first seen date in VirusTotal (Total, 2019) and trained two base Adaptive Random Forest classifiers (Gomes et al., 2017b) with the



(a) **Traditional data stream pipeline.** Feature extraction if applied to the raw data stream data before using them in the learning cycle. Then, the classifier generates the prediction of the current sample and use it to update the model, adding it to a buffer or changing the model according to the concept drift detector level.

(b) **New data stream pipeline with feature extractor.** Every time a new sample is obtained from the raw data stream, its features are extracted and presented to a classifier, generating a prediction, which is used by a drift detector that defines the next step: update the classifier or retrain both classifier and feature extractor (Ceschin et al., 2022).

Figure 2.8: **Data stream pipeline.** Comparing the traditional data stream pipeline with a possible mitigation which adds the feature extractor in the process, generating updated features as time goes by (Ceschin et al., 2022).

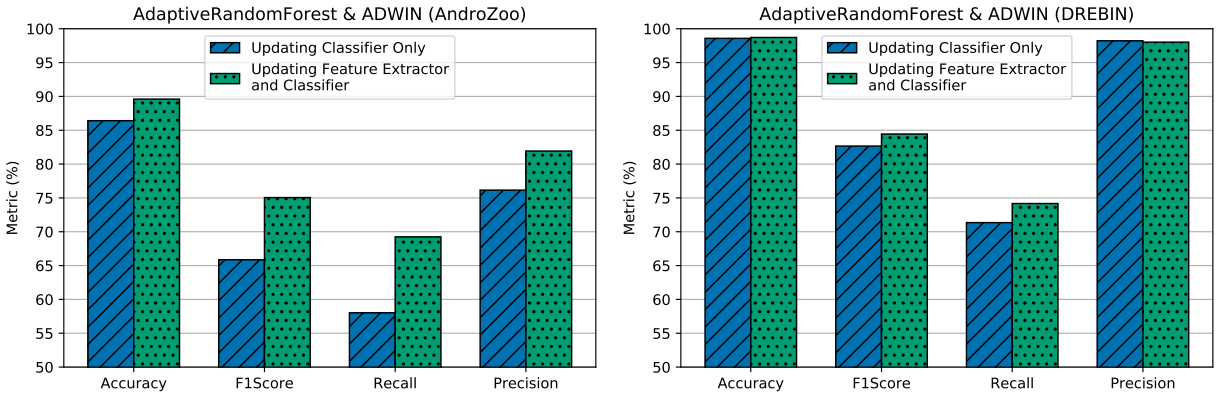
first year of data, both of them include the ADWIN drift detector (Bifet and Gavaldà, 2007), which usually has the best classification performance in the literature. When a concept drift is detected, the first classifier is updated using always the same features from the start, while the second one is entirely retrained from scratch, updating not only the classifier but also the feature extractor. In a traditional data stream learning problem that includes concept drift, the classifier is updated with new samples, which already had their features extracted previously using a feature extractor, when a change occurs (generally the ones that created the drift), as shown by steps in Figure 2.8(a) (Gama et al., 2014b). Alternatively, the data stream pipeline proposed in this experiment as mitigation to this problem also considers the feature extractor under changes, retraining both feature extractor and classifier according to the following five steps in Figure 2.8(b) (Ceschin et al., 2022).

Figure 2.9 shows the results presented by the authors regarding the impact on classification performance caused when the feature extractor is updated after a concept drift is detected (AndroZoo dataset in Figure 2.9(a) and DREBIN dataset in Figure 2.9(b)), improving by almost 10 percentage points of f1score when classifying AndroZoo dataset (from 65.86% to 75.05%). Thus, we emphasize the importance of including the feature extractor in the incremental learning process (Ceschin et al., 2022).

2.1.6.2 Adversarial Features (Robustness)

There are multiple ways to choose features that represent the samples involved in a security problem, and the final accuracy is not the only metric that should be considered during the choosing step. Another important point to take into account is the robustness of the resulting model against adversarial machine learning: attackers may try to adapt their malicious samples to make their features look similar to benign samples while maintaining the same original behavior (Ceschin et al., 2019). Thus, researchers must think like an attacker when choosing which attributes and features will be used to build an ML system for cybersecurity, given that some of them might be easily changed to trick the ML model.

To illustrate the impact of adversarial machine learning in cybersecurity solutions, let's consider the following malware detection models proposed in the academic literature: (i) `MalConv` (Raff et al., 2017) and `Non-Negative MalConv` (Fleshman et al., 2018),



(a) **AndroZoo dataset.** Adaptive Random Forest with ADWIN when classifying AndroZoo (Ceschin et al., 2022).

(b) **DREBIN Dataset.** Adaptive Random Forest with ADWIN when classifying DREBIN (Ceschin et al., 2022).

Figure 2.9: **Adapting features improves classification performance.** When considering the feature extractor in the pipeline, updating it when drift occurs is better than using a static representation (using a unique feature extractor based on the first training set) (Ceschin et al., 2022).

which are deep learning classifiers whose features are the raw bytes of an input file; and (ii) `LightGBM` (LightGBM, 2018), whose features consist of a matrix created using hashing trick and histograms based on the inputted binary files characteristics (PE header information, file size, timestamp, imported libraries, strings, etc.). Both of these models can be easily bypassed using simple strategies that create totally functional adversarial malware. The formers (raw bytes-based models) can be tricked by simply appending goodware bytes or strings present in goodware applications at the end of the malware binary. The performed appendage does not affect the original binary execution and biases the detector towards the statistical identification of these goodware features. The latter (file characteristics-based model) can be bypassed by embedding the original binary into a generic file dropper, which extracts the embedded malicious content in runtime and executes it. The dropping technique, presented in Figure 2.10, bypasses detection because the classifier would inspect the external dropper (only contains characteristics of benign applications, such as headers) instead of the embedded payload (the de facto malicious application). Previous work showed that the combination of the aforementioned counter-ML strategies can generate adversarial malware capable of bypassing both types of detection models, as well as affect the detection rate of antiviruses that rely on ML in their engines (as shown in Figure 2.10, where a malware with 91.69% average confidence is transformed into a goodware with 93.28% of average confidence) (Ceschin et al., 2019).

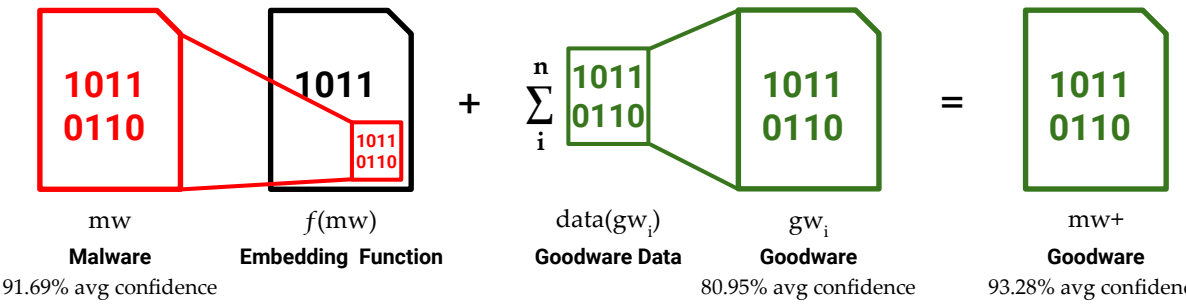


Figure 2.10: **Adversarial malware generation.** It is possible to change classifiers' output by just using an embedding function to add malware payloads within a new file and adding goodware data to it, such as strings and bytes from a set of goodware (Ceschin et al., 2019).

Other types of attributes may be more resistant to these attacks. Although approaches based on Control Flow Graphs (CFGs) are not affected by the appending of extra, spurious bytes, they can be bypassed by malware samples that modify their internal control flow structures to resemble legitimate software pieces (Calleja et al., 2018). A strategy to handle this problem is to select the features that are more resistant to modifications on the original binary (e.g., the use of loop instructions whose code is enough to distinguish malicious programs, followed by the building of a feature space containing a set of labels for each of them, thus making adversarial feature vectors more difficult to attackers (Machiry et al., 2018)).

In summary, we advocate for more studies about the robustness of features for cybersecurity, given that it is something crucial for the development of real applications. Thus, the aphorism "Garbage In, Garbage Out" used in many ML contexts is also valid for the quality of a solution, since it may become useless if subject to successful adversarial attacks. Proper ML models require high-quality training data and robust features in order to produce high-quality and robust classifiers (Geiger et al., 2020). Security-wise, it is important to understand that each ML model and feature extraction algorithm serve different threat models. Therefore, the resistance of a feature to a given type of attack should be evaluated considering the occurrence, prevalence, and impact of this type of attack in the specific application scenario (e.g., newly-released binaries being distributed with no validation codes, such as signatures and/or MACs are more prone to be vulnerable to random data append attacks than the cases in which the original binary integrity is verified).

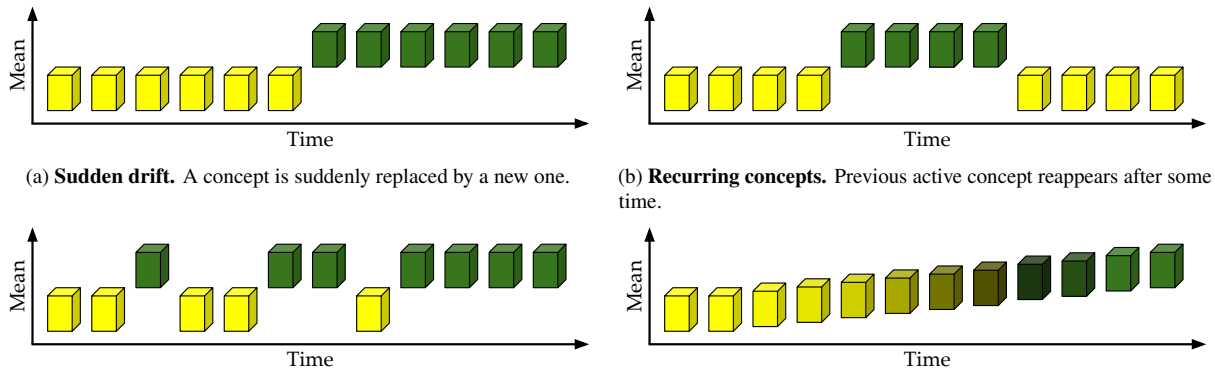
2.1.7 ML Modelling Issues and Solutions

An ML model is a mathematical model that generates predictions by finding relationships between patterns of the input features and labels in the data (Services, 2020). Thus, when using machine learning for any task, it is common to test different types of models and fine-tune them to find the one that best suits the application (Bishop, 2006). In cybersecurity, due to the dynamic scenarios presented in many tasks, streaming data models are strongly recommended to achieve a good performance, given that they belong to non-stationary distributions, new data are produced all the time, and they can be easily updated or adapted with them (Bifet et al., 2018). As a consequence, it is important to understand how to effectively use and, sometimes, implement an ML model in these scenarios, given that they may present many drawbacks that are not feasible in a real application.

2.1.7.1 Concept Drift and Evolution

Concept drift is the situation in which the relation between the input data and the target variable (the variable that needs to be learned, such as class or regression variable) changes over time (Gama et al., 2014b). It usually happens when there are changes in a hidden context, which makes it challenging since this problem spans different research fields (Wang et al., 2011). In cybersecurity, these changes are caused by the arms race between attackers and defenders, once attackers are constantly changing their attack vectors when trying to bypass defenders' solutions (Ceschin et al., 2019). In addition, concept evolution is another problem related to this challenge, which refers to the process of defining and refining concepts, resulting in new labels according to the underlying concepts (Kulesza et al., 2014). Thus, both problems (drift and evolution) might be correlated in cybersecurity, given that new concepts may result in new labels, such as new types of attacks produced by attackers. As shown in Figure 2.11, there are four types of concept drift according to the literature: (i) sudden drift, when a concept is suddenly replaced by a new one; (ii) recurring concepts, when a previous active concept reappears after some time; (iii) gradual

drift, when the probability of finding the previous concept decreases and the new one increases until it is completely replaced; and (iv) incremental drift, when the difference between the old concept and the new one is very small and the difference is only noticed when looking at a longer period (Lemaire et al., 2015). In security contexts, a sudden drift is when an attacker creates a totally new attack; gradual drift is when new types of attacks are created and gradually replace previous ones; the recurring concept is when an old type of attack starts to appear again after a given time; and incremental drift is when the attackers make few modifications in their attacks in a way that their concepts change over a large period.



(a) **Sudden drift.** A concept is suddenly replaced by a new one.

(b) **Recurring concepts.** Previous active concept reappears after some time.

(c) **Gradual drift.** The probability of finding the previous concept decreases and the new one increases until it is completely replaced.

(d) **Incremental drift.** The difference between the old concept and the new one is very small and the difference is only noticed when looking at a longer period.

Figure 2.11: **Drift types.** Different types of concept drift presented in the literature (Lemaire et al., 2015).

Despite being considered a challenge in cybersecurity (Gibert et al., 2020), few works addressed both problems in the literature. For instance, Masud et al., to the best of our knowledge, were the first to treat malware detection as a data stream classification problem and mention concept drift. The authors proposed an ensemble of classifiers that are trained from consecutive chunks of data using v-fold partitioning of the data, reducing classification error compared to other ensembles and making it more resistant to changes when classifying real botnet traffic data and real malicious executables (Masud et al., 2008). Singh et al. proposed two measures to track concept drift in static features of malware families: relative temporal similarity and meta-features (Singh et al., 2012). The former is based on the similarity score (cosine similarity or Jaccard index) between two time-ordered pairs of samples and can be used to infer the direction of the drift. The latter summarizes information from a large number of features, which is an easier task than monitoring each feature individually. Narayanan et al. presented an online ML-based framework named DroidOL to handle it and detect malware (Narayanan et al., 2016). To do so, they use inter-procedural control-flow sub-graph features in an online passive-aggressive classifier, which adapts to the malware drift and evolution by updating the model more aggressive when the error is large and less aggressively when it is small. They also propose a variable feature-set regimen that includes new features to samples, including their values when present and ignoring them when absent (i.e., their values are zero). Deo et al. proposed the use of Venn-Abers predictors to measure the quality of binary classification tasks and identify antiquated models, which resulted in a framework capable of identifying when they tend to become obsolete (Deo et al., 2016). Jordaney et al. presented Transcend, a framework to identify concept drift in classification models which compares the samples used to train the models with those seen during deployment (Jordaney et al., 2017). To do it, their framework uses conformal evaluator to compute algorithm credibility and confidence, capturing the quality of the produced results that may help to detect concept drift. Anderson et al. shown that, by using reinforcement learning

to generate adversarial samples, it is possible to retrain a model and make these attacks less effective, also protecting it against possible concept drift, given that it hardens a machine learning model against worst-case inputs (Anderson et al., 2018). Xu et al. proposed DroidEvolver, an Android malware detection system that can be automatically updated without any human involvement, requiring neither retraining nor true labels to update itself (Xu et al., 2019). The authors use online learning techniques with evolving feature sets and pseudo labels, keeping a pool of different detection models and calculating a juvenilization indicator, which determines when to update its feature set and each detection model. Finally, Ceschin et al. compared a set of Windows malware detection classifiers that use batch machine-learning models with ones that take into account the change of concept using data streams, emphasizing the need to update the decision model immediately after a concept drift is detected by a concept drift detector, which are state-of-the-art techniques used in the data stream learning literature (Ceschin et al., 2018). The authors also show that the malware concept drift is strictly related to their concept evolution, i.e., due to the appearance of new malware families.

In contrast, data stream learning literature already proposed some approaches to deal with concept drift and evolution, called concept drift detectors, that, to the best of our knowledge, were not totally explored by cybersecurity researchers. There are supervised drift detectors that take into account the ground-truth label to make a decision and unsupervised ones that do not. DDM (Drift Detection Method (Gama et al., 2004)), EDDM (Early Drift Detection Method (Baena-Garcıa et al., 2006)) and ADWIN (ADaptive WINdowing (Bifet and Gavaldà, 2007)) are examples of supervised approaches. Both DDM and EDDM are online supervised methods based on sequential error (prequential) monitoring, where each incoming example is processed separately estimating the prequential error rate. This way, they assume that the increase in consecutive error rate suggests the occurrence of concept drifts. DDM directly uses the error rate, while EDDM uses the distance error rate, which measures the number of examples between two classification errors (Baena-Garcıa et al., 2006). These errors trigger two levels: warning and drift. The warning level suggests that the concept starts to drift, updating an alternative classifier using the examples which rely on this level. The drift level suggests that the concept drift occurred, and the alternative classifier build during the warning level replaces the current classifier. ADWIN keeps statistics from sliding windows of variable size, which are used to compute the average of the change observed by cutting these windows at different points. If the difference between two windows is greater than a pre-defined threshold, it considers that a concept drift happened, and the data from the first window is discarded (Bifet and Gavaldà, 2007). Different from the other two methods, ADWIN has no warning level. Once a change occurs, the data that is out of the window is discarded and the remaining ones are used to retrain the classifier. Unsupervised drift detectors such as the ones proposed by Žliobaité et al. may be useful when delays are expected given that they do not rely on the real label of the samples, which need to be known by supervised methods, and most of the time in cybersecurity it does not happen in practice (Žliobaité, 2010). These unsupervised strategies consist in comparing different detection windows of fixed length using statistical tests over the data themselves, on the classifier output labels or its estimations (that may contain errors) to detect if both come from the same source. In addition, active learning may complement these unsupervised methods by requiring the labels of only a subset of the unlabeled samples, which could improve the drift detection and the overall classification performance.

Some authors also created different classification models and strategies that deal with both concept drift and concept evolution. Shao et al. proposed SyncStream, a classification model for evolving data streams that use prototype-based data representation, P-Tree data structure, and just a small set of both short and long-term samples based on error-drive representativeness

learning (instead of using base classifiers or windows of data) (Shao et al., 2014). ZareMoodi et al. created a new supervised chunk-based method for novel class detection using ensemble learners, local patterns, and connected components of neighborhood graphs (ZareMoodi et al., 2015). The same authors also proposed a new way to detect evolving concepts by optimizing an objective function using a fuzzy agglomerative clustering method (ZareMoodi et al., 2019). Hosseini et al. created SPASC (Semi-supervised Pool and Accuracy-based Stream Classification), an ensemble of classifiers where each classifier holds a specific concept, and new samples are used to add new classifiers to the ensemble or to update the existing ones according to their similarity to the concepts (Hosseini et al., 2015). Dehghan et al. proposed a method based on the ensemble to detect concept drift by monitoring the distribution of its error, training a new classifier on the new concept to keep the model updated (Dehghan et al., 2016). Ahmadi et al. created GraphPool, a classification framework that deals with recurrent concepts by looking at the correlation among features, using a statistical multivariate likelihood test, and maintaining the transition among concepts via a first-order Markov chain (Ahmadi and Kramer, 2017). Gomes et al. presented the Adaptive Random Forest (ARF) algorithm, an adaptation of the classical random forest algorithm with dynamic update methods to deal with evolving data streams. The ARF also contains an adaptive strategy that uses a concept drift detector in each tree to track possible changes and to train new trees in the background (Gomes et al., 2017b). Finally, Siahroudi et al. proposed a method using multiple kernel learning to detect novel classes in non-stationary data streams (Siahroudi et al., 2018). The authors do it by classifying each new instance by computing their distance to the previously known classes in the feature space and updating the model based on their true labels.

We advocate for more collaboration between data stream learning and cybersecurity, given that the majority of cybersecurity works presented in this section do not use data stream approaches (including concept drift detectors), they both have a lot of practice problems in common and may benefit each other. For instance, data stream learning could benefit from real cybersecurity datasets that could be used to build real-world ML security solutions, resulting in higher quality research that may also be useful in other ML research fields. Finally, developing new drift detection algorithms is important to test their effectiveness in different cybersecurity scenarios and ML models.

2.1.7.2 Adversarial Attacks

In most cybersecurity solutions that use Machine Learning, models are prone to suffer adversarial attacks, where attackers modify their malicious vectors to somehow make them not being detected (Ceschin et al., 2019). We already mentioned this problem related to feature robustness in Section 2.1.6.2, but ML models are also subject to adversaries. These adversarial attacks may have several consequences such as allowing the execution of malicious software, poisoning an ML model or drift detector if they use new unknown samples to update their definitions (without a ground-truth from other sources), and producing, as a consequence, concept drift and evolution. Thus, when developing cybersecurity solutions using ML, both features and models must be robust against adversaries.

Aside from using adversarial features, attackers may also directly attack ML models. There are two types of attacks: white-box attacks, where the adversary has full access to the model, and black-box attacks, where the adversary has access only to the output produced by the model, without directly accessing it (Athalye et al., 2018). A good example of white-box attacks are gradient-based adversarial attacks, which consist in using the weights of a neural network to obtain perturbation vectors that, combined with an original instance, can generate an adversarial one that may be classified by the model as being from another class (Goodfellow et al., 2014b).

Many strategies use neural network weights to produce these perturbations (Athalye et al., 2018), which not only affects neural networks but a wide variety of models (Goodfellow et al., 2014b). Other simpler white-box attacks such as analyzing the model, for instance, the nodes of a decision tree or the support vectors used by an SVM, could be used to manually craft adversarial vectors by simply changing the original characteristics of a given sample in a way that it can affect its output label. In contrast, black-box attacks tend to be more challenging and real for adversaries, given that they usually do not have access to implementations of cybersecurity solutions or ML models, i.e., they have no knowledge about which features and classifier a given solution is using and usually only known which is the raw input and the output. Thus, black-box attacks rely on simply creating random perturbations and testing them in the input data (Guo et al., 2019), changing characteristics from samples looking at instances from all classes (Athalye et al., 2018), or trying to mimic the original model by creating a local model trained with samples submitted to the original one, using the labels returned by it, and then analyzing or using this new model to create an adversarial sample (Papernot et al., 2016).

In response to adversarial attacks, defenders may try different strategies to overcome them, searching for more robust models that make this task harder for adversaries. One response to these attacks is Generative Adversarial Networks (GANs), which are 2-part, coupled deep learning systems in which one part is trained to classify the inputs generated by the other. The two parts simultaneously try to maximize their performance, improving the generation of adversaries, that are used to defeat the classifier, and then used to improve their detection by training the classifier with them (Goodfellow et al., 2014a; Hu and Tan, 2017). Another valid strategy is to create an algorithm that, given a malign sample, automatically generates adversarial samples, similar to data augmentation or oversampling techniques that insert benign characteristics into it, which are then used to train or update a model. This way, the model will learn not only the normal concept of a sample but also the concept of its adversaries' versions, which will make it more resistant to attacks (Papernot et al., 2015a; Grosse et al., 2017). Some approaches also tried to fix limitations of already developed models, such as `MalConv` (Fleshman et al., 2018), an end-to-end deep learning model, which takes as input raw bytes of a file to determine its maliciousness. `Non-Negative MalConv` proposes an improvement to `MalConv`, with an identical structure, but having only non-negative weights, which forces the model to look only for malicious evidence rather than looking for both malicious and benign ones, being less prone to adversaries that try to copy benign behavior (Fleshman et al., 2018). Despite that, even `Non-Negative MalConv` has weaknesses that can be explored by attackers (Ceschin et al., 2019), which makes this topic an open problem to be solved by future research. We advocate for more work and competitions, such as the Machine Learning Security Evasion Competition (MLSEC) (Balazs, 2020), that encourage the implementation of new defense solutions that minimize the effects of adversarial attacks.

2.1.7.3 Class imbalance

Class imbalance is a problem already mentioned in this work, but on the dataset side (Section 2.1.4.3). In this section we are going to discuss the effects of class imbalance in the ML model and present some possible mitigation techniques that rely on improving the learning process (cost-sensitive learning), using ensemble learning (algorithms that combine the results of a set of classifiers to make a decision) or anomaly detection (or one-class) models (Kaur et al., 2019; Gomes et al., 2019). This way, when using cost-sensitive learning approaches, the generalization made by most algorithms, which makes minority classes ignored, is adapted to give each class the same importance, reducing the negative impact caused by class imbalance. Usually, cost-sensitive learning approaches increase the cost of incorrect predictions of minority

classes, biasing the model in their favor and resulting in better overall classification results (Kaur et al., 2019). Such techniques are not easy to implement in comparison to sampling methods presented in Section 2.1.4.3 but tend to be much faster given that they just adapt the learning process, without generating any artificial data (Gomes et al., 2019).

In addition, ensemble learning methods that rely on bagging (Breiman, 1996) or boosting techniques (such as AdaBoost (Freund and Schapire, 1997)) present good results with imbalanced data (Galar et al., 2012), which is one of the reasons that random forest perform well in many cybersecurity tasks with class imbalance problems, such as malware detection (Ceschin et al., 2018). Bagging consists in training the classifiers from an ensemble with different subsets of the training dataset (with replacement), introducing diversity to the ensemble, and improving overall classification performance (Breiman, 1996; Galar et al., 2012). The AdaBoost technique consists in training each classifier from the ensemble with the whole training dataset in iterations. After each iteration, the algorithm gives more importance to difficult samples, trying to correctly classify the samples that were incorrectly classified by giving them different weights, very similar to what cost-sensitive learning does, but without using a cost to update the weights (Freund and Schapire, 1997; Galar et al., 2012).

Even though all the methods presented so far are valid strategies to handle imbalanced datasets, sometimes the distribution of classes is too different that it is not viable to use one of them, given that the majority of the data will be discarded (undersampling), poor data will be generated (oversampling), or the model will not be able to learn the concept of the minority class (Gomes et al., 2019). In these cases, anomaly detection algorithms are strongly recommended, given that they are trained over the majority class only and the remaining ones (minority class) are considered anomalous instances (Salehi and Rashidi, 2018; Gomes et al., 2019). Two great examples of anomaly detection models are isolation forest (Liu et al., 2012) and one-class SVM (Schölkopf et al., 1999). Both of them try to fit the regions where the training data is most concentrated, creating a decision boundary that defines what is normal and what is an anomaly.

Finally, when building an ML solution that has imbalanced data, as well as testing several classifiers and feature extractors, it is also important to consider the approaches presented here for both dataset and model sides. Also, it is possible to combine more than one method, for instance, generating a set of artificial data and using cost-sensitive learning strategies, which could increase classification performance in some cases. We strongly recommend that the cybersecurity researchers include some of these strategies in their work, given that it is difficult to find solutions that actually consider class imbalance.

2.1.7.4 Transfer Learning

Transfer learning is the process of learning a given task by transferring knowledge from a related task that has already been learned. It has shown to be very effective in many ML applications (Torrey and Shavlik, 2010), such as image classification (Raghu et al., 2019; Hussain et al., 2019) and natural language processing problems (Howard and Ruder, 2018; Radford, 2018). Recently, Microsoft and Intel researchers proposed the use of transfer learning from computer vision to static malware detection (Chen, 2018), representing binaries as grayscale images and using inception-v1 (Szegedy et al., 2014) as the base model to transfer knowledge (Chen et al., 2020). The results presented by the authors show a recall of 87.05%, with only a 0.1% of false positive rate, indicating that transfer learning may help to improve malware classification without the need of searching for optimal hyperparameters and architectures, reducing the training time and the use of resources.

In addition, if the network used as the base model is robust, they probably contain robust feature extractors. Consequently, by using these feature extractors, the new model produced inherits their robustness, producing new solutions that are also robust to adversarial attacks, achieving high classification performances, without much data, and with no need to use a lot of resources as some adversarial training approaches (Shafahi et al., 2019). At the same time that transfer learning might be an advantage, it may also be a problem according to the base model used because usually, these base models are publicly available, which means that any potential attackers might have access to them and produce an adversarial vector that might affect both models: the base and the new one (Rezaei and Liu, 2019). Thus, it is important to consider the robustness of the base model when using it to transfer learning to produce a solution without security weaknesses. Finally, despite presenting promising results, the model proposed to detect malware by using transfer learning cited at the beginning of this subsection (Chen et al., 2020) may be affected by adversarial attacks, given that its base model is affected by them as already shown in the literature (Goodfellow et al., 2014b; Carlini and Wagner, 2017).

2.1.7.5 Implementation

Building a good Machine Learning model is not the last challenge to deploying ML approaches in practice. The implementation of these approaches might also be challenging. The existing frameworks, such as scikit-learn (Pedregosa et al., 2011) and Weka (Hall et al., 2009), usually rely on batch learning algorithms, which may not be useful in dynamic scenarios where new data are available all the time (as a stream), requiring the model to be updated frequently with them (Gomes et al., 2019). In these cases, ML implementations for streaming data, such as Scikit-Multiflow (Montiel et al., 2018), Massive Online Analysis (MOA) (Bifet et al., 2010), River (Montiel et al., 2021), and Spark (Spark, 2020; Matei Zaharia and Xin, 2016) are highly recommended, once they provide ML algorithms that could be easily used in real cybersecurity applications. Also, adversarial machine learning frameworks, such as CleverHans (Papernot et al., 2018a) and SecML (Melis et al., 2019), are important to test and evaluate the security of ML solutions proposed. Thus, contributing to streaming data and adversarial machine learning projects is as important as contributing to well-known ML libraries, and we advocate for that to make all research closer to real-world applications. Note that we are not just talking specifically about contributing with new models, but also prepossessing and evaluating algorithms that may be designed only in batch learning, and could also be a good contribution to streaming learning libraries. We believe that more contributions to these projects would benefit both industry and academia with higher quality solutions and research, given the high number of researches using only batch learning algorithms nowadays, even in cybersecurity problems.

In addition, multi-language codebases may be a serious challenge when implementing a solution, once different components may be written in different languages, not being completely compatible, becoming incompatible with new releases, or being too slow according to the code implementation and language used (Schelter et al., 2018). Thus, it is common to see ML implementations being optimized by C and C++ under the hood, given that they are much faster and more efficient than Python and Java, for instance. Despite such optimizations being needed to make many solutions feasible in real-world solutions, they are not always performed given that (i) researchers create their solutions as prototypes that only simulate the real world, not requiring optimizations, and (ii) optimizations require knowledge about code optimization techniques that are specific or may be limited to a given type of hardware, such as GPUs (Schelter et al., 2018). Also, implementing data stream algorithms is a hard task, given that we need to execute the whole pipeline continuously: if any component of this pipeline fails, the whole system may fail (Gomes et al., 2019).

Another challenge is to ensure a good performance for the proposed algorithms and models (Coleman et al., 2019). A good performance is essential to deploy ML in the security context because most of the detection solutions operate in runtime to detect attacks as early as possible and slow models will result in a significant slowdown to the whole system operation. To overcome the performance barriers of software implementations, many security solutions opt to outsource the processing of ML algorithms to third-party components. A frequent approach in the security context is to propose hardware devices to perform critical tasks, among which is the processing of ML algorithms (Botacin et al., 2019). Alternatively, security solutions might also outsource scanning procedures to the cloud. Many research works proposed cloud-based AVs (Dev et al., 2016; Jarabek et al., 2012), which have the potential to include ML-based scans among their detection capabilities and streamline such checks to the market. We understand that these scenarios should be considered for the proposal of new ML-based detection solutions.

2.1.8 Evaluation

Knowing how to correctly evaluate an ML system is essential to building a security solution, given that some evaluations may result in wrong conclusions that may backfire in security contexts, even when using traditional ML evaluation best practices (Cavallaro, 2019; Pendlebury et al., 2019; Ceschin et al., 2018). For instance, consider that a malicious threat detection model is evaluated using ten samples: eight of them are benign and two are malign. This model has an accuracy of 80%. Is 80% a good accuracy? Assuming that the model classifies correctly only the eight benign samples, it is not capable of identifying any malign sample and yet it has an accuracy of 80%, giving a false impression that the model works significantly well. Thus, it is important to take into account the metric used to produce high-quality systems that solve the problem proposed by a certain threat model.

2.1.8.1 Metrics

To correctly evaluate a solution the right metrics need to be selected to provide significant insights that can present different perspectives of the problem according to its real needs, reflecting the real world (Gron, 2017). One of the most used metrics by ML solutions is accuracy, which consists in measuring the percentage of samples correctly classified by the model divided by the total number of samples seen by it (usually from the testing set) (Ferri et al., 2009). The main problem with this metric is that it may provide wrong conclusions according to the distribution of the datasets used, as already shown by the malicious threat detection model example. Thus, if the dataset is imbalanced, accuracy is not recommended since it will give much more importance to the majority class, presenting, for instance, high values even if a minority class is completely ignored by the classifier (Ferri et al., 2009).

An interesting way to evaluate model performance is by checking the confusion matrix, a matrix where each row represents the real label and each column represents a predicted class (or vice versa) (Gron, 2017). By using this matrix, it is possible to check a lot of information about the model, for instance, which class is more difficult to classify or which ones are being confused the most. In addition, it is also possible to calculate false positives and false negatives, which lead us to recall, precision, and, consequently, f1score. Recall measures the percentage of positive examples that are correctly classified, precision measures the percentage of examples classified as positive that are positive, and f1score is the harmonic mean of both of them (Davis and Goadrich, 2006). With these metrics, it is possible to calibrate the model according to the task being performed (Ceschin et al., 2018). For a malware detector, for instance, it may be better to not detect malware than to block benign software (high precision), given that it would

affect directly the user experience. Imagine that a user is using his computer and, when opening Microsoft Word, the classifier believes that it is malware and blocks its execution. Even detecting the majority of the malware, the classifier may turn the computer useless, as it would block a great part of the benign applications. In contrast, in more sensitive systems, one might want the opposite (high recall), blocking all the malign actions, even with some benign being ”sacrificed“.

Recently, some authors introduced metrics to evaluate the quality of the models produced. Jordaney et al. proposed Conformal Evaluator (CE), an evaluation framework that computes two metrics (algorithm confidence and credibility) to measure the quality of the produced ML results, evaluating the robustness of the predictions made by the algorithm and their qualities (Jordaney et al., 2017). Pendlebury et al. introduced another evaluation framework called TESSERACT, which compares classifiers in a realistic setting by introducing a new metric, Area Under Time (AUT). This metric captures the impact of time decay on a classifier, which is not evaluated in many works, confirming that some of them are biased. Thus, we support the development of these kinds of work to better evaluate new ML solutions considering a real-world scenario that allows the implementation to be used in practice.

2.1.8.2 Comparing Apples to Orange

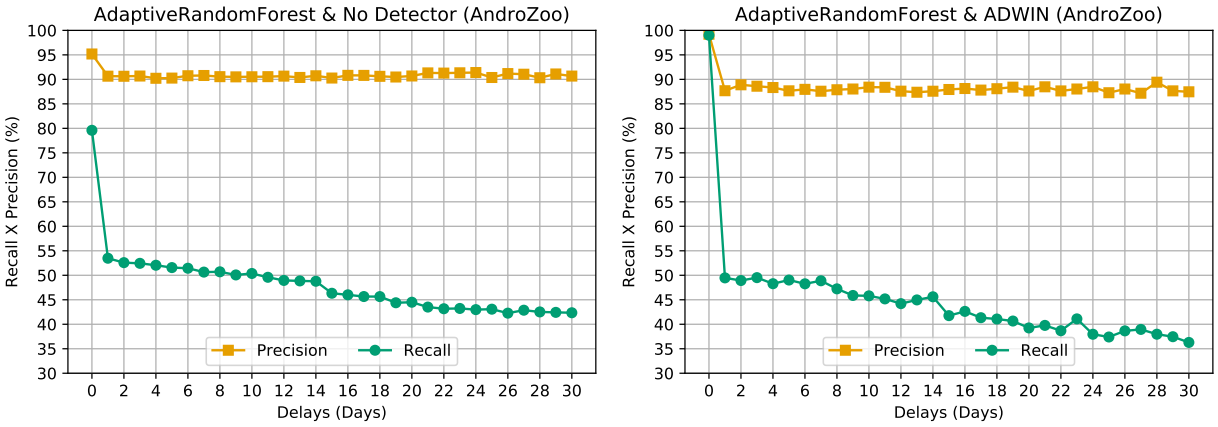
It is not unusual for researchers to compare their evaluation results with other prior literature solutions that face the same problems (e.g., comparing the accuracy of two detection models). Such comparison, however, should be carefully performed to avoid misleading conclusions.

The lack of standard publicly available repositories makes much work to use their own dataset when building a solution. These new solutions have their results usually compared with other works reported in the literature. Whereas comparing approaches seems to be straightforward, authors should care to perform fair evaluations, such as comparing studies leveraging the same datasets, avoiding presenting results deemed to outperform literature results but which do not achieve such performance in actual scenarios.

As an analogy, consider image classification problems whose objective is to identify objects represented in images (for instance, buildings, animals, locations, etc). These challenges often provide multiple datasets that are used as baseline by many solutions. For instance, the CIFAR challenge (Krizhevsky, 2012) is composed of two datasets: CIFAR-100, which has one hundred classes of images, and CIFAR-10, which is a filtered version of CIFAR-100, containing just ten classes. Imagine two research work proposing distinct engineering solutions for image classification, one of them leveraging CIFAR-10 and the other leveraging CIFAR-100. Although one of the approaches presents a higher accuracy than the other, is it fair to say that this one is better than the other? No, because the task involved in classifying distinct classes is also distinct. The same reasoning is valid for any cybersecurity research involving ML. Thus, authors should care to not perform comparisons involving distinct classes of applications, such as comparing, for instance, approaches involving Dynamic System Call Dependency Graphs, a computationally costly approach, with static feature extraction approaches is misleading because each type of work presents different nature and challenges. Finally, it is strongly recommended that researchers share their source codes with the community to make their work compatible with any other dataset (which in the majority of the works are not shared), allowing future researchers to compare different approaches in the same scenario.

2.1.8.3 Delayed Labels Evaluation

One particularity of security data is that they usually do not have ground-truth labels available right after new data are collected, as already shown in Section 2.1.4.2. Due to that, there is a



(a) **AndroZoo dataset without drift detector.** Results for the adaptive random forest with no drift detector using AndroZoo.

(b) **AndroZoo dataset with drift detector.** Results for the adaptive random forest with ADWIN drift detector using AndroZoo.

Figure 2.12: **Delayed Labels Evaluations.** Models with drift detection may not perform as expected in real-world solutions, despite having the best performance in scenarios where delays are not considered.

gap between the data collection and their labeling process, which is not considered in many cybersecurity types of research that uses ML, i.e., in many proposed solutions in the literature, the labels are available at the same time as the data, even in online learning solutions. Some of them ignore the ground-truth label and use the same labels that the ML classifier predicts (Xu et al., 2019), which may make the model subject to poisoning attacks and, consequently, decrease its detection rate as time goes by (Taheri et al., 2019).

Considering malware detection, the majority of the works use only a single snapshot of scanning results from platforms like VirusTotal, without considering a given delay before actually using the labels, which may vary from ten days to more than two years (Zhu et al., 2020). A recent study from Botacin et al. analyzed the labels provided by VirusTotal for 30 consecutive days from two distinct representative malware datasets (from Windows, Linux, and Android) and showed that solutions took about 19 days to detect the majority of the threats (Botacin et al., 2020b). To study the impact of these delayed labels in malware detectors using ML, we simulated a scenario using the AndroZoo dataset (Allix et al., 2016a) and online ML techniques with and without drift detectors by providing the labels of each sample N days after they are available to further update the decision model (Adaptive Random Forest (Gomes et al., 2017b) trained using TF-IDF feature extractor). The results of this experiment are shown in Figures 2.12(a) and 2.12(b). We can note that when not considering a delay, using a drift detector improves the detection rate a lot. However, after one day of delay, this is not true anymore: the model that does not consider concept drift performs better overall, despite both being affected by this problem, dropping to half of the original precision in the case of Adaptive Random Forest with ADWIN. These results indicate that: (i) in scenarios where delayed labels exist, ML models do not perform the way they are evaluated without these conditions, and (ii) models that make use of drift detectors, such as ADWIN, present lower detection rates in comparison to those that do not use them, since the concept being learned by the model is outdated when a drift is detected, increasing false negatives.

Finally, we advocate for drift detection strategies that consider the delay of labels and mitigation techniques to overcome this problem, such as active learning or approaches to provide real labels with less delay, to produce better solutions (Krempl et al., 2014).

2.1.8.4 Online vs Offline Solutions

We previously advocated for real-world considerations when developing an ML model (Section 2.1.7). We also advocate for real-world considerations when evaluating the developed solutions. Each evaluation should consider the scenario in which it will be applied. A major implication of this requirement is that online and offline solutions must be evaluated using distinct criteria. Offline solutions are often leveraged by security companies in their internal analysis (e.g., an AV company investigating whether a given payload is malicious or not to develop a signature for it that will be distributed to the customers in the future). Online solutions, in turn, are leveraged by the endpoints installed in the user machines (e.g., a real-time AV monitoring a process behavior, a packet filter inspecting the network traffic, and so on). Due to their distinct nature, offline solutions are much more flexible than online solutions. Whereas offline solutions can rely on complex models that are run on large clusters, online solutions must be fast enough to be processed in real-time without adding a significant performance overhead to the system. Moreover, offline solutions can collect huge amounts of data during a sandboxed execution and thus input them to models relying on a large moving window for classification. Online solutions, in turn, operate in memory-limited environments and aim to detect the threat as soon as possible. Thus, they cannot rely on large moving windows. These differences must be considered when evaluating the models to avoid common pitfalls, such as applying the same criteria for both applications. Due to their distinct nature, online solutions are expected to present more false negatives than offline solutions, as they have make fast decisions about the threat. On the other hand, offline solutions will detect more samples, as they have more data to decide, but the detection is likely to happen later than in an online detector, only after multiple windows (e.g., when a malware already infected the system and/or when a remote payload already exploited a vulnerable application). For a more complete security treatment, modern security solutions should consider hybrid models that employ multiple classifiers, each one with a distinct window size (Sun et al., 2020).

2.1.9 Discussion: Understanding ML

Once we have discussed all the steps required to apply ML to security problems, we now discuss how this application should be understood, its limitations, and existing open gaps.

2.1.9.1 A ML model is a type of signature

A frequent claim of most products and research work leveraging ML as part of their operation, mainly antivirus and intrusion detection systems, is that this use makes their approaches more flexible and precise than the so-far applied signature schemes (Permeh, 2017). This is somehow a pitfall, as in the last instance, an ML model is just a weighted signature of boolean values. One can even convert an ML model to typical YARA signatures (Saxe, 2020), as deployed by many security solutions. Although the weights can be adjusted to add some flexibility to an ML model, the model itself cannot be automatically extended beyond the initially considered boolean values without additional processing (re-training or feature re-extraction), which is an already existing drawback for signature schemes. In this sense, the adversarial attacks against the ML models can see as analogous to the polymorphic attacks against typical signature schemes (Tasiopoulos and Katsikas, 2014): whereas a typical polymorphic threat attempt to directly change its features to pass the checks (weight=1), a modern attack against ML models indirectly attempts to do so by presenting a relative feature frequency higher or lower than the considered in the ML model.

2.1.9.2 There is no such thing as a 0-day detector

It is usual for many ML-based detector proposals to state that their approach is resistant to 0-day attacks (i.e., attacks leveraging so-far unknown threats and/or payloads) because they rely on an ML model (Abri et al., 2019; Kumar and Bhim Bhan Singh, 2018). This is also somehow a pitfall, and we credit this as the poor understanding that ML models are a type of signature, as previously presented. In fact, the ability of a model to detect a new threat depends on the definition of what is new. If a new sample is understood as any sample that an attacker can create, certainly ML models will be able to detect many 0-days, as many of these new payloads will be variations of previously known threats, a scenario for which weighted signatures generalize well. However, if we consider as new something unknown to the model in all senses (e.g., a payload having a feature that is not present in the model), no ML model will be able to detect it as malicious, thus nature of the presented problem of concept drift and evolution. Therefore, ML models should no be understood as simply more resistant detectors than typical signature schemes, but as a type of signature that can be more generalized.

2.1.9.3 Security & Explainable Learning

Most research works put a big focus on achieving good detection rates, but not all of them put the same effort into discussing and explaining the results (Arp et al., 2014), which would be essential to allow security incident response teams (CSIRTs) to fill the identified security breaches. The scenario is even more complicated when deep learning approaches are considered, as in most cases there is not a clear explanation for the model operation (Gade et al., 2020). A model to detect cybersecurity threats can yield valuable insights besides its predictions since based on the explanation provided by the model, the security around the monitored object can be improved in future versions. As an example, a model able to explain that some threats were identified due to the exploitation of a given vulnerability might trigger patching procedures to fix the associated applications. Explainability has different levels of relevance according to the domain; we argue that for several cybersecurity tasks, they are essential to make it possible to apply countermeasures to security threats.

2.1.9.4 The arms race will always exist

The arms race between attack and defense solutions is a cycle that will always exist, requiring that both sides keep investing in new approaches to overcome their adversaries. Thus, new approaches from one the sides result in a reaction from another one, as shown in Figure 2.13, where defense solutions follow the steps mentioned in this work to build ML defense solutions and attackers follow the steps required to create an attack: they first find the weakness of the defense solutions, use this weakness to develop an exploit, which is then delivered to be executed, producing new data for ML models to be trained, restarting the cycle (Huang et al., 2018). Finally, we advocate that defense solutions try to reduce the gap present in this cycle (the development of new attacks and the generation of solutions for them) by following the recommendations of this work and other works in the literature that make use of robustness verification to prevent attacks against adversaries (Li et al., 2020).

To help the development of future ML solutions for cybersecurity, we created a checklist[1] that is an adaptation based on the challenges, pitfalls, and problems reported in this work. Thus,

[1]The draft version of the checklist is available at `https://secret.inf.ufpr.br/machine-learning-in-security-checklist/`.

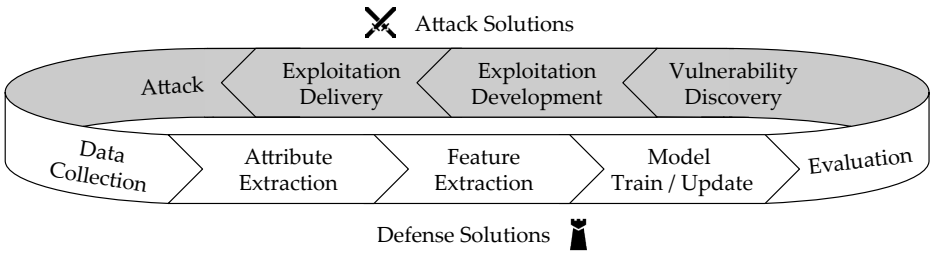


Figure 2.13: **Attack VS Defense Solutions.** The arms race created by both generates a never-ending cycle.

anyone developing or reviewing a solution can fill the questions on this checklist and get feedback reporting what could be improved or corrected according to our findings.

2.1.9.5 The future of ML for cybersecurity

ML for cybersecurity has become so popular that is hard to imagine a future without it. However, this does not mean that there is no room for improvement. As an immediate task, researchers will put their effort to mitigate adversarial attacks, which will enable even more applications to migate ML solutions. A massive migration will require not only robust security guarantees but also privacy ones, which is likely to be achieved via merging ML and cryptography. The field is experiencing the rise of ML classifiers based on homomorphic cryptography (Phong et al., 2018), which allows data to be classified without decryption. We believe that this type of approach has the potential to upstream many ML-powered solutions, such as cloud-based AVs, which will be able to scan files in the cloud while preserving the privacy of the file's owners.

2.1.10 Conclusion

In this paper, we introduced a set of problems and challenges that have been observed (too often) in ML techniques applied to diverse cybersecurity solutions. We presented practical examples of cybersecurity scenarios in which ML might either be incorrectly applied or contain blind spots (important details that were not considered, discussed, or observed). When possible, we showed techniques to address those common issues. To make a step toward that, in this paper we summarized and provided insights on the following main points: data collection issues (data leakage, data labeling, class imbalance, and dataset size definition); modeling (different attribute and feature extraction methods, adaptation to changes, i.e. concept drift/evolution, adversarial features and model attacks, one-class models, cost-sensitive, ensemble learning, transfer learning, and implementation challenges); and evaluation concerns (adequate use of metrics, thorough comparison of previous/existing solutions, delayed labels, and online vs. offline experiments).

Finally, our main recommendations to improve ML in security are the following:

- **Stop looking only at metrics, and start looking at effects:** many of the challenges presented in this work remain as open research problems, which would benefit both academy and industry if properly solved. In addition, their mitigation would foster the use of ML for cybersecurity problems, and improve the cybersecurity field in the same way ML advanced other research fields. Community players (security and machine learning) have to take into consideration the plenty of peculiarities associated with cybersecurity data and its sources.

- **Commit yourself to the real world:** we advocate that future research works always keep the motto "machine learning that matters" (Wagstaff, 2012) in mind when developing

new solutions. If your work is losing connection to problems of the real world, science, and society, we have a problem!

- **Check your work:** our recommendation is to carefully observe all the items, and to consider each of them during the design and implementation of ML models for cybersecurity. To encourage that, we presented a checklist (draft version available at `https://secret.inf.ufpr.br/machine-learning-in-security-checklist/`; validation tests ongoing) to serve as a reminder and prevent researchers and practitioners from committing these common mistakes or at least being aware of their existence.

3 ADVERSARIAL MACHINE LEARNING

In this chapter, I present some adversarial machine learning strategies to attack Windows malware detectors. To do so, I show two papers that are related to my experience in a Machine Learning Security Evasion Competition (MLSEC) that happened in 2019 and 2020 (Anderson, 2019; Balazs, 2020), which had 32 and 60 teams respectively, and my team managed to win the 2020 and 2021 editions (in total, there were more than 100 teams in all the challenges). First, I investigate some simple attack strategies that are surprisingly effective, affecting malware detection in all the models proposed in the challenge. Then, I propose an attack that not only affects ML models but also real anti viruses (AVs), shedding some light on what can be improved in Windows malware detectors. Finally, I test my own ML malware detector with different attack strategies and discover interesting insights that may help future research, such as (i) ML models should find a way to stop detecting packers (and not malware); (ii) adversarial samples should be carefully used to update detection models, given that they may poison them; and (iii) malware detectors should be able to extract embedded payloads to classify them without considering the first layer only.

3.1 SHALLOW SECURITY: ON THE CREATION OF ADVERSARIAL VARIANTS TO EVADE MACHINE LEARNING-BASED MALWARE DETECTORS

This paper was published in the Proceedings of the 3rd Reversing and Offensive-Oriented Trends Symposium (ROOTS), 2019 (Ceschin et al., 2019).

Fabrício Ceschin[1], Marcus Botacin[1], Heitor Murilo Gomes[2], Luiz S. Oliveira[1], André Grégio[1]
[1]Federal University of Paraná, Brazil
{fjoceschin, mfbotacin, lesoliveira, gregio}@inf.ufpr.br
[2]University of Waikato, New Zealand
heitor.gomes@waikato.ac.nz

3.1.1 Abstract

The use of Machine Learning (ML) techniques for malware detection has been a trend in the last two decades. More recently, researchers started to investigate adversarial approaches to bypass these ML-based malware detectors. Adversarial attacks became so popular that a large Internet company has launched a public challenge to encourage researchers to bypass their (three) ML-based static malware detectors. Our research group teamed to participate in this challenge in August/2019, accomplishing the bypass of all 150 tests proposed by the company. To do so, we implemented an automatic exploitation method which moves the original malware binary sections to resources and includes new chunks of data to it to create adversarial samples that not only bypassed their ML detectors, but also real AV engines as well (with a lower detection rate than the original samples). In this paper, we detail our methodological approach to overcome the challenge and report our findings. With these results, we expect to contribute with the community and provide better understanding on ML-based detectors weaknesses. We also pinpoint future research directions toward the development of more robust malware detectors against adversarial machine learning.

3.1.2 Introduction

Malware detection is an ever growing research field due to the challenges imposed by constant evolving threats. The current state-of-the-art techniques for malware detection are Machine Learning (ML)-based approaches. However, they are not perfect and still have breaches to be exploited by attackers (Goodfellow et al., 2014b), despite solving many previously existing problems, such as the classification of dense volume of malware data. Therefore, the use of ML stimulated the already existing arms race, with attackers generating malware variants to exploit the drawbacks of ML-based approaches and defenders developing new classification models.

Adversarial attacks against machines learning models have become so popular to the point of an Internet company (Endgame, Inc) launching a challenge (EndGame, 2019) in August/2019 to evaluate the resistance of three static analysis-based ML models against malware variants. Two of these models are deep neural networks which use raw data as input, while the last model is a decision tree which uses file metadata as input. Our team participated in this challenge and was able to bypass the three models using modified versions of the 50 samples originally provided by the organizers. To generate the adversarial malware, we implemented an automatic exploitation method moving the original malware binary sections to resources and including new chunks of data to it to create adversarial samples that not only bypassed challenge's detectors, but also real AVs as well (with a lower detection rate than the original samples).

More than competing, our main goal was to investigate real models robustness against adversarial malware samples. Our experiments revealed that the models have severe weaknesses so that they can be easily bypassed by attackers motivated to exploit real systems. These findings motivated us to write this report pinpointing possible mitigation and future research work in the ML-based malware detection field.

During our evaluation of the provided models, we discovered that:

1. ML models based on raw binary data can be bypassed by appending data to the binaries.
2. Frequency-based ML models can be bypassed by embedding goodware strings in malware binaries.
3. PE format-aware classifiers can be biased towards the detection of packers instead of actually learning the concept of a malicious binary.

We propose three approaches to mitigate the drawbacks we found:

1. OS loaders should be more aware of binary inconsistencies when loading files so as to avoid loading malformed binaries derived from binary data insertion.
2. ML models should focus more in the presence of malicious features instead on their frequency to be more resistant to data appendix.
3. We advocate for the assessment of malware variant-robustness in the evaluation of hereafter proposed ML-based malware detectors.

In summary, our contributions are as follows:

1. We describe our participation in a challenge to develop adversarial attacks against ML-based malware detectors.
2. We describe the model's weaknesses we found during our experiments.
3. We propose measures to the development of upcoming ML-based malware detection research work.

The remainder of the paper is organized as follows: in Section 3.1.3, we describe the challenge, datasets and models; in Section 3.1.4, we show the model's weaknesses by discussing our experiments and results; in Section 3.1.5, we present how we automated the exploitation; in Section 3.1.6, we show the weaknesses identified by us and pinpoint possible mitigation; in Section 3.1.7, we present the related work; finally, we draw our conclusions in Section 3.1.8.

3.1.3 The Challenge

The challenge hereby described (EndGame, 2019) is a competition run by an Internet company whose winner is the one to first achieve more points, up to 150. The challenge is composed of a total of 50 tasks, in which each one of the 50 distributed distinct binaries are classified by three distinct models. Each bypassed classifier for each binary accounts for 1 point. Two of the models are based on raw data classification and one is based on PE features. All models are based on static analysis feature extraction procedures. However, all submitted binaries are executed on a sandboxed environment and must produce the same Indicators of Compromise (IoCs) as the originally distributed binaries. Thus, our objective was to create adversarial malware that behave

the same way the original do by moving their binary sections and appending chunks of data (goodware bytes and strings) to them.

The competition has started in August/2019 and it is still taking place. We opted to report our findings even before its finish since we already investigated all samples and gained insights that we consider important to be shared with the community. During the competition, the scoreboard (one can check the current scoreboard online (MLSec, 2019)) has been reset many times by the organizers due to problems with the sandbox solution. After all resets, our team figured among the top-scorer participants. In spite of that, our main goal was not to win the competition but to investigate the drawbacks of a real ML model deployed for the classification of actual malware samples. We consider the investigation of third-part models more realistic than developing our own because these would be subject to our development biases.

The Dataset provided by the organizer is composed of 50 PE (Portable Executable (Microsoft, 2017b)) samples of varied malware families (grouping of malware based on their common characteristics (Levin et al., 2019)) for Microsoft Windows. The variety of samples aims to ensure a diversity of implementations so as the challenge also requires diversified approaches to bypass sample's detection. Figure 3.1 shows malware families distribution according to VirusTotal (Total, 2019) labels, normalized using AVClass (Sebastián et al., 2016). In total, there are 21 malware families in which four of them have at least four samples. **Emotet** is the most prevalent family in the dataset (5 samples) and is a banking trojan malware that steals sensitive and private information (such as credit card details) by downloading or dropping other banking malware, typically started by phishing emails and spread to other devices on the network after the infection (Drolet, 2019). **Loki** has 4 samples in the dataset and is a malware built to steal private data (such as stored passwords, login credentials and cryptocurrency wallets) and exfiltrate it to a C&C (Command and Control) host via `HTTP POST` (Malpedia, 2019). **Ramnit** also has 4 samples in the dataset and is a worm that can steal cookies, login credentials and files from the infected machine. It is also able to create a backdoor that allows the attackers to send all the data to the C&C server (Symantec, 2015). **Xtrat** also has 4 samples in the dataset and allows the attacker to interact with the victim via one or more C&C servers. Attackers are able to manage the infected machine's registry keys, files, process, servers, and also record content from any connected devices, such as webcameras and/or microphones (FireEye, 2014).

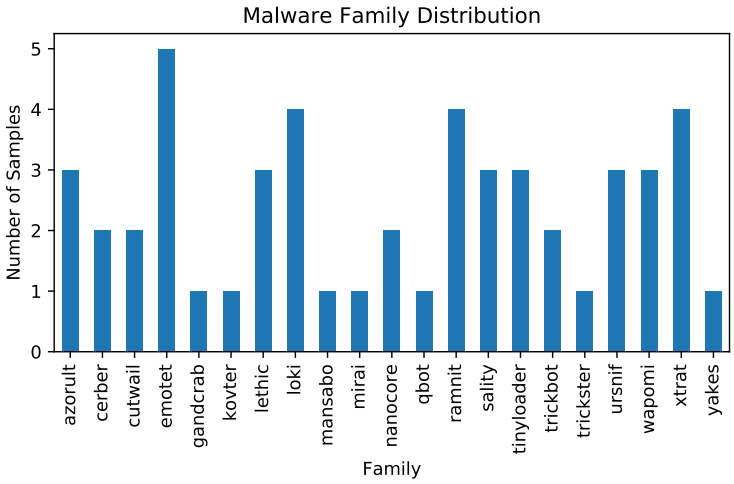


Figure 3.1: **Malware Family Distribution.** The dataset is composed of varied malware families, each one supposedly implemented in a distinct way, thus requiring distinct approaches to bypass their detection.

All samples are real malware samples that actually executed in sandboxed environments. We confirmed that by executing all samples in an available monitoring solution (Botacin et al.,

2018), as the challenge's organizers have not initially provided access to their sandbox's logs to the competitors.

The ML models provided by the organizers were: (i) `MalConv` (Raff et al., 2017); (ii) `Non-Negative MalConv` (Fleshman et al., 2018); and (iii) `LightGBM` (LightGBM, 2018). All models were trained using the `Ember` 2018 dataset (Anderson and Roth, 2018), which is composed of 1.1M binary files: 900K training samples and 200K testing samples. `MalConv` is an end-to-end deep learning model, which takes as input raw bytes of a file to determine its maliciousness. The model first creates a representation of the input using an 8-dimensional embedding (which takes as input tokenized bytes). The output of this embedding is then presented to a gated 1D convolution layer, with a filter width of 500 bytes, stride of 500 and 128 filters, followed by a fully connected layer of 128 units and a softmax output for each class. `Non-Negative MalConv` has an identical structure to `MalConv`, but it has only non-negative weights, forcing the model to look only for malicious evidences rather than looking for both malicious and benign ones. `LightGBM` is a gradient boosting decision tree which operates over a feature matrix. This matrix is created using hashing trick and histograms based on the inputted binary files characteristics, such as PE header information, file size, timestamp, imported libraries, strings, etc.

We noticed that the models which use raw data are very biased towards the detection of malware samples, which results in a high False Positive Rate (FPR) when handling benign data. Table 3.1 shows FPR for the 448 `exe` and 2422 `DLLs` of a pristine `Windows` installation.

Table 3.1: **False Positive Rate of native Windows files classification.** The raw models are very biased towards the detection of malware samples.

FileType	MalConv	Non-Neg. MalConv	LightGBM
EXEs	71.21%	87.72%	0.00%
DLLs	56.40%	80.55%	0.00%

3.1.4 Model's Weaknesses

We conducted a series of experiments to identify model's weaknesses before starting bypassing the models. In this section, we present the conducted experiments and discuss their results.

Appending random data to the binaries might be a successful strategy to bypass ML models based on raw binary inputs. We evaluated this hypothesis by repeatedly generating growing chunks of random data, up to the limit of 5MB defined by the challenge's organizers, and testing the resulting binaries' detection by all models.

Figure 3.2 presents the false negatives of the multiple models for the distinct random data chunks. We notice that some models, noticeably the original `Malconv`, are more susceptible to this strategy than others. It highlights the need of developing and evaluating distinct classifiers and features for the same task, since some might be more robust than others. We also discovered that the effects are more severe for data chunks greater than 1MB. This type of evaluation is important to understand the limits of the proposed solutions. Finally, we notice that the model based on the PE structure is the less affected by the appending of unrelated data.

Appending goodware strings to malware binaries might be a successful strategy to bypass classifiers based on features frequency. We evaluated this hypothesis by repeatedly retrieving strings presented by goodware files and appending them to the malware binaries before submitting them to all ML models.

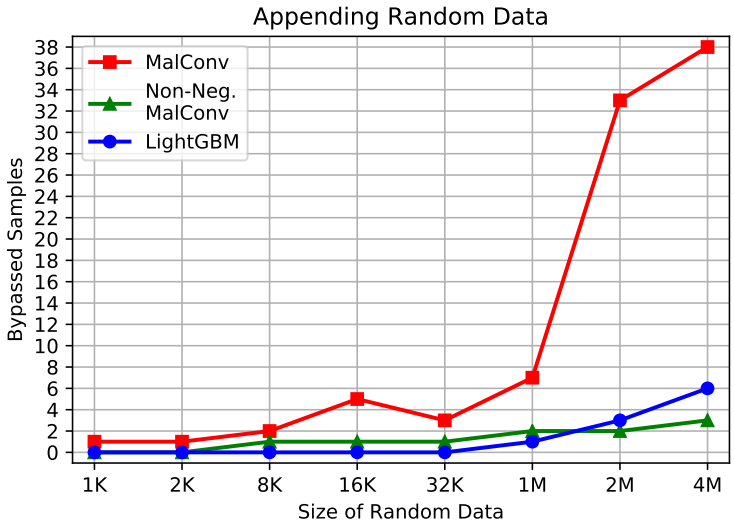


Figure 3.2: **Models FN after appending random data to malware binaries.** ML models based on raw data are susceptible to be evaded by this technique.

Figure 3.3 presents the FP rate of each ML model according to the number of appended goodware strings. We notice that as for the previous case, each model is affected by this technique in a distinct manner. However, all models are significantly affected when 10K+ strings are appended. This result holds true even for the model that also considers PE data.

Changing binary headers might be a successful strategy to bypass classifiers based on PE features. We evaluated this hypothesis by replacing some header fields of malware binaries with the values of header fields of goodware binaries. This replacement is enabled by the project decision took by Microsoft when implementing the Windows' binary loader: it silently ignores some PE binary fields (Ferrie, 2008), such as version numbers and checksums, thus allowing even corrupted binaries to run.

We replaced all binaries automatically by developing a `Python` script powered by the `PEFile` library (Erocarrera, 2019), although the modification could be manually performed by using any hexadecimal editor, such as `hteditor` (Biallas and Weyergraf, 2015).

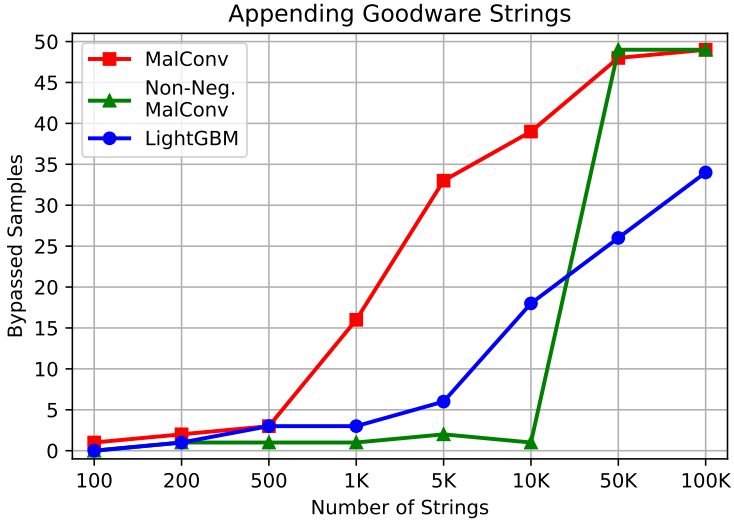


Figure 3.3: **Models FN after appending goodware strings to binaries.** All models are significantly affected by this technique.

Listing 3.1: PE header modification script. Modifying header values did not significantly increased the FP rate.

```
# open base gw
base_pe = pefile.PE(GWR_BASE)
# iterate over output samples, changing their PE HEADER
for m in adv_list:
   # open adversarial sample
   m_pe = pefile.PE(m)
   # update adversarial header
   m_pe.OPTIONAL_HEADER.MajorLinkerVersion = base_pe.OPTIONAL_HEADER.
      MajorLinkerVersion
   m_pe.OPTIONAL_HEADER.MinorLinkerVersion = base_pe.OPTIONAL_HEADER.
      MinorLinkerVersion
   m_pe.OPTIONAL_HEADER.CheckSum = base_pe.OPTIONAL_HEADER.CheckSum
   m_pe.OPTIONAL_HEADER.MajorOperatingSystemVersion = base_pe.
      OPTIONAL_HEADER.MajorOperatingSystemVersion
   m_pe.OPTIONAL_HEADER.MinorOperatingSystemVersion = base_pe.
      OPTIONAL_HEADER.MinorOperatingSystemVersion
   m_pe.OPTIONAL_HEADER.MajorImageVersion = base_pe.OPTIONAL_HEADER.
      MajorImageVersion
   m_pe.OPTIONAL_HEADER.MinorImageVersion = base_pe.OPTIONAL_HEADER.
      MinorImageVersion
   # write updated sample
   m_pe.write(m)
```

Code Snippet 3.1 illustrates the script operation and which fields were modified. This approach leads to the overall bypass of only six samples in all models. This result suggests that the model based on PE features learned other characteristics than the header values.

Packing and Unpacking samples with UPX, the popular open-source packer (Oberhumer et al., 2018), might be a successful strategy to bypass classifiers based on general PE features other than the headers. This approach might succeed on bypassing models as the UPX solution compresses entire PE into other PE sections, changing the external PE binary's aspect. We evaluated this hypothesis by packing and unpacking the provided binary samples.

The obtained results showed that classifiers were easily bypassed when strings were appended to the UPX-extracted payloads but not when directly appended to the UPX-packed payloads. This result suggested that the distributed model presented a bias against the UPX packer, since any file packed with UPX was classified as malicious. We evaluated this possibility in an exploratory fashion by randomly picking 150 UPX-packed and 150 non-packed samples from the `malshare` database (Malshare, 2019) and classifying them using the provided models.

Table 3.2: **UPX-Packed Samples Detection.** Results suggest that models might have a bias against UPX-packed samples.

Dataset	MalConv	Non-Neg MalConv	LightGBM
Originally Packed			
UPX	63.64%	55.37%	89.26%
Extracted UPX	59.50%	53.72%	66.12%
Originally Non-Packed			
Original	65.35%	54.77%	67.23%
UPX Packed	67.43%	56.43%	88.12%

Table 3.2 shows the overall detection results for the samples originally packed with UPX, their extracted version, the originally unpacked samples, and their packed version. We notice that the UPX-packed versions are more detected by all classifiers than the unpacked versions, thus suggesting that the classifiers might be biased towards the detection of UPX binaries, despite their content. In summary, although we were able to bypass one of the UPX-packed binaries provided by the challenge via packer extraction and string appendix, we decided to investigate additional approaches that operate in context for which the classifiers are less biased.

Packing samples with a distinct packer might be a successful strategy to bypass classifiers that present a bias against the popular UPX. We evaluated this hypothesis by packing the provided samples with `TeLock` (Telock, 2018). This strategy might succeed because this packer compresses and encrypts the original binary sections into a new one so as the original content cannot be identified by the classifiers. This strategy has proven to be effective on bypassing all models when combined with data appendix. However, some samples such as the ones from the `Extreme RAT` family (FireEye, 2014) do not execute properly when packed with this solution. Therefore, we opted to investigate more alternatives.

Embedding samples in a dropper (Kwon et al., 2015; Kang et al., 2007) might be a successful strategy to bypass classifiers and keep samples execution properly. This strategy might succeed because despite embedding the binary in a new section, this one is not encrypted nor compressed, avoiding unpacking issues. We evaluated this hypothesis by embedding the provided samples in the `Dr0p1t` dropper (D4Vinci, 2017). This approach along data appendix has proven to be able to bypass all detectors without breaking sample's execution. However, this solution generates binaries greater than 5MB, incompatible with the challenge's rules.

3.1.5 Automatic Exploitation

Table 3.3: **Models' confidence when classifying a malware (mw), a goodware (gw_i) and an adversarial malware (mw_i).** Results show that the adversarial sample is classified as goodware with higher confidence than a real goodware.

	Malware (mw)		Goodware (gw_i)		Adversarial Malware (mw+)	
Model	**Class**	**Confidence**	**Class**	**Confidence**	**Class**	**Confidence**
MalConv	Malware	99.99%	Goodware	69.54%	Goodware	81.22%
Non-Neg. MalConv	Malware	75.09%	Goodware	73.32%	Goodware	98.65%
LightGBM	Malware	100.00%	Goodware	99.99%	Goodware	99.97%
Average	Malware	91.69%	Goodware	80.95%	Goodware	93.28%

In this section, we present how we automated the exploitation of the weaknesses previously identified in the ML models and also present their impact on real AV detection engines.

Automating Models Exploitation. Once we discovered the major weaknesses of the provided models, we automated the procedure of generating malware variants able to bypass their detection.

In particular, our previous model's analyses revealed that:

1. Some samples, such as RATs, do not work well when data is directly appended.
2. `LightGBM` model detection is activated when unusual headers and sections are present.
3. `LightGBM` model can be bypassed by packing and/or embedding the original binary within a dropper with standard header and sections.

4. Appending data to packed and embedded code samples allows bypassing the `Malconv`-based models without negatively affecting the dropped code execution.

Based on these findings, we automated the process of packing/embedding all payloads within a new file, with standard header and sections, and after appending data by using a `Python` script. As the challenge limited the maximum file size to 5MB, we were not able to use existing solutions such as `Telock` (Telock, 2018) and `Dr0p1t` (D4Vinci, 2017) because they generated larger binary files. Instead, we implemented our own dropper.

We implemented the dropper by embedding the original malware sample as a PE binary resource (Microsoft, 2018). Code Snippet 3.2 illustrates the dropper operation. It (i) retrieves a pointer to the binary resource (line 3 to 5); (ii) creates a new file to drops the resource content (line 7); (iii) drop the entire content (line 8 to 10); and (iv) launches a process based on the dropped file (line 13). We were able to bypass all challenges without breaking samples execution by using this technique.

Listing 3.2: Malware Dropper. The original malware file is embedded as a PE binary resource at compile time and extracted in runtime.

```
1  int main(){
2        HMODULE h = GetModuleHandle(NULL);
3        HRSRC r = FindResource(h, ...);
4        HGLOBAL rc = LoadResource(h,r);
5        void* data = LockResource(rc);
6        DWORD size = SizeofResource(h,r);
7     FILE *f = fopen("dropped.exe","wb");
8     for(int i=0;i<size;i++){
9        unsigned char c1 = ((char*)data)[i];
10       fprintf(f,"%c",c1);
11    }
12    fclose(f);
13       CreateProcess("dropped.exe", ...);
```

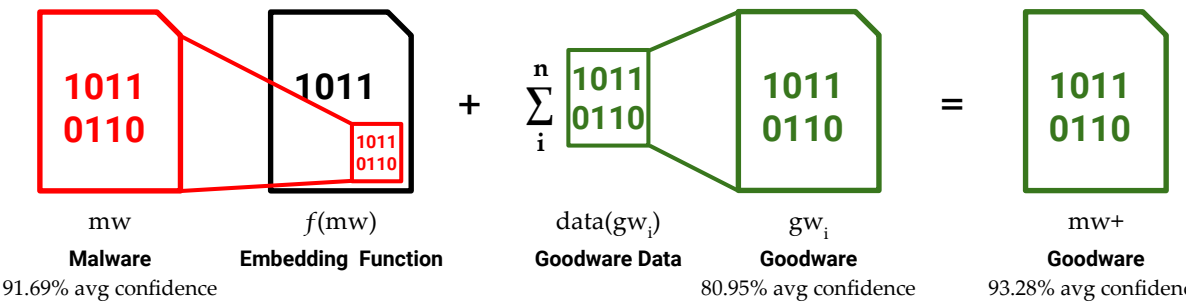


Figure 3.4: **Adversarial malware generation.** By just using an embedding function to add malware payloads within a new file and adding goodware data to it, such as strings and bytes from a set of goodware, we can change classifiers' output.

Figure 3.4 shows an overview of the variant generation process, which takes an original malware (mw) from the dataset to generate an adversarial malware ($mw+$), as defined by Equation 3.1. More specifically, the original malware (mw) is first used as input to an embedding function (f), which generates an entirely new file with standard PE headers and section definitions to host the original malware payload as a resource. We select one or more goodware gw_i from the set of all n goodware samples available, which, in our case, consists of all system files from a

pristine Windows installation. After that, we retrieve strings and/or bytes information of each one of the goodware samples via an extraction function ($data$). The extracted chunks $data(gw_i)$ are appended to the new file created using the function $f(mw)$ so as to ensure a bias towards the goodware class. The function outcome is an adversarial malware sample ($mw+$). One can iterate this procedure so as to consider multiple goodware samples, thus repeatedly appending data to the end of $f(mw)$. In the example shown in Table 3.3, the malware (mw), a sample from the family `xtrat` (sample number 35 on Table A.1), was classified by the three models as malware, with 99.99% of confidence by `malConv` and `LightGBM`, and with 75.05% of confidence by `Non-Negative Malconv`. The `ntoskrnl.exe` goodware (gw_i) used to produce the appending data was classified as goodware with 69.54% of confidence by `MalConv`, 73.32% by `Non-Negative MalConv` and 99.99% by `LightGBM`. The resulting adversarial sample ($mw+$) completely deceived all classifiers into considering it as being a goodware with 81.21%, 98.65% and 99.99% of confidence by `MalConv`, `Non-Negative MalConv` and `LightGBM`, respectively. On average, the confidence changed from 91.68% of being a malware (mw) to 93.28% of being a goodware ($mw+$).

$$mw+ = f(mw) + \sum_{i}^{n} data(gw_i) \tag{3.1}$$

Adversarial malware in real world. After we bypassed all challenge's models, we investigated whether the strategies we deployed for experimentation purposes could also be successfully leveraged in practice by actual attackers. To do so, we submitted the original and the modified samples to the VirusTotal service and retrieved the overall detection rates, as shown in Figure 3.5.

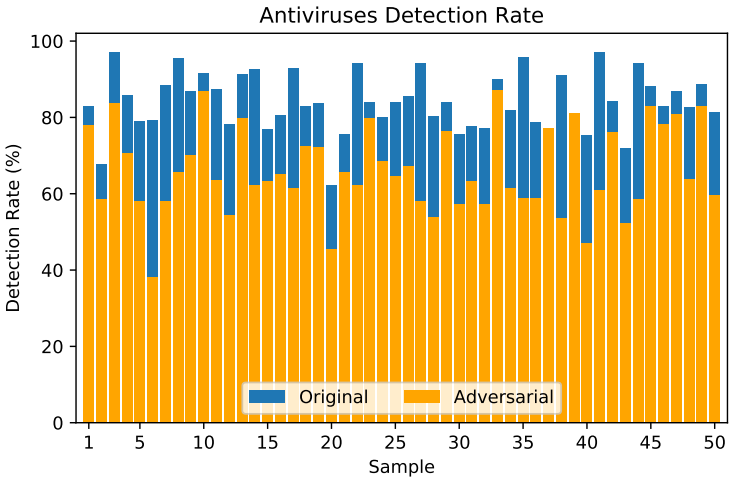


Figure 3.5: **Samples Detection Rate.** The developed malware variant samples were also less detected by the Virustotal's AV engines in addition to bypassing the challenge's models.

We noticed that our approach also affected real AV engines; the retrieved detection rates were smaller for all samples, in some cases (sample 6) dropping almost in half. This result is explained by current AV engines also being powered by ML models, which might be subject to the same weaknesses and biases that we identified for the challenge's models. This result highlights the need of developing more robust ML models and features for malware variants detection.

A practical drawback of generating adversarial malware samples is that their binaries become larger than the original ones due to additional data appended to the original malware. The appended data is not even used by the malware, but must be there to create a bias towards

goodware class. Figure 3.6 presents a comparison between original and ML-evasive samples per malware family. We discovered that adversarial malware (whose maximum size is around 5MB due to the limits imposed by the challenge organizers) are, in general, at least around twice the size of original ones (whose maximum size is around 1.5MB).

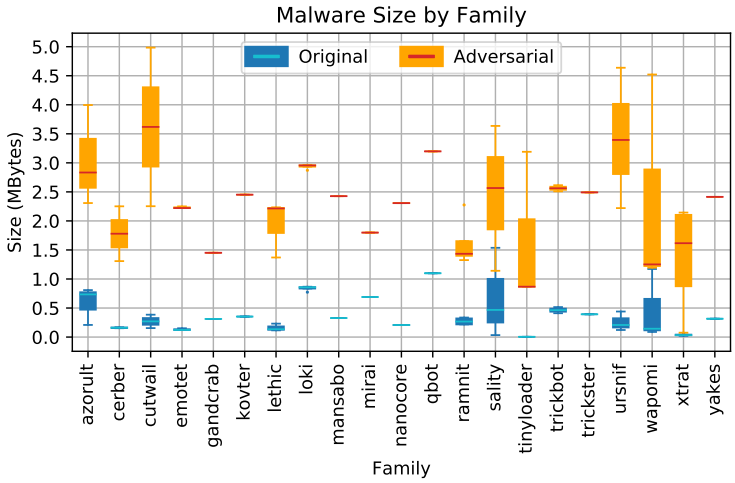


Figure 3.6: **Comparison between original and adversarial malware size in MBytes by malware family.** Adversarial samples are much bigger than the original ones due to the additional data used to bypass classifiers.

3.1.6 Discussion

Bypassing a detector means that one reasoned about an implementation and/or technology and identified weaknesses. In this section, we present the weaknesses identified by our team and pinpoint possible mitigation.

Susceptibility to appended data is a major weakness of raw models. In most cases, this simple strategy was enough to defeat the two raw data-based models, indicating that the concept learned by these models is not robust enough against adversarial attacks.

Appending data affects detection but not PE loading, since the Windows loader ignores some PE fields and even resolve conflicting sizes and checksums in runtime (Ferrie, 2008). This lax policy allows attackers to append content to the binaries without affecting its functionalities. We advocate for more strict loading policies so as to mitigate the impact of this type of bypass technique. For instance, the loader should check if a binary has more sections than declared in its header and/or if the section content exceeds the boundaries defined by the section size header field.

Adversarial malware are much bigger than original ones due to additional data appended to them, which are needed to bypass classifiers, such as strings and bytes. These data create a bias towards goodware class but also make their size greater, which can make it difficult for attackers to distribute them for new victims. We consider this as a challenge to be considered by any attacker, which needs to create an adversarial sample with the minimum size as possible.

Developing models based on the presence of features instead on their frequency is an strategy that should be adopted to mitigate the impact of appended data on classification models. We discovered that most classifiers changed their decision from malware to goodware when goodware strings were added to the binary, thus masking the impact of malware strings in the overall model. The use of pertinence models mitigate this effect as malicious strings need to be still present in the binaries to keep its functional.

Domain-specific models might present biases and not effectively learn a concept. We discovered that the model based on PE binaries features presented a bias against the UPX packer,

used by most malware but also present in benign software. Packing benign software with UPX revealed that the detector learned to mistakenly always flag UPX binaries as malicious.

Adopting malware variants robustness as a criteria to assess ML-based detectors is an essential step to moving forward the malware detection field. Our results revealed that even deep learning models might be easily bypassed, which makes them less effective in practice. We advocate for the adoption of variants robustness testing as a criteria for assessing the further developed malware classifiers so as to contribute for more real world-targeted solutions. This criteria might be included in the process of correct evaluating a malware detector, which already includes handling concept drift and evolution, class imbalance, degradation, etc (Ceschin et al., 2018; Pendlebury et al., 2019; Dasgupta, 2018).

Creating a robust representation is another essential step for malware detection field, given that attackers might include goodware characteristics into their malware to evade any model. Thus, creating a representation that is invariant to these characteristics is fundamental to avoid adversarial malware.

Checking file resources and embedded PE files should be part of ML feature extraction procedures. It would allow classifiers to detect embedded malicious payload instead of being easily deceived by malware droppers.

Converting samples into downloaders (Rossow et al., 2013) is also a successful strategy to bypass static detectors. In this case, the malicious payload is retrieved from the Internet whereas the undetected loader is submitted to ML scan. This shows that defenders should not only focus in the classifier accuracy rate but also should reason about the whole threat model to cover all attack possibilities. We implemented downloader versions of all malware samples provided by the challenge organizers, but we did not submit them to the challenge validation system because the challenge's sandbox solutions was network-isolated.

Generating adversarial samples to malware detector is a particular case of adversarial attacks, that can be performed against multiple domains and targets, such as image classifiers (Goodfellow et al., 2014b). Despite of having the same goal of bypassing a classification, different techniques are required depending on the problem domain. For some image classifiers, for example, adversarial images should look similar to the original ones (usually being indistinguishable to the human eye). In contrast, for malware detectors, adversarial malware only need to perform the same action as the original ones, even if they are completely different in their structures, as we did by embedding the original malware into a dropper. Therefore, simply adding a noise to a malware (as done in adversarial images (Goodfellow et al., 2014b)) might generate an invalid adversarial malware that does not work or does not perform the same action as the original one.

3.1.7 Related Work

In this section, we present related work that provide insights about malware evasion techniques and countermeasures. We believe that these related work might be useful in future community's research work, although they are not directly related to the described malware evasion challenge.

Adversarial Machine Learning are techniques aimed to defeat ML models. These techniques can rely on manual, heuristics or even ML-based strategies. A noticeable example of the latter are Generative Adversarial Networks (GANs). GANs are 2-part, coupled deep learning systems in which one part is trained to classify the inputs generated by the other. The two parts simultaneously try to maximize their performance. As a side effect, the input generator learns the best way to defeat its corresponding classifier. In this sense, GANs have been successfully used to create adversarial samples to bypass malware detectors (Hu and Tan, 2017). In some cases, even proposed defensive solutions, such as the use of defensive distillation, a procedure to train

deep neural networks that are more robust to perturbations (Papernot et al., 2015b), are prone to adversarial machine learning attacks (in this case, just by using different attack algorithms (Carlini and Wagner, 2017)). We believe that the use GANs will be the standard approach to solve future challenges such as the one that our team participated due to their classification power and scalability to large-scale datasets.

Binary Mutations were the so-far prevalent techniques to generate malware variants. These techniques implement transformations such as code replacement, instruction swapping, variable changes, dead code insertion and control flow obfuscation to bypass malware detectors (Borello and Mé, 2008). These techniques can also be applied In the context of this work. However, they are now targeting implicit neural-network models instead of the previously clear-defined behavioral (Christodorescu et al., 2005) or Intermediate Representation (IR)-defined (Shao and Smith, 2009) models.

Automated Binary Exploitation are techniques to automatically discover flaws and binaries and exploit them. These type of techniques have also been the subject of public challenges (e.g., DARPA challenge (Shoshitaishvili et al., 2016)). We believe that this type of technique can also be applied for malware detection evasion, despite being originally designed for a distinct purpose. The automatic identification of a critical execution path might indicate potential binary regions to be patched by an attacker.

3.1.8 Conclusion

In this paper, we reported the participation of our team in a public challenge launched by an Internet company in August/2019 to develop adversarial attacks against Machine Learning-based malware detectors. It challenged the participants to simultaneously bypass three distinct static analysis-based ML models. Our team developed custom binaries able to bypass all 150 challenges. We discovered that models leveraging raw binary data are easily evaded by appending additional data to the original binary files. We also discovered that models based on the Windows PE file structure learn malicious section names as suspicious, such that these detectors can be bypassed by replacing section names. We suggest the adoption of malware variant-resilience testing as an additional criteria for the evaluation and assessment of future developments of ML-based malware detectors. This ensures that the developed detectors can be applied to actual scenarios without the risk of being easily bypassed by attackers.

Reproducibility. The source code of the developed prototype to embed malware samples into unsuspicious binaries was released as open source and it is available on `github`: `https://github.com/marcusbotacin/Dropper`. All analysis reports of evasive and non-evasive samples execution and their similarities are available on the `corvus` platform: `https://corvus.inf.ufpr.br/`.

3.2 NO NEED TO TEACH NEW TRICKS TO OLD MALWARE: WINNING AN EVASION CHALLENGE WITH XOR-BASED ADVERSARIAL SAMPLES

This paper was published in the Proceedings of the 4th Reversing and Offensive-Oriented Trends Symposium (ROOTS), 2020 (Ceschin et al., 2020a).

Fabrício Ceschin[1], Marcus Botacin[1], Gabriel Lüders[1], Heitor Murilo Gomes[2], Luiz S. Oliveira[1], André Grégio[1]
[1]Federal University of Paraná, Brazil
{fjoceschin, mfbotacin, gl19, lesoliveira, gregio}@inf.ufpr.br
[2]University of Waikato, New Zealand
heitor.gomes@waikato.ac.nz

3.2.1 Abstract

Adversarial attacks to Machine Learning (ML) models became such a concern that tech companies (Microsoft and CUJO AI's Vulnerability Research Lab) decided to launch contests to better understand their impact on practice. During the contest's first edition (2019), participating teams were challenged to bypass three ML models in a white box manner. Our team bypassed all the three of them and reported interesting insights about models' weaknesses. In the second edition (2020), the challenge evolved to an attack-and-defense model: the teams should either propose defensive models and attack other teams' models in a black box manner. Despite the difficulty increase, our team was able to bypass all models again. In this paper, we describe our insights for this year's contest regarding on attacking models, as well defending them from adversarial attacks. In particular, we show how frequency-based models (e.g., TF-IDF) are vulnerable to the addition of dead function imports, and how models based on raw bytes are vulnerable to payload-embedding obfuscation (e.g., XOR and base64 encoding).

3.2.2 Introduction

Malicious programs have been a major security concern during the last four decades, with a plethora of proposed solutions over time to counter their threat. More recently, Machine Learning (ML) approaches have been widely applied to malware detection and classification. Although ML has significant advantages over other approaches, such as signature-based ones, it also has significant limitations (e.g., ML models are vulnerable to adversarial attacks).

In the malware context, adversarial attacks consist of modifying samples so as to disturb the classifier to the point of a malware sample being classified as a legitimate, non-malicious software. The field of adversarial attacks has been growing, both academically and industrially, since those attacks are increasingly common in practice.

Adversarial attacks aiming at ML models became so popular that tech companies decided to launch a contest to better understand their actual impact: The Machine Learning Security Evasion Competition (MLSEC). In this contest, the organizers provide working malware binaries to participants, in addition to classifiers able to detect all malware samples given. The participants are then challenged to transform the binaries of the provided samples in a way that those new malware bypasses the classifiers, while maintaining their same previous/original behavior when executed in a sandbox.

In the first contest edition (2019), the teams were challenged to bypass three ML models in a white box manner. Our team bypassed all models and reported interesting insights about

models' weaknesses. In the second edition (2020), the challenge evolved to an attack-and-defense model, with teams proposing defensive models, as well as attacking the models produced by other teams in a black box manner. Although the use of a black box approach certainly increased the challenge difficulty, our team was still able to bypass all models and, consequently, the first team to achieve maximum scoring in the context.

In this paper, we describe the experience gathered in the 2020's MLSEC contest, and the insights gained on both attacking ML models and defending them from adversarial attacks. On the one hand, our experience in the development of defensive models showed that the function distribution in 32-bit and 64-bit Windows libraries, and between their Debug and Release compilations are different, which affects detection. On the other hand, our experience in attacking models showed that embedding the malware payload into another binary eliminates most detection capabilities presented by the models.

In this year's contest, we were challenged to bypass three models. We discovered that: (i) the first model operates by looking into Portable Executable (PE) header features, thus being evaded by embedding malicious payloads in a dropper executable; (ii) the second model classifies function imports and libraries using a TF-IDF method, being evaded by the addition of fake imports to the dropper; (iii) the last model also checks for strings in the embedded content, being evaded by the encoding of the payload using XORing or base64 techniques.

One of the main contributions of this paper is to show that adversarial attacks are more practical in real life models than previously thought. Whereas there are approaches for adversarial attacks generation based on complex techniques, some even leveraging the same ML techniques used to defend against attacks (e.g., reinforcement learning (Filar, 2020)), we show that it is possible to generate attacks using known, simple techniques, such as XORing strings and encoding binaries in base64. Unlike automated attack generators, our approach is fully explainable and our insights can be used as feedback for the development of more robust models.

We highlight the impact of these techniques in practice by demonstrating that the detection rate of antiviruses (AV) from Virus Total decreased when the evasive samples were submitted to scanning (in comparison to the original malware), even though no specific AV was targeted in the competition. Since adversarial attacks pose a significant problem, we decided to make our attack and defense solutions available to the community, so anyone can train with it and design new security solutions. More specifically, we are releasing the source code of a dropper, and a Web interface in which users are able to generate adversarial malware using this dropper and test the transformed samples against multiple classifiers, including the those present in the previous challenge editions.

In summary, our contributions are as follows:

- We describe the experience in an ML-based malware detection evasion challenge.
- We describe our defensive ML model and discuss considerations to be made when developing a detection model.
- We present the attack techniques we leveraged in the contest to bypass all ML models.
- We discuss the impact of adversarial malware in practice via the detection rate of the evasive samples when inspected by real AVs.
- We release code and a platform for the development of experiments with adversarial malware.

This paper is organized as follows: in Section 3.2.3, we present the contest and our accomplished achievements there; in section 3.2.4, we discuss the reasons why defending from

adversarial attacks may be hard; in Section 3.2.5, we discuss our findings and how they provide insights and feedback for future security solutions; in Section 3.2.6f, we present the related work; finally, we draw our conclusions in Section 3.2.7.

3.2.3 The Challenge

We start this section presenting the contest rules, and the provided samples and models. Then, we detail how we managed to implement a defensive model. Finally, we show how we attack the models.

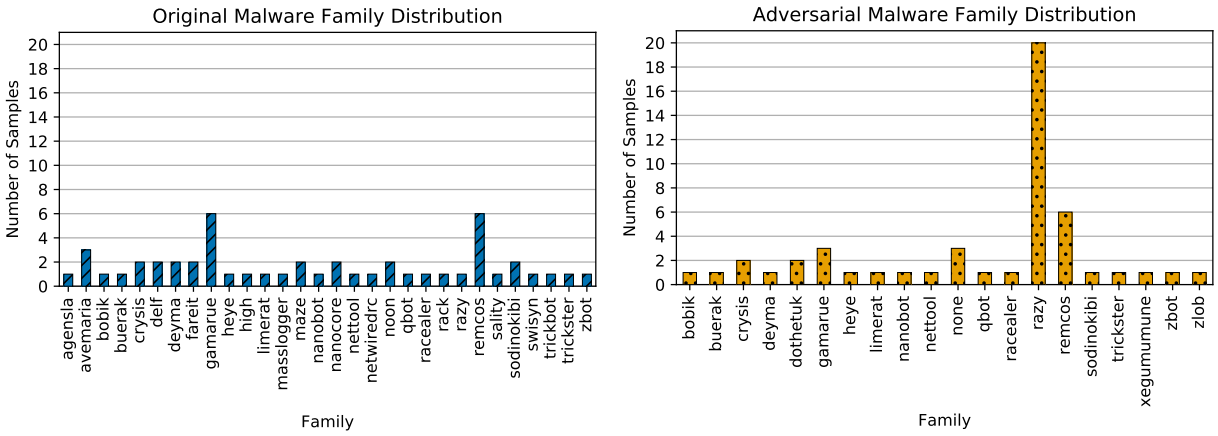
3.2.3.1 Definitions

In 2019, tech companies (Elastic, Endgame, MRG-Effitas, and VM-Ray) launched a contest challenging participants to bypass ML-based malware detectors with adversarial samples (Anderson, 2019). In this contest, the organizers provided participants with working malware binaries and malware classifiers (white box model) that initially detected all these malware samples. The participants were challenged to provide new binaries for the same malware samples, and those new malware should be able to bypass the classifiers, whereas still presenting the same original behavior when executed in a sandbox. Our team joined this challenge and we were able to generate adversarial samples that bypassed all the three models. Our findings were reported in a previous paper (Ceschin et al., 2019).

In 2020, other tech companies (Microsoft and CUJO AI's Vulnerability Research Lab) joined the mission of exercising adversarial attacks in practice. That second edition of the Machine Learning Security Evasion Competition (MLSEC) (Balazs, 2020) was an incremental version of its previous edition. In this year's competition, the organizers added one extra step: the generation of defensive solutions to be further attacked by the participants.

In the "defender's challenge", the participants were free to develop their own machine learning defensive solutions, with models of their own choice and trained using any dataset. The entire defensive solution should be saved by the participants in a docker image, which was then tested using an unknown dataset with three requirements: (i) the model should accomplish less than 1% of False Positive Rate (FPR), (ii) less than 10% of False Negative Rate (FNR), and (iii) it must return a response within 5 seconds for any presented sample. In total, three models met the proposed requirements: `ember` (provided by the organizers), `needforspeed` (our model), and `domumpqb` (provided by another team that published their solution after the final results (Quiring et al., 2020)). We do not know how many teams participated in this step of the challenge, apart from the fact that only those aforementioned models have met the requirements. Furthermore, the results of this part of the challenge would depend on how the attackers perform against each model, i.e., the one that performs better against adversarial attacks wins.

In the "attacker's challenge", all models that achieved the previous requirements were made available to be attacked by further black box attacks. Thus, the attackers would have access only to the output produced by those models, without directly accessing them (Guo et al., 2019). Black box attacks were conducted with 50 unique Windows malware samples provided by the contest organizers, which should be modified in order to bypass their detection in the defense solutions. Each new sample produced by the participants should have its behavior identical to the original malware sample from which it was based on. Behavioral validation is accomplished based on running the modified sample in a sandbox made available in the Web site of the contest, which should result in the same Indicators of Compromise (IoCs) of the original ones. Thus, each bypassed classifier for each binary accounts for 1 point, summing up to 150 points. Moreover, as a tiebreaker rule in case of similar bypass scores, the competition also stores the number of ML



(a) **Original malware family distribution.** In total, 30 different families were present in the samples provided this year, 9 more than last year (21 families).

(b) **Adversarial malware family distribution.** From the 30 original families, our samples were re-classified into only 19 ones, and 3 of them could not be classified at all.

Figure 3.7: **Malware families distribution.** Differences between original malware samples and adversarial ones are notable.

queries (number of times that samples are tested) used by each participant. Therefore, the team that achieves the lowest number of queries wins.

We submitted the 50 samples provided by the organizers to the Virus Total API and then used AVClass (Sebastián et al., 2016) to normalize the resulting labels. In Figure 3.7(a), we show the number of samples distributed in malware families, with two families with higher prevalence (`gamarue` and `remcos`) and 30 families in total. Notice that all malware available for this challenge consist of real samples and have already been seen in the wild, such as the family that steal crypto-currency from its victims (Bisson, 2019). The `gamarue` family can give a malicious hacker control of the PC, stealing sensitive information and changing security settings (Microsoft Security Intelligence, 2017). In addition, the `remcos` family embed a XML code that allows for any binary with parameters to be executed, in this case a REMCOS RAT, which gives the attackers full control over the infected PC, allowing them to run keyloggers and surveillance tools (Malwarebytes Labs, 2017). Thus, despite of being just a competition, all the malicious samples presented in this work may present real risks.

3.2.3.2 Defenders challenge

To develop our defense solution, we selected a model developed by our research team that achieved good metrics using textual features (TF-IDF) on top of static analysis and Random Forest classifier (Ceschin et al., 2018) as **baseline**. We considered this challenge as an opportunity to test our research model against adversarial attacks in practice.

Initially, we considered using our baseline model as it is originally presented in our paper (Ceschin et al., 2018), with malware samples collected in the Brazilian cyberspace as the training set. Our choice is usually enough to reach good result metrics (almost 98% of f1-score with a low false-negative rate). When we tested our baseline model against the samples provided by the organization in the 2019 edition of the competition, the results were surprisingly bad and totally different from what we expected them to be. We hypothesized that those results might be biased due to the distinct characteristics of the samples considered by the organizers in contrast to ours. More specifically, since the threat landscape in Brazil is very different from the rest of the world (Botacin et al., 2019), we hypothesized that the classifiers we used in Brazilian

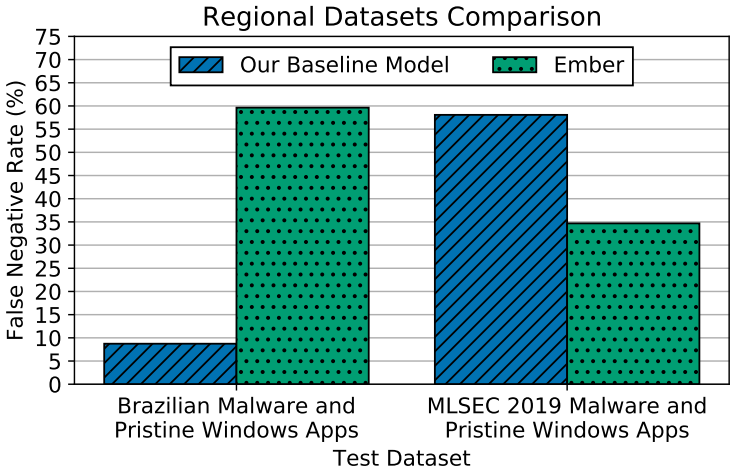


Figure 3.8: **Regional datasets and models.** Each model performs better in their own region, indicating that detectors must be specially crafted for a given region.

cyberspace were not the most suitable ones for classifying the likely-global samples provided by the organization.

To validate this hypothesis, we decided to retrain the classifier using the `ember` dataset as input, similar to the organizers' approach to train their LightGBM model (Anderson and Roth, 2018). When we compared the results of training our model by testing it with the Brazilian and the Ember samples, we noticed that the latter indeed generalize better. In Figure 3.8, we show the False-Negative Rate (FNR) of the two versions of our model (trained with Brazilian and Ember samples, respectively) after evaluation against a subset of these same datasets. We noticed that each classifier works marginally well in their own region, in a way that they could detect malware without too many false-positives, i.e., the model trained with global samples presents high detection rate, but performs extremely bad with Brazilian samples, and vice-versa.

Consequently, if these classifiers were applied in actual scenarios as a generalization of global threats, they would let many threats originated from other regions than the ones they were trained to be executed without raising warnings. Thus, we conclude that ML models must be specially crafted for each region in which they are going to operate, given that they may detect more samples and be more effective. For the remainder of this paper, we refer to our model as the one trained with EMBER, since that was the more suitable dataset for the task at hand. It is worth to mention that the competition organizers also made all the adversarial samples from last year challenge available, but we decided to not use them in the training step, so we could use them to verify our model's robustness in the testing step.

To create our definitive model for the competition, we selected the attributes from the EMBER datasets (Anderson and Roth, 2018) (both 2017 and 2018 version) we believed to be less prone to be affected by adversaries, according to our previous experience. We categorized the attributes in three types: numerical, which are integer or float numbers; categorical, which represents categories; and textual, which are a set of strings. To train our model, we used the EMBER's 1.6 million labeled samples as input to the scikit-learn Random Forest (scikit learn, 2020c) with 100 estimators. Below, we list the attributes we selected for our model. The detailed description of all of them is available in the EMBER's dataset paper and source code (Anderson and Roth, 2018).

1. Numerical
    - string_paths

- string_urls
- string_registry
- string_MZ
- virtual_size
- has_debug
- imports
- exports
- has_relocations
- has_resources
- has_signature
- has_tls
- symbols
- timestamp
- numberof_sections
- major_image_version
- minor_image_version
- major_linker_version
- minor_linker_version
- major_operating_system_version
- minor_operating_system_version
- major_subsystem_version
- minor_subsystem_version
- sizeof_code
- sizeof_headers
- sizeof_heap_commit

2. Categorical
    - machine
    - magic
3. Textual
    - libraries
    - functions
    - exports_list
    - dll_characteristics_list
    - characteristics_list

The categorical attributes `machine` and `magic` were obtained from the PE header and transformed into one-hot encoding array (scikit learn, 2020b) where each binary column represent a value (1 for present, 0 otherwise). The textual features, such as libraries and functions used by a software, were also obtained from the PE header and transformed into texts, separated by spaces. These texts were used as input to TF-IDF (scikit learn, 2020d), transforming every text into a sparse array containing the TF-IDF values for the 300 top words in all the texts created (the ones with most frequency). The numerical features were all extracted from the PE header, except `string_path`, `string_urls`, `string_registry`, and `string_MZ`, which extracts the number of strings that contains a system paths, URLs, registries, and MZ headers, respectively. Finally, after we transformed all attributes into numerical features, we normalized them using MinMaxScaler (scikit learn, 2020a) to use as input of our model. Given that EMBER dataset is available in CSV format (with all the attributes already extracted), we also created a module that extracts the very same attributes from raw PE binaries. This module is used to test attackers' samples, and it also could be used by any real-world testing solution.

"*The sad state of PE parsing*" (Part 1): The lack of a standard and complete library for PE parsing and manipulation is a long-term complaint (lucasg, 2017) and this affected the development of both the defensive and the attacking solutions. In the first case, here reported, our original solution was deployed on top of `PEfile` (Erocarrera, 2019), but it did not achieve the required performance by the challenge, resulting in classification timeout (some samples were taking more than 5 seconds to be parsed). We then ported our implementation to `lief` (Quarkslab, 2019), since this was the tool used for binary parsing in the contest demonstration script. `Lief` is in fact much faster than `PEfile` (the same aforementioned samples were taking about 2 seconds when using `lief` as parser), but its parsing results are a bit different. Due to this fact, we had to slightly change our model to consider some features as categorical instead of numerical, since fields such as `machine` and `magic`, which are parsed as integer numbers by `PEfile`, are represented by strings (flags) in `lief` (it ended up not affecting the results of our classifier but it could).

To fine tune our model, we created a new prediction function that uses the model class probabilities as input. To do so, we defined a threshold T and used it to define the output class: if the probability of being a "goodware" is greater than T, the current sample will be classified as a goodware. Otherwise, it will be classified as a malware. It was required to make our classifier perform as required by the competition, achieving less than 0.1% of FPR with a threshold $T = 80\%$. This technique has proven to be much better than using the default Random Forest prediction function, which achieved a FPR of 8.5%.

Our model vs. last year adversaries: The initial test of our model consisted of submitting the adversarial samples provided by the organizers from last year's challenge to it, and then analyzing the resulting detection rate. In total, there a were 594 samples, all of them variations of the 50 original samples from last year's challenge. Our model was able to detect 88.91% of the samples. Considering that all 2019 models were bypassed by those samples, we have agreed that this was a significant good result. It also confirmed our findings from the previous challenge, i.e., that models based on parsing PE files are better than the ones that make use of raw data (Ceschin et al., 2019).

3.2.3.3 Attackers challenge

We started our attacks by trying to replicate the strategy leveraged in the previous year (2019) with this year's (2020) classifiers and samples. Thus, we first appended goodware strings and random bytes to the original samples. This strategy resulted in 44 points, with 36 samples bypassing

ember, 8 bypassing needforspeed, and none of them bypassing domumpqb. These results show that this year's models were really stronger than the previous ones.

Then, we moved to the next strategy that was successful last year: embedding the original sample in a "Dropper", a new binary that embeds the original malware sample as a resource, writes it to a file in runtime, and launches it from there. In most cases, the samples were executed directly, as they were typical PE files. In one of the cases, the original malware sample was a DLL. Thus, we modified our Dropper from last year to inject this DLL into a host process. As this DLL did not export any function, we launched its main function, invoking it from its ordinal number (rundll32 dll_name,#1). This approach succeeded on fully bypassing the first model (ember), which was based on PE headers. However, while this step was the final one in the last year's challenge, now we just accomplished the first third of the challenge.

Despite bypassing the first detector, the dropper malware was not able to fully bypass the other detectors. We then focused our attention in bypassing our own model, since we could leverage previous knowledge on the attack. As our model is based on the library imports and their respective functions, we guessed that our model was detecting the dropper as malicious. This might be happening, for instance, due to the presence of a function such as FindResource, which is largely used by malware droppers (and also by a few benign applications). Our first thought was to hide the FindResource API calls from the classifier. To do so, we tried to compress our samples with Telock (Telock, 2018), PELock (PELock, 2016), and Themida (Technologies, 2011) packers. Interestingly, reducing the number of imports only increased the confidence on the malware label, which reinforces our last year's claim that most classifiers learn packers as malicious feature regardless of the binary content. This phenomenon was also reported to happen for real AVs (Aghakhani et al., 2020).

The remaining alternative to bypass this model was to search for some benign sample likely used to test the model that present the same imports. Interestingly, the Calculator (calc.exe) presents these characteristics, importing a series of functions, including FindResource, and was report as benign with 100% of confidence level by our classifier. Thus, our goal turned into building a new dropper binary mimicking the calculator.

Listing 3.3: **Lief script example.** Automatic Section and Function Inclusion.

```
import lief
# Parse
gw = lief.parse(GOODWARE)
mw = lief.parse(MALWARE)
# Get Sections
gw_sections = [s for s in gw.sections]
mw_sections = [s for s in mw.sections]
# Add Missing Sections
sec_diff = len(gw_sections) - len(mw_sections)
for i in range(1,sec_diff):
  mw.add_section(gw_sections[i])
for lib in gw.imports:
  lib_name = lib.name
  # Add Missing Libs
  if lib_name not in mw.libraries:
    mw.add_library(lib_name)
    # Add Missing Functions for the Lib
  for func in lib.entries:
    func_name = func.name
    if func_name != '':
      if func_name not in [f.name for f in mw.imported_functions]:
        mw.add_import_function(lib_name,func.name)
# Build New Binary
builder = lief.PE.Builder(mw)
builder.build_imports(True)
builder.patch_imports(True)
builder.build()
builder.write(NEW_MALWARE)
```

Listing 3.4: **Dead code insertion.** These functions play no role in the Dropper execution.

```
void dead()
{
  ShellMessageBox(NULL,NULL,NULL,NULL,NULL);
  RegEnumKeyExW(NULL,NULL,NULL,NULL,NULL,NULL,NULL);
  BSTR_UserFree(NULL,NULL);
  CoInitialize(NULL);
  IsThemeActive();
  ...
}
```

"*The sad state of PE parsing*" (Part 2): The lack of a standard library for PE manipulation also affected the development of the adversarial samples. `PEfile` has no native support for section inclusions and whereas it can be extended for this task (Cheron, 2017), the whole process is manual and laborious. In turn, the `lief` solution has native methods for adding sections and even function imports. We implemented a code on top of it to perform this task, as shown in Code 3.3. Unfortunately, `lief` has some known issues (stevielavern, 2017) regarding these functions and it ended up breaking all malware binaries (although it worked with the simplest test cases we developed).

Since the existing solutions did not allow us to patch the compiled binaries, we opted to compile the Dropper with the extra function. However, as these functions do not play any role regarding the Dropper's operation, we added them as dead code into a never called function, as shown in Code 3.4. We compiled the code without optimization, since these functions are not

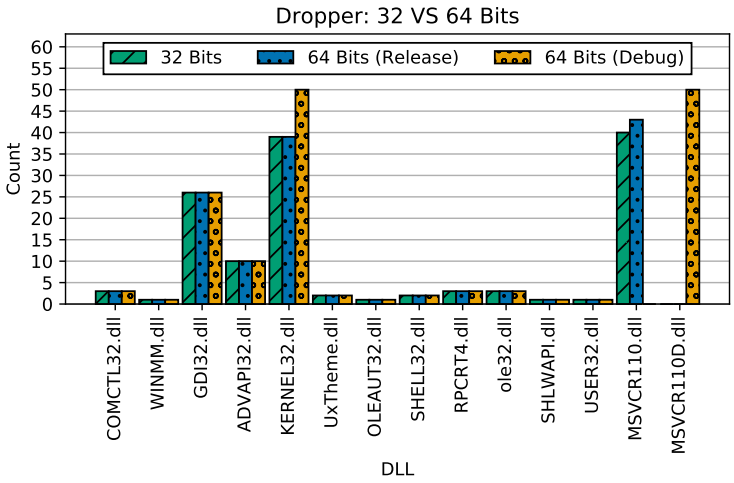


Figure 3.9: **Number of functions in each library.** Compiling libraries for 32-bit and 64-bit systems and in the Debug or in the Release mode affect their distribution.

eliminated. Therefore, their imports were available to the ML models, but they did not affect the Dropper execution. This strategy resulted in the complete bypass of our model.

It is important to notice that the correct working of this approach depends on the complete mimicry of all libraries and functions, which includes compiling the code for the same architecture and linking it with the same libraries as the goodware binary. Otherwise, the sample could still be detected due to the combination of the extracted functions and libraries.

In Figure 3.9, we show the impact of multiple compilation settings on the import of the same functions. We observed that changing the compilation from 32 to 64 bits results in the addition of three additional imports to the `MSVCR110.dll`. Even worse, compiling the code in the Debug mode results in the linkage of a distinct library `MSVCR110D.dll`—the debug version—with a distinct number of function imports. Since we considered the Calculator binary as goodware, and it was compiled with Release configurations and for the 64-bit architecture, all dropper binaries were configured the same way. The evasion was only possible using this setting.

Black box is harder, but not impossible. So far, we have bypassed two of the three models. Although we performed the black box tests, we have some degree of previous knowledge about the models because they were either deployed in the last year's contest, or developed by us. To bypass the last model, we had to deploy a full black box attack.

At this point, a few samples had already bypassed the third model (21 samples). Since all droppers were similar by construction, we hypothesized that the third model was detecting some part of the embedded payload, as it was the only part allowed to change from one compilation to another. Therefore, we should hide the embedded payload to bypass this detector. We tested two approaches for this task: (i) encoding the malware binary as a base64 string; and (ii) XORing the malware binary with a key. In the former case, the resource must be decoded before being written in disk, whereas in the latter, the same XOR key can be used to decode the buffer before it is dumped to the file system.

These strategies are enough to bypass most of the verification performed by the model, as it hides the original strings and magic numbers. For instance, the `MZ` flag of the embedded payload is not present anymore in the resource section, thus not being identified by mechanisms that look for embedded files (as is the case of our own model). In addition, XORing strings often result in non-printable characters that are not handled by the `strings` utility. By combining these strategies, we were able to fully bypass the third model. Thus, despite requiring additional reasoning and implementation efforts, the black box bypass of ML detectors is completely viable.

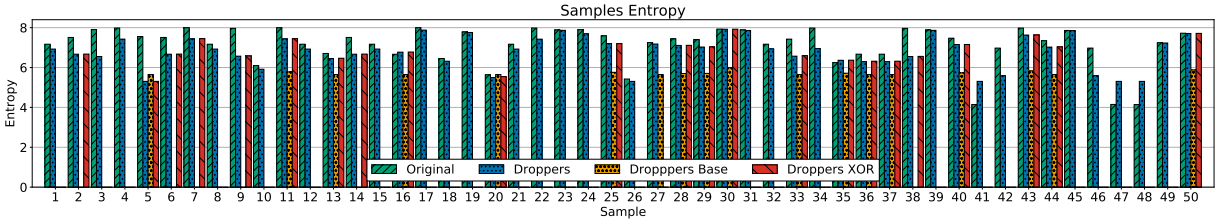


Figure 3.10: **Sample's maximum section entropy.** Embedding the original malware samples into binary droppers did not generate sections with greater entropy values.

Table 3.4: **Average number of queries.** We bypassed all models with an average rate lower than 5 queries per sample.

Team	Bypasses	Queries	Average
Ours	150	741	4.94
2nd	47	162	3.44
3rd	44	150	3,40
4th	1	78	78

It is worth to emphasize the robustness of this approach, despite its relative simplicity. We first hypothesized that the dropping approach could be detected by a model that considers section entropy values as a feature (e.g., histogram of section's entropy), because packing, compression, and embedding often result in entropy increase (Ugarte-Pedrero et al., 2015). In practice, this effect was not observed. In Figure 3.10, we show the section maximum entropy values for the original samples and for the multiple dropper variations. We observe that most of the dropper's section maximum entropy values are equal to the ones of the original samples (in most cases, these values were already high). In some cases, noticeably when using base64, the values are even lower than the original samples. Therefore, the embedding of content in the droppers would not be detectable in an indistinguishable manner when compared to the high entropy of the original binaries.

In Table 3.4, we show the total and average number of queries performed until breaking all the models. On average, it took us less than 5 queries per sample to bypass the three models. We consider this number very low, even when considering that we had previous knowledge about some models, which can also be expected from a skilled, motivated attacker performing targeted attacks against real systems. Worse than that, if these results hold true for an actual security solution, it is plausible to hypothesize that five attempts is even below the threshold of a typical Intrusion Detection System (IDS), thus an intrusion could occur unnoticeable.

To better demonstrate the impact of adversarial attacks over real systems, we submitted samples to the Virus Total service (Total, 2019) and compared the detection rate of the original and the evasive samples. Figure 3.11 shows that the AVs were also affected by the samples' modifications, even though the challenge originally did not target any AV. This happens because (i) hiding the payload from the ML models also hides them from the AV scanners; and (ii) the ML models used by the AVs are also affected by the changes in the binaries features introduced by the malware dropper. This latter phenomenon can be clearly observed in practice if we focus on the detection labels provided by the AVs that claim to use ML (CrowdStrike, 2020; BlackBerry - Cylance, 2020; Cynet, 2020; Banon, 2020; Palo Alto Networks, 2020), as shown in . Table 3.5 for the samples that less (22) and most (27) affected AV detection. We highlight that even the samples which less affected AVs bypassed at least one ML-based malware detection solution.

The impact of hiding the payload from the AV can also be observed in the assigned labels, as shown in Figure 3.7(b). We noticed that the labels assigned to the adversarial samples

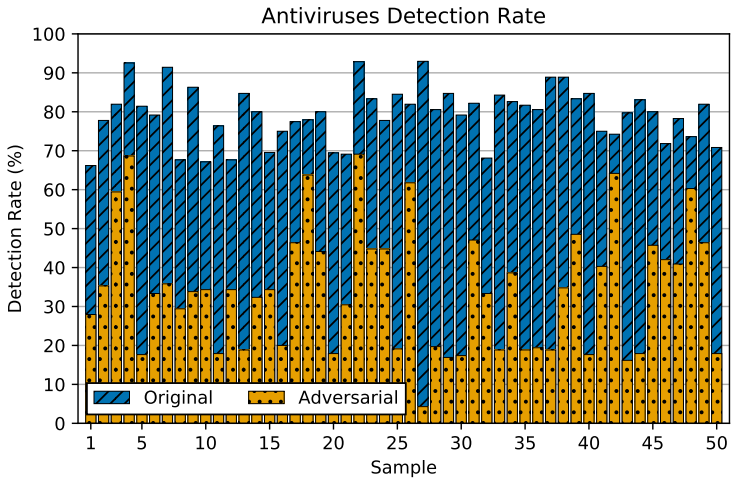


Figure 3.11: **Detection rate of AVs.** Real AVs were also affected by our deployed evasion techniques.

Table 3.5: **ML and AntiVirus.** AVs that claim to use ML are also affected by our adversarial malware samples.

Sample	Version	AntiVirus				
		CrowdStrike (CrowdStrike, 2020)	Cylance (BlackBerry - Cylance, 2020)	Cynet (Cynet, 2020)	Elastic (Banon, 2020)	Paloalto (Palo Alto Networks, 2020)
22	Original	True (100%)	True	True (100%)	True (high confidence)	True
	Adversarial	True (60%)	True	False	False	False
27	Original	True (100%)	True	True (100%)	True (high confidence)	True
	Adversarial	False	False	False	True	False

were significantly different from those assigned to the original samples, which suggests that they were detected using distinct rules, heuristics, patterns etc. In our case, the majority of the samples (20) were turned into `razy` family, which consists in malware that attack browser extensions to steal crypto-currency (Bisson, 2019). We believe that this phenomenon might be related to the fact that many `razy` samples are distributed in the form of a dropper.

A side-effect of embedding the payloads into a dropper is that the dropper binaries become similar, as they share the same headers, instructions, libraries, and so on. In Figure 3.12, we present the samples similarity according to `ssdeep` (ssdeep, 2017)'s scores. It shows that embedding the original malware samples into dropper binaries increased the number of samples reported as similar, even though reducing the relative frequency of very similar sample's scores. The first effect occurs because the dropper's similarities are identified by the similarity matching solution. In turn, the second effect occurs because the similar bytes between two binaries are "diluted" among the dropper's bytes, thus reducing the similarity score.

We believe that the first phenomenon might provide an interesting mechanism for the detection of the adversarial malware. If similarity scores were considered by a security solution in addition to the ML model's verdicts, the solution could be able to use it to correlate a detected dropper with an evasive sample. From the attacker's perspective, it would be now required to bypass all ML models all the time, instead of risking that a single similar, detected dropper raises a detection warning.

3.2.4 Why Defending is Harder?

A frequent question related to the bypass of ML models is why classifiers did not detect the adversarial samples. In the hereby reported case, the adversarial samples we produced in the attackers challenge were not detected by any model because we made them look like a goodware application (the Calculator), as shown in the previous section. In face of this scenario, it is common to hear proposals for training the models with adversarial samples so as to harden them

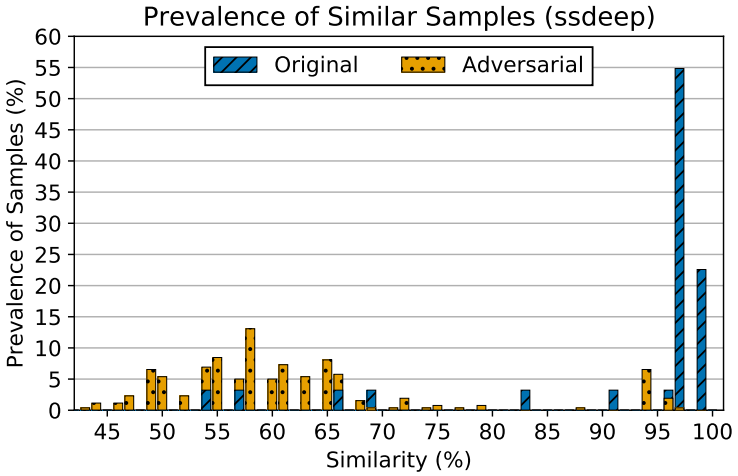


Figure 3.12: **Sample's similarity.** Encoding the payload reduces the sample's similarity score.

against evasion. If this were effective, it would increase the security coverage of most security solutions. For instance, AVs would be allowed to update their models to detect adversarial samples as soon as some previously undetected samples were uncovered.

To test this hypothesis, we compared our original model—the one submitted to the competition—with a new one, trained with the same EMBER datasets, but including the 594 MLSEC 2019 adversarial samples provided by the organizers. Then, we tested these models with two datasets: one containing the adversaries we developed, and another containing the original samples. Both of them included the same pristine Windows applications as the "goodware" set.

In Table 3.6, we present the results for the multiple training and test sets combinations. On the one hand, when we trained our model using only EMBER datasets, it detected all the competition original malware samples ($FNR = 0\%$) and correctly labelled almost all the goodware samples ($FPR = 0.1\%$). However, the model was unable to detect our developed adversarial malware ($FNR = 100\%$). On the other hand, when we trained our model with last year's' adversarial samples in addition to the EMBER datasets, it was able to detect all malware samples from both original and adversarial datasets ($FNR = 0\$$). However, the accomplished results incurred in a very high rate of false-positives ($FPR = 78.54\%$), i.e., our model started to recognize the majority of goodware as a malware.

The presented results indicate that the adversarial malware samples created by our attack are very difficult to distinguish apart from goodware, even if we use an updated ML model. This happens because they present the same features of goodware applications. Therefore, our updated model started to consider plenty of goodware samples as malware.

In practice, the problem of detecting adversarial samples might be even more complicated. AV companies will not have all adversarial samples at once, but will collect them over time as soon as they are uncovered. Hence, the problem of adversarial attacks mixes with the concept drift problem (Ceschin et al., 2018), when samples context shifts from time T to time $T + 1$. In our particular scenario, the malware samples evolve to prevent being detected. Whereas there are existing approaches to detect concept drift, these might face difficulties to handle adversarial samples, On the one hand, approaches that consider all the models at once (Xu et al., 2019) might be vulnerable to poisoning (Taheri et al., 2019), which fall backs to the aforementioned scenario. On the other hand, approaches that perform partial retraining (e.g., ADWIN (Bifet and Gavaldà, 2007), DDM (Gama et al., 2014a), and EDDM (Baena-Garcıa et al., 2006)) could increase the FP rate, since they discard goodware concepts to reduce the classifier's confusion

Table 3.6: **Original vs updated model.** Training with adversarial malware does not help to improve classification performance.

Train Dataset	Test Dataset	False Negative Rate (FNR)	False Positive Rate (FPR)
EMBER only	MLSEC 2020 Original Samples and Pristine Windows Apps	0%	0.1%
EMBER and MLSEC 2019 Adversaries		0%	78.54%
EMBER only	Our MLSEC 2020 Adversaries and Pristine Windows Apps	100%	0.1%
EMBER and MLSEC 2019 Adversaries		0%	78.54%

between the classes. Therefore, there is a long path towards the development of effective and real-world solutions to handle adversarial malware attacks.

3.2.5 Discussion

In this section, we revisit our findings after beating the MLSEC 2020, and discuss their implications for ML-based anti-malware.

Increasing ML Robustness. Our results suggest that ML-based malware detectors must be more robust against adversarial attacks to be practical, effective defenses. This type of detector should consider the great variety of malware samples, which may be distributed by attackers with simple modifications on them and, at the same time, exhibit the intended behavior/malicious actions. Despite being more resistant to attackers than previous year's models, all the models submitted in this year's competition were easily evaded. This indicates that the selected features are not robust enough to ensure a good detection model. The problem persisted even when we trained a new classifier with similar adversarial samples. Therefore, the investigation of new, more robust ways to represent malware is still a long path to be followed in future research work.

Explainable Attacks & Defenses. Although there are many existing automated approaches for adversarial attacks, we hereby presented attacks to ML models based on the attacker's knowledge about the models and binary implementations. This approach is more laborious, but it has the significant advantage of providing feedback information for the development of the next generation of security solutions. In an analogy, deep learning models are often criticized for not being explainable, despite being effective. We here extend the criticism to automated attacks, which are not explainable, despite their effectiveness. We believe that knowing when an ML model fails is extremely important to understand how to correct it.

Adversarial Attacks for the Masses. Our results show that adversarial attacks really happen, and they are effective. Thus, we claim that these attacks must be taken into account in threat models, training, and experiments. To encourage this practice, we are publicly releasing the code of our dropper to the community, so anyone may be able to practice with it (and improve it), as well as to consider adversarial attacks in their own research. Moreover, we are also making available a Web-based, automated solution to generate adversarial samples based on files uploaded by the user. Each submitted file is checked against multiple ML classifiers, including

the ones from 2019's challenge and our classifier for the 2020's challenge. With that, we hope to allow users in checking the robustness of multiple models and the viability of attacking them.

Feedback for Future Work & Arms-Race. We believe that our findings might provide valuable feedback for the development of the next-generation security solutions. We discuss some insights that might lead to ML models improvement, and how they can be potentially attacked. Our findings show that embedding payloads into a binary is a simple yet effective way to defeat classifiers. Hence, the next-generation ML classifiers cannot be limited to look only into the first binary layer (the Dropper), but they will need to extract embedded payloads (e.g., via file carving) to classify them. When this strategy become mainstream, attackers will probably streamline the encoding of the payloads (e.g., using XOR, as we demonstrated). To handle this case, feature extractors should also try to guess the XOR key (Smirnov, 2016). In the challenge, these steps were not performed due to the artificial timing constraints imposed to simulate the actual performance constraints of real systems. Therefore, performance-efficient tools for tasks such as key guessing need to be developed so as to make those approaches become practical. Alternatively, a possibility that might tackle the issue of ML detection using static features would be to change the samples representation, aiming at covering less mutable features. In the Dropper case, a representation based on the instruction disassembly would be more suitable than one based on the PE characteristics, since all Droppers perform the same actions before dumping the payload to the file system. However, in a future in which this detection approach becomes more popular, attackers will likely inject tons of dead code constructions into the binaries (as we did for function imports) to defeat the classifier. Therefore, we can expect the reemergence of the arms-race started in the signature-based malware detection realm and its extension to the ML-based malware detection scenario.

3.2.6 Related Work

In this section, we present related work that propose defense or attack solutions for malware detection using machine learning in the literature.

There are many works in literature that propose defense solutions using machine learning. The majority of them consider using dynamic analysis, but they are not recommended for scenarios where a decision must be fast, given that a sandbox environment is required in order to extract dynamic attributes (Ye et al., 2017). Thus, static analysis is performed in these cases, looking at the content of the samples without requiring their execution by extracting byte Sequences, opcodes, API and system calls, strings, disassembly code, control-flow and data-flow graphs, or PE file characteristics (Gandotra et al., 2014; Ceschin et al., 2018; Anderson and Roth, 2018; Ye et al., 2017). Some solutions also consider strategies to defend (or to be more robust) against adversarial attacks, such as Label-based Semi-supervised Defense (LSD), Clustering-based Semi-supervised Defense (CSD) (Taheri et al., 2019), the use of control-flow graphs loop instructions as features (Machiry et al., 2018), or generative adversarial networks (GANs) as classification model (Hu and Tan, 2017).

In response to many of these defense solutions, attackers may try to evade them thought a great variation of attacks. Some of them consider the machine learning model being used, such as the perturbations used to attack neural networks and deep learning models (Carlini and Wagner, 2017), Silhouette Clustering-based Label Flipping Attack (SCLFA) (Taheri et al., 2019), or the samples generated by Generative Adversarial Networks (GANs), which may be used to attack other models (Hu and Tan, 2017). Other strategies consists in generating new variants of malware by using different packers (Aghakhani et al., 2020), creating binary mutations by applying transformations such as code replacement, instruction swapping, variable changes, dead code insertion and control flow obfuscation (Borello and Mé, 2008; Shao and

Smith, 2009), or by creating automated binary exploitation that automatically discover flaws and exploits (Shoshitaishvili et al., 2016).

In contrast to these related work, this research presents a practical experience from the 2020 edition of the Machine Learning Security Evasion Competition (MLSEC) regarding on attacking ML models and its effects on them and AVs, producing useful insights for future community's research work.

3.2.7 Conclusion

In this paper, we reported our experience on a malware detection evasion contest (MLSEC 2020) and presented our insights on how to attack and defend machine learning models focused on classifying programs into malicious or not. We were challenged to bypass the detection of 50 samples, all of them submitted to three distinct ML models in a black box manner. We were able to bypass all models and were the first team to reach 150 points (perfect score) in the contest. During the period of the contest, we discovered that: (i) the first model operate by looking to PE header characteristics, thus being evaded by embedding malicious payloads in a dropper executable; (ii) the second model classified function imports and libraries via a TF-IDF method, being evaded by the addition of fake imports to the dropper; (iii) the last model also checks for strings in the embedded content, being evaded by the encoding of the payload using XORing or base64 techniques. We highlighted the impact of these techniques in practice by demonstrating that the detection rate of the Virus Total AVs decreased when the evasive samples were considered in comparison to the original ones, even though no specific AV was targeted in the competition. By describing the steps we took during the competition, we present how attackers and defenders reason about the problem and the challenges they face. We expect that this work might help those being introduced to the field of adversarial attacks.

Reproducibility. The source code of the developed dropper was released as open-source and it is available on `github`: `https://github.com/marcusbotacin/Dropper`. The source code of the tool used to append goodware data to our adversaries is open-source and available on `github`: `https://github.com/ludersGabriel/BreakingGood`. The source code of our detection model is also open-source and available on `github`: `https://github.com/fabriciojoc/mlsec2020-needforspeed`. A Web-based version of our obfuscation mechanism is available on the `corvus` platform: `https://corvus.inf.ufpr.br`.

4 CONCEPT DRIFT

In this chapter, I present two papers related to concept drift in different cybersecurity data: user profiling and malware detection. In the former, I collected ecologically-valid computer usage profiles from 31 MS Windows computer users over 8 weeks and submitted this data to comprehensive machine learning analysis involving a diverse set of online and offline models. We found that: (i) profiles were mostly consistent over time, with repeating habits on a daily basis; (ii) computer usage profiling has the potential to uniquely characterize computer users; (iii) network-related events were the most relevant features to recognize profiles; and (iv) binary models were the most well-suited for profile recognition. In the latter, I compared some possible approaches to mitigate concept drift in malware detection and propose a novel data stream pipeline that updates both the classifier and the feature extractor, conducting a longitudinal evaluation by (i) classifying malware samples collected over nine years (2009-2018); (ii) reviewing concept drift detection algorithms to attest their pervasiveness; (iii) comparing distinct ML approaches to mitigate the issue; and (iv) proposing an ML data stream pipeline that outperformed literature approaches.

4.1 ONLINE BINARY MODELS ARE PROMISING FOR DISTINGUISHING TEMPORALLY CONSISTENT COMPUTER USAGE PROFILES

This paper was published in the IEEE Transactions on Biometrics, Behavior, and Identity Science journal, 2022 (Giovanini et al., 2022).

Luiz Giovanini[1,2], Fabrício Ceschin[1,3], Mirela Silva[2], Aokun Chen[2], Ramchandra Kulkarni[3], Sanjay Banda[3], Madison Lysaght[3], Heng Qiao[2], Nikolaos Sapountzis[4], Ruimin Sun[5], Brandon Matthews[6], Dapeng Oliver Wu[2], Andre Grégio[3], Daniela Oliveira[2]
[2]University of Florida, Gainesville, FL
lfrancogiovanini@ufl.edu
[3]Federal University of Paraná, Brazil
{fjoceschin, gregio}@inf.ufpr.br
[4]Cisco, San Jose, CA
[5]Khoury College of Computer Science, Northeastern University, Boston
[6]Electro-Optical Systems Laboratory, Georgia Tech Research Institute, Atlanta, GA

4.1.1 Abstract

This paper investigates whether computer usage profiles comprised of process-, network-, mouse-, and keystroke-related events are unique and consistent over time in a naturalistic setting, discussing challenges and opportunities of using such profiles in applications of continuous authentication. We collected ecologically-valid computer usage profiles from 31 MS Windows 10 computer users over 8 weeks and submitted this data to comprehensive machine learning analysis involving a diverse set of online and offline classifiers. We found that: (i) profiles were mostly consistent over the 8-week data collection period, with most (83.9%) repeating computer usage habits on a daily basis; (ii) computer usage profiling has the potential to uniquely characterize computer users (with a maximum F-score of 99.90%); (iii) network-related events were the most relevant features to accurately recognize profiles (95.69% of the top features distinguishing users were network-related); and (iv) binary models were the most well-suited for profile recognition, with better results achieved in the online setting compared to the offline setting (maximum F-score of 99.90% vs. 95.50%).

4.1.2 Introduction

Computer user profiling is the procedure of constructing a behavior-based digital identity of a person by leveraging their computer usage data, such as network traffic, process activity, mouse and keyboard dynamics (Kim et al., 2010; Yang et al., 2015). This procedure has the potential to uniquely characterize computer users by their usage patterns in terms of activity (e.g., process and network events) and temporal consistency (i.e., events repeating on a regular basis) (Tossell et al., 2012; Fridman et al., 2017; Payne et al., 2013). For example, a middle-aged CEO from a Fortune 500 company uses his computer very differently from a young software developer in a tech startup in California. While the former might uses email client, office software, web browser, and customized company management software, the latter will likely IDE, CAD, and software versioning tools, alongside a web browser, email client, and regularly accesses `.com.br` websites given their Brazilian descent.

[1]The first two authors have equal contribution.

Several factors make computer usage profiles (referred to as simply *profiles* for the remainder of this paper) well-suited for applications in continuous authentication (CA). The automatic recording of profiles can be implemented in a transparent fashion without requiring user intervention, thus facilitating usability and acceptability (Chuang et al., 2013). Furthermore, especially with regards to corporate employees, although some might perceive the recording of computer usage as potentially privacy-invasive, most already have a very limited expectation of privacy (Emami-Naeini et al., 2021). In fact, some employee activities (e.g., email and computer usage) are often recorded by their employers in both the public and private sectors (FindLaw, 2016).

This paper does **not** propose a CA solution targeting personal or corporate computer users. Instead, based on naturalistic data collected from an ecologically-valid user study, we seek to provide initial evidence on whether profiles constitute a *useful* and *feasible* source of data to uniquely identify computer users while they use their devices, discussing challenges and opportunities of leveraging profiles for applications in CA. Towards this end, we also investigated *temporal changes* in profiles (i.e., changes in computer usage habits taking place over time), which may impose additional challenges for CA and inform the design of more robust CA solutions. Our study focused on answering the following research questions:

- **RQ1**: Are computer usage profiles consistent over time (i.e., do computer usage habits repeat periodically)?
- **RQ2**: Do computer users have unique profiles? In other words, are profiles distinguishable from one another?
- **RQ3**: What features (e.g., network events, processes) are most important for constructing a unique profile?

To address these questions, we conducted an Institutional Review Board (IRB) approved user study to collect profiles from 31 computer users (MS Windows 10) over an 8-week period (taking place asynchronously between September 2020 and March 2021) in an ecologically-valid setting, wherein users interacted with their computers naturally without any interference or probing from the research team. Importantly, our definition of profiles encompasses process-, mouse-, network-, and keystroke-related events on the users' computers. We then used time series analysis techniques to investigate temporal changes in the profiles over the study period (**RQ1**). Our dataset was also submitted to a comprehensive machine learning analysis involving a diverse set of classifiers (one-class and binary models in both offline and online settings) aimed to assess profile uniqueness (**RQ2**) and identify the most important features in distinguishing among profiles (**RQ3**). All artifacts created throughout the course of this study, including our module for extracting profiles and our dataset of ecologically-valid computer usage data itself, will be made publicly available for vetted research usage.[2]

The main takeaways of our analyses are: (1) profiles were mostly consistent over time and repeating on a daily basis, yet exhibited some level of irregularity related to factors such as days off, and change of habits due to the COVID-19 pandemic; (2) computer usage profiles have the potential to uniquely identify computer users; (3) network features were the most relevant to accurately distinguish users; and (4) online binary models are preferred over offline binary and one-class models due to better performance in distinguishing profiles and robustness to temporal changes in computer usage.

[2]We are discussing with our IRB and legal counsel about options of data release based on University-mediated agreements with vetted researchers to protect the de-identified dataset to ensure that the data is used for ethical and responsible research only.

The remainder of the paper is organized as follows. Section 4.1.3 discusses the threat model faced by CA approaches and the assumptions of our analysis. Section 4.1.4 reviews related work. Section 4.1.5 presents our user study methodology. Section 4.1.6 describes the design and implementation of the profile extractor used to gather users' computer usage data. Section 4.1.7 describes the methodology of our machine learning analyses. Section 4.1.8 presents our experimental results. Section 4.1.9 discusses study's findings, limitations, and suggestions for future work. Section 4.1.10 concludes this paper.

4.1.3 Threat Model and Assumptions

This section discusses the main assumptions of this work and the threat model faced by CA approaches. First, we assume that CA methods are better suited to be employed in corporate environments, which are disproportionately targeted by internal and external adversaries, and consists of large groups of employees performing their primary tasks with a institutional computer device. Second, although our discussion is focused on CA, this paper does not propose yet a new CA solution. Third, we consider that corporate employees already face low expectations of privacy in their work environment (FindLaw, 2016). For example, in many organizations, network traffic, files and emails are monitored, and the devices and applications employees can access are restricted. The development of a CA solution leveraging computer usage profiles involves the recording of activity. Despite that we do not advocate the development of privacy-invasive CA solutions without the users being aware of their low expectations of privacy.

Threat Model. As discussed in Sec. 4.1.2, the main appeal of a CA system is not to replace traditional, point-of-entry authentication schemes (e.g., passwords or security keys), but to complement them and address their limitations—mainly that, after a user is authenticated into the system, their identity is not subsequently verified. A CA approach can potentially flag events that do not fit a learned user profile, e.g., connections to certain IP addresses or subnets, and patterns of exfiltration of information even after the user is authenticated, including insider attacks based on activity. For example, it is plausible that employees working on the same project will have similarities in profiles (e.g., same applications, files, schedules). Thus, in constructing a group profile, the malicious employee's outlier behavior (i.e., an insider attack) could be flagged as unusual activity. Given the challenges and privacy, security, and legal implications of conducting such study in a private corporation or government organization, we conducted our study with mainly university students and considering this large, diverse university environment as a proxy of an organization.

4.1.4 Related Work

There is a vast literature on leveraging user profiling for CA using a variety of data, such as network traffic (e.g., (Yang et al., 2015; Shi and Li, 2018)), use of applications (e.g.,(Fridman et al., 2017; Mahbub et al., 2019)), and keystroke or mouse dynamics (Ahmed and Traore, 2014; Kang and Cho, 2015; Sayed et al., 2013; Sun et al., 2016). User profiling has also been used for intrusion detection, where deviations from a "normal behavior" are considered as intrusions (Lazarevic et al., 2005).

There is, nonetheless, a scarcity of publicly available and diverse datasets containing users' application data (Mahbub et al., 2019; Murphy et al., 2017). Some large-scale micro-longitudinal datasets collected from several user study participants are available (e.g., (Mahbub et al., 2016; Ferreira et al., 2015)), but primarily focus on smartphone data instead of ecologically valid data from a personal laptop or desktop computer.

A notable exception is the dataset by Murphy et al. (Murphy et al., 2017), wherein the keystroke data, mouse movements and clicks, and background processes were collected from 103 users over 2.5 years on each participant's private computer in an ecologically valid setting. The authors found that authentication performance degraded significantly when data was collected in less controlled environments. Though this dataset is larger than ours, it did not include network metadata, which in our analysis, was the most relevant type of data to distinguish user profiles; we additionally did not collect keystrokes from users. Moreover, Murphy et al. did not analyze the temporal consistency of the profiles.

One line of research that closely resembles ours is that of Payne et al. (Payne et al., 2013), where the authors collected system events from seven volunteers for two hours in a controlled environment. However, unlike our user study, time information was not associated with user activities. Their experiment was also not ecologically valid because the users were not using their own devices nor their computers naturalistically. López *et al.* (López et al., 2019) used Self-Organizing Maps (SOM) to enhance insight and interpretability about user behavior data by untangling hidden relationships between variables. The authors generated the SOM visualizations of survey results that analyzed user behavior, security incidents, fraud, and data from a malware scanning tool to contrast Internet users' digital confidence with the level of malware infection. Their approach focused on drawing qualitative conclusions from their self-reported dataset. In contrast, we go beyond by exploring a naturalistic dataset quantitatively using multiple time series analysis methods and machine learning models.

In sum, our review of related work on user profiling for CA showed that none of the existing works evaluated the feasibility of desktop-based based computer usage profiling in a naturalistic way, as proposed here. Instead, many data collection procedures restricted user behaviors to specific tasks (Ayotte et al., 2021; Zhang et al., 2015) or devices (Ribeiro et al., 2015). Some CA proposals operate by building profiles based on short-term user activity records (Payne et al., 2013), while others did not explore changes in user behavior over time (Huang et al., 2017; Zhang et al., 2015). Thus, though these studies provide valuable insight into what technologies can be employed for computer usage profiling, this procedure's feasibility and temporal robustness remains understudied. Our findings shed light on such aspects, providing an actionable recommendation for future research and designing of effective, micro-longitudinal, behavior-based CA systems.

4.1.5 User Study Methodology

This IRB-approved study requested that participants installed our extractor module on their personal computers and used their devices naturally for 8 weeks. The entire study took place asynchronously from September 2020 to March 2021, wherein we successfully captured 8 weeks of computer usage data from 31 of the total 63 participants enrolled. In this section, we detail our methodology.

Participants. The study was originally comprised of 63 participants who were recruited via SONA , flyers, Internet advertising (e.g., the UF Facebook page that advertises studies), and word-of-mouth. Interested individuals were guided towards an online survey to determine eligibility which, if cleared to participate, would ask the participant for informed consent along with demographic information. After study completion, the participants who were recruited via SONA were compensated with two course credits, while the remainder with a $50 Amazon Gift Card. For inclusion in the data analysis, participants were required to complete the entire 8-week study period. This excluded 32 of the original 63 enrolled participants (who experienced technical issues as discussed below), thus reducing our sample size to 31 participants ranging

from 18–53 years (M = 27.32 years, SD = 8.27; 64.52% female, 32.26% male, 3.23% gender non-conforming). Table B.1 in Appendix B.1 summarizes the demographics of our participants. On average, we recorded 272 hours of computer usage data from each participant (SD = 189.9 hours, range: $31 - 859$), which corresponds to approximately 4.9 hours per study day (SD = 3.4 hr/day, range: $0.6 - 15.3$). We found some instances of high computer usage (e.g., 15.3 hr/day) that we hypothesize may be due to the participant leaving their device turned on for extended periods of time.

Procedure. At the beginning of the study, enrolled participants first were asked to complete an online survey to determine their eligibility: (i) above 18 years of age, (ii) own a personal computer (desktop or laptop) that is not shared and (iii) is used regularly, (iv) have regular access to the Internet, (v) have Windows 10 Operating System installed, and (vi) reside in the United States (as per UF IRB regulations for compensation). Once the participants were deemed eligible, they received a consent form disclosing the study procedures, minimal study risks, and data protection measures. The study did not involve deception in that participants were informed of the actual research purpose: continuously record their computer activities for 8 weeks to increase understanding of computer usage profiles. Participants were informed that keyboard keys and data sent over the network such as file contents **would not** be recorded, but that the following **would** be collected throughout the duration of the study: metadata related to software and network activity. Having read and electronically signed the informed consent form, participants were asked a series of demographic questions (e.g., age, ethnicity) and to install our extractor on their personal computer. Of note, our informed consent was approved with fairly general language that made it clear that we would be installing monitoring software on the participants' computers and collecting various kinds of computer usage information. This language was deemed clear and sufficient by UF IRB for purposes of informed consent because it is common to use language that omits technical terms/jargon pertaining to the study focused research area. Certain jargon would not only be meaningless to most subjects, but also possibly give them a false sense of security towards the power of these procedures or engender a false sense of protection against potential risks the study entails. This extractor would record logs of system-level events and uploads them to our lab servers in a secure fashion. For each participant, the 8-week study period began on the day successful installation could be verified within our systems. Participants were instructed to use their computers naturally and were reminded not to share their personal computers with anyone while in the study. Only IRB-trained research assistants who were part of the project interacted with participants to assist them with any questions or concerns they might have. Upon completion of the study and after uninstalling the extractor, participants were asked to complete a final debriefing questionnaire comprised of seven questions pertaining to the consistency of their computer usage during the study period. After completing the debrief, participants were compensated.

Data Attrition. A few issues occurred during the initial run of the study. First, because international participants were ineligible as per UF IRB, we discarded three participants that were located outside the U.S. We compensated them nonetheless (with UF IRB approval) and discarded their collected data. Second, our extractor only collected the destination IP address from 41 participants, thus we were unable to resolve which domains they accessed. To remedy this situation, we began capturing DNS Queries to resolve the destination IP address to its respective domain. All 41 participants who encountered this issue were communicated about an issue with data collected and invited to restart the study, receiving extra compensation for their extended time. The 19 participants who accepted were then asked to sign an addendum and promptly restarted the study once the new software update was available; the remaining 22 who declined were compensated upon study completion and had their data discarded from the study.

All of these issues were properly reported to the UF IRB and approval was received to restart the study. Lastly, after study completion, we noticed that seven participants had technical issues with the extractor; we therefore discarded them from our sample. These issues, coupled with participant attrition, decreased our sample from 63 to 31 participants.

4.1.6 Computer Usage Profile Extraction

In this section, we describe the design and implementation of the MS Windows 10 profile extractor our team developed to collect computer usage data from the study participants. We first considered using existing user-level and system event extractor tools (e.g., (Inc, 2020; Russinovich, 2019; Microsoft, 2017a; Russinovich and Garnier, 2019)) in a standalone fashion, but decided against these options either because (1) the tool was intended for diagnostic tasks, thus incurring prohibitive performance overheads for continuous usage, such as required for our user study (Russinovich, 2019); (2) the tool required nodes where data was collected to be on the same network, which would make our user study infeasible (Inc, 2020); or (3) the tool did not provide fine-grained process activities, such creation or termination of process/thread, the suspension/resumption of processes/threads, process memory access (Microsoft, 2017a; Russinovich and Garnier, 2019), which we deemed crucial in our design for the construction of computer usage-based profiles.

In our extractor, there are two main components installed on the users' systems: one for collecting data and one for uploading the data. The data collection component was based on a Sysmon-based logger to collect system level events, and a user-level application to collect keyboard and mouse related events. To collect system-level events (process creation and termination, network connection), we developed a logger based on Sysmon (Russinovich and Garnier, 2019), a suite of tools to debug, manage, diagnose Windows systems. To collect mouse and keyboard events, we developed a Windows application to collect three types of events: mouse clicks, keyboard clicks, and process information, complementing information collected by the Sysmon-based logger. Whenever a mouse click is made on any application, the timestamp is recorded and associated with the application where the click occurred. After the initial click, if the user continues clicking on the same application, no further entries for that application are made over the next five minutes. However, during this 5-minute window, if the user switches to another application and makes a mouse click, that click timestamp is indeed recorded (see log entry examples below). Timestamps for keyboard events are recorded using this same method.

- `1288$|$C:/Program Files/Google/Chrome/ Application/chrome.exe$|$132162206271021146$|$`
- `8056$|$C:/Program Files/AVAST/Application/ AvastBrowser.exe$|$132162207043754581$|$`

The motivations behind this 5-minute window were two-fold. First, we wanted to capture the user's profile of software usage as reflected by user activity in windows of five minutes. There is a tradeoff in deciding the window size: the shorter, the more accurate the profile is, the longer, the better the performance of the capturing module and less interference with user activities. Our design relies on windows of 5-minute of software activity as a proxy for software usage on that window. We considered three types of software activity: system-level events (e.g., process creation and termination, network connection) and mouse and keystroke events and the occurrence of either of them was considered software activity on the 5-minute window. Second, the mouse and keyboard activity added diversity to the type of data collected during the 5-minute

window, which helps enriching the uniqueness of the user profiles. Thus, although the window size brings tradeoffs and limitations, it captures whether there as user software activity on a given interval.

This logging application also records complementary process information. It records the amount of time the process stays in user vs. kernel mode in one-minute granularities. The second component installed in the user's computers is an uploader, which is a python script that runs every 5 minutes. All logs captured by the extractor module are securely transferred to our lab server.

4.1.7 Data Analysis

This section goes over our data pre-processing steps and analyzes, which involved a diverse set of time series analysis methods and machine learning (ML) models.

4.1.7.1 Data Pre-processing

We pre-processed the raw data of each participant to generate an $N \times 6$ matrix where each line represented one minute of computer activity on a given study day (the number of lines therefore varied among users according to their computer usage). The columns contained: (i) a timestamp, (ii) a list of all active processes, (iii) a list of all domains accessed, (iv) the number of clicks associated with the timestamp, (v) the number of keystrokes associated with the timestamp, and (vi) an indicator (binary) of background processes activity that was detected via the occurrence of network traffic coinciding with lack of keyboard/mouse events, indicating a network activity that was generated by the process independently and without the interaction of the user (e.g., automatic updates running without the user's knowledge, or the user listening to music on YouTube without interacting with the browser for a long time period).

4.1.7.2 Profile Temporal Consistency Analysis (RQ1)

We aimed to investigate whether the profiles were *consistent over time* or *periodic* (RQ1), that is, whether study participants repeated computer usage habits periodically. Towards this goal, from the pre-processed data of each user, we extracted a time series of the number of active minutes (y-axis) per hour within the 8-week study period (x-axis) in two conditions: (i) considering the entire activity exhibited by the user, which included background process activity, and (ii) discarding background process activity. We included the second condition because the portion of background process activity generated without user knowledge (e.g., automatic software updates) may add a periodic component to the profile that is not related with the user behavior. In other words, each of the 31 users was represented by two time series of $1,344$ data points each (56 study days $\times$ 24 hours/day) ranging from 0 to 60.

In a nutshell, our analysis was comprised of two steps. First, we used quantitative methods to investigate whether the time series describing the profiles were periodic (i.e., structured, correlated in time) or random/stochastic (i.e., unstructured, uncorrelated in time). Next, for all time series differing from unstructured data, we used standard techniques to check for periodicity and identify the period.

In the first step of our analysis, we used the surrogate data testing method (Theiler et al., 1992) to check for stochastic behavior in our time series. This method consists of comparing a particular nonlinear measure obtained from the time series with the distribution of the same measure obtained from a set of constructed time series—also known as *surrogates*—with the same statistical properties (e.g., mean, variance) but completely uncorrelated in time. If the result

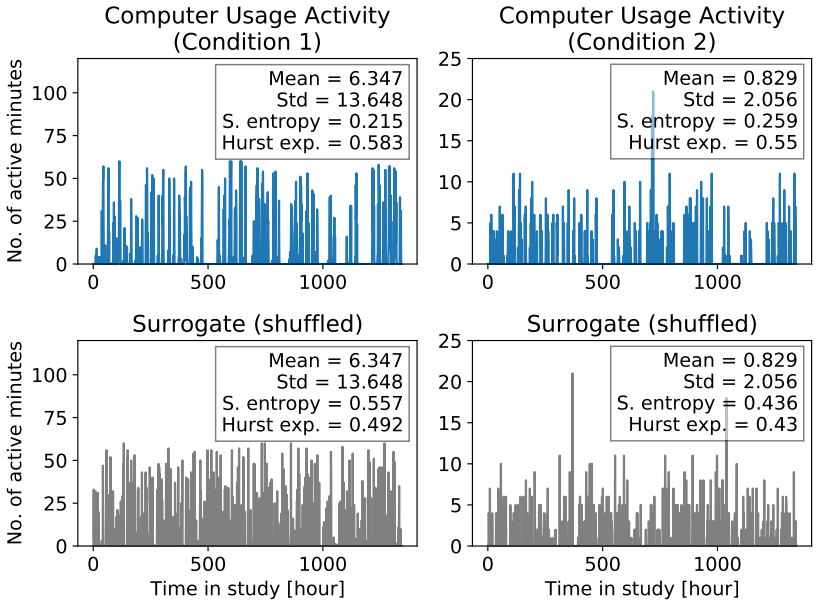


Figure 4.1: **Time series.** Time series of computer usage data and its respective surrogate counterpart for User 1 in Conditions 1 (left) and 2 (right).

from the time series deviates significantly from the surrogate distribution, one can conclude that the time series is unlikely to be purely stochastic and therefore possesses some level of correlation in time. We chose two well-known nonlinear measures able to capture temporal correlations in time series data:

- *Sample entropy* corresponds to the negative natural logarithm of the conditional probability wherein sequences similar for m points remain similar after adding one more data point ($m + 1$) within a tolerance threshold r (Richman and Moorman, 2000). The more unstructured (random) the time series, the larger the sample entropy.

- *Hurst exponent* is a measure of long-term memory (or long-range correlations) of a time series, ranging between 0 and 1. A Hurst exponent of 0.5 suggests a stochastic behavior. Results lower than 0.5 suggest an anti-persistent behavior (a high value is followed by a low value and vice-versa) while results higher than 0.5 suggest a trending behavior (a high value is followed by a higher one). The closer the Hurst exponent is to 0 or 1, the stronger the anti-persistent or trending behavior of the time series, respectively.

We created 100 unique surrogates for each time series by shuffling its data points, thus destroying any existing temporal correlation—which we measured via sample entropy and Hurst exponent—while preserving the statistical properties, as illustrated in Figure 4.1. We computed the sample entropy and Hurst exponent using the `nolds` Python library with default parameters ($m = 2$ and $r = 0.2$ in the former). Before calculating sample entropy, we normalized the time series and surrogates to zero mean and unit variance to remove any non-stationary components that may confound results (Costa et al., 2007). We did not normalize the time series nor the surrogates to extract the Hurst exponent since we are interested in capturing any existing trends in the data. We computed both metrics 100 times for each time series and then for its 100 unique surrogates. Next, after checking the normality of the distribution of the results with the Shapiro-Wilk test ($\alpha = 0.05$), we compared the distribution of each metric between time series vs. surrogates using either the paired t-test or the Wilcoxon signed-rank test ($\alpha = 0.001$).

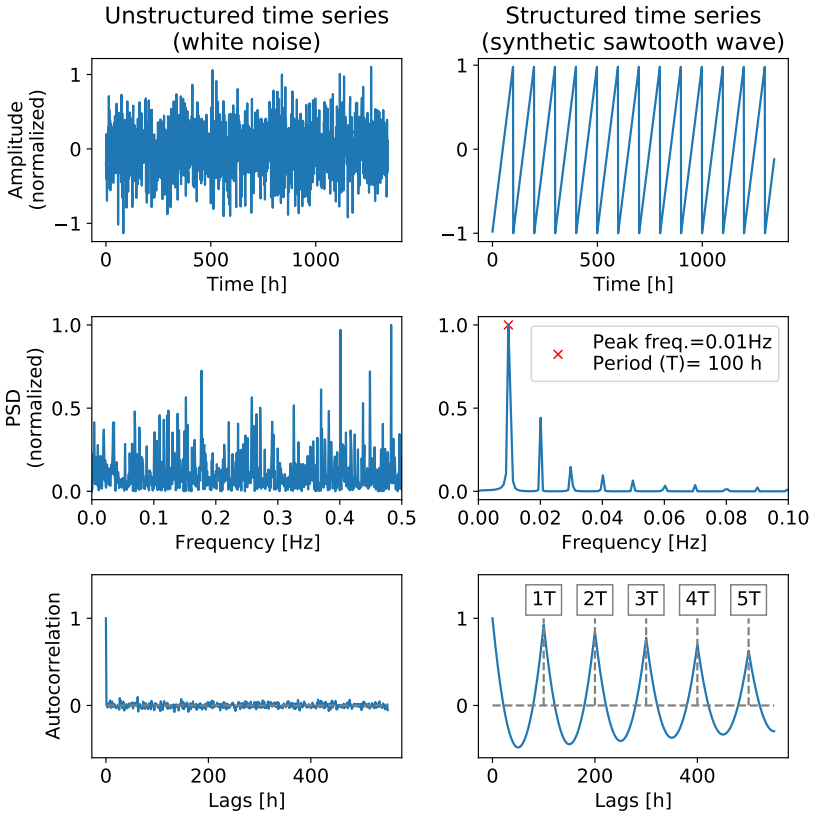


Figure 4.2: **Periodogram and autocorrelation.** Periodogram and autocorrelation for an unstructured (left panel) and structured (right panel) time series.

After verifying that our times series (profiles) differed from stochastic data, we attempted to identify their period using two popular signal processing techniques: the *periodogram* and the *autocorrelation function* (Box et al., 2015). The former is obtained through the discrete Fourier transform, a method for expressing a time series as a sum of periodic components, and contains an approximation of the power spectral density (PSD, i.e., the power over different frequency values) of that time series. As illustrated in Fig. 4.2, the periodogram of a stochastic time series contains many components spread over different frequency values, thus not clearly exhibiting a main component. Conversely, the periodogram of a structured time series exhibits clear peaks for the main frequency (0.01Hz in the given example) and its multiples (or harmonics). The frequency value containing the highest spectral power (aka peak frequency, f_p) is associated with the strongest periodic component of the signal through the relation $T = 1/f_p$. We therefore estimated the period of each of our time series as the inverse of the peak frequency in its periodogram (Box et al., 2015).

The second technique we used to estimate the period of our time series and validate the results obtained with the periodogram was the autocorrelation. The autocorrelation measures the self-similarity of a given time series over different delay times or lags (τ), ranging from -1 (perfectly negative autocorrelation) to 1 (perfectly positive autocorrelation) (Box et al., 2015). As illustrated in Fig. 4.2, for a stochastic (unstructured) time series, the autocorrelation is nearly zero for all values of lags because the data points are uncorrelated in time. On the other hand, for a structured time series with period T ($T = 100$ hours in the given example), the autocorrelation exhibits peaks for $\tau = k.T$ where $k = \{1, 2, 3, ...\}$, and its amplitude decreases as the lag increases. A common way to estimate the period of a given time series is through the examination of N

consecutive peaks in the autocorrelation function. In our analysis, we examined whether the five first peaks of each time series autocorrelation matched the period found through the periodogram. In other words, we verified whether the autocorrelation exhibited peaks for $\tau = k.T$ where $k = \{1, 2, ..., 5\}$.

4.1.7.3 Machine Learning Analysis

Our ML experiments aimed to assess profile uniqueness **(RQ2)** and identify top features in distinguishing among profiles **(RQ3)**.

Feature Extraction

An important aspect to be considered when analyzing profiles with ML models is the window of profile data needed for deciding whether the current behavior actually belongs to a certain user. We define this period as a *sliding window of time*. Since our logs were collected on a minute-basis, our window size t can assume any integer corresponding to the number of minutes (i.e., $t = 1, 2, 3, ...$). In our strategy, we summed the number of clicks, keystrokes, and background traffic activity (all integer numbers), and concatenated the strings of the processes and domains used within consecutive t-minute windows, keeping the last timestamp of the window (Fig. B.1 in Appendix B.2 illustrates this process for a window size of 3 minutes). Therefore, every minute, a newly acquired window size of t minutes is analyzed by the ML classifier, which makes a prediction about whether such window of activity belongs to a certain user. A disadvantage of this technique is that larger t values are more susceptible to the cold-start problem (i.e., when there is a lack of initial information to start the recognition task, common in recommendation systems (Yuan et al., 2016)). In our experiments, we opted to test several windows sizes to analyze the impact of small and large windows in predicting user identity based on the profiles. We tested $t = \{1, 2, 5, 10, 30, 60\}$ minutes.

Given that the numerical features (# of clicks, keystrokes, and background traffic activity) only need to be normalized before being used by a classifier, our feature extraction process focused on the textual attributes of the matrix (list of processes and domains). For this purpose, we used TF-IDF, a statistical measure that evaluates how important a word (in our case, a process or domain name) is to a text in relation to a collection of texts (Manning et al., 2008). Each text is represented by a sparse array that contains their TF-IDF values for each word in the vocabulary. Thus, each textual attribute from a window was transformed into its corresponding sparse array containing the TF-IDF values of each process and domain list (this process is also illustrated in Fig. B.1, Appendix B.2). We envisioned that TF-IDF brings many advantages for applications in CA, given that (i) the more a word (e.g., process) appears in an instance, the larger its feature weight is; (ii) the less a word appears in all files, the higher its importance is to distinguish instances; and (iii) it can be periodically updated to accommodate new processes and websites using evolving feature sets or retraining its vocabulary when, for instance, a concept drift is detected (Ceschin et al., 2018; Xu et al., 2019).

By combining the sliding window technique and TF-IDF features, the classifier receives as input an array that represents the window, containing the numerical features (sum of number of clicks, keystrokes, and background traffic activity) and the TF-IDF values for the processes and domains in the window, all normalized using maximum absolute scaler (Pedregosa et al., 2011). Thus, the textual attributes are used as a "document" that represents the user for a given moment.

Profile Uniqueness Analysis (RQ2)

The goal of the first set of machine learning experiments was to examine whether the profiles are unique (**RQ2**). For this purpose, we considered two types of ML models: (i) *offline*, where the classifier is trained only once using an initial portion of the data and then performs predictions for the remaining data without retraining; and (ii) *online*, where the classifier is periodically updated with more recent data. Offline classifiers are more popular in the ML field, mainly in stationary distribution problems where data does not change over time (e.g., object recognition). On the other hand, online models are conceptually more suitable to handle non-stationary distribution problems where data change over time (Bifet et al., 2018). Since we do not know which of these scenarios best describe our profiles. For comprehensiveness, we considered both types of classifiers in our analyses.

In both cases (offline and online learning), we conducted *binary* and *one-class classification* experiments. In the former, multi-class classifiers were used as binary discriminant functions, using the data from a given class (user) as positive, and the data from all the other classes (the remaining users) as negative (Bishop, 2006). In the latter, one predictive model was created for *each* user based on their data only (i.e., regardless of other users' data) (Han et al., 2011). Though popular in the ML field, multi-class classifiers were left out of our experiments due to their lack of feasibility for profile-based CA. This is because, in the multi-class setting, a single predictive model is created based on the data of *all* users and then used to distinguish them among each other, thus requiring retraining every time a new user is added or removed from the group. Considering all variations of classifiers, parameters, number of runs, and window sizes, we trained and tested 20,460 models in our experiments.

Offline Classification. For the offline experiments, we chose four multi-class classifiers used in the binary classification (Random Forest, Stochastic Gradient Descent (SGD), Multi-Layer Perceptron (MLP) and LinearSVC (Pedregosa et al., 2011)) and two one-class (Isolation Forest, and One-Class SVM with both RBF and linear kernel). For the binary classifiers, we used the default parameters in Scikit Learn (Pedregosa et al., 2011) for Random Forest (100 estimators), Stochastic Gradient Descent (hinge loss function), Multi-Layer Perceptron (100 hidden layers and relu activation function), and LinearSVC (Support Vector Machine linear kernel) for the SVM classifier. To evaluate these models, we split the data (ordered by timestamps) in two sets: the first seven days of data (of each user) were used for training and the remaining for testing. For the binary models, we created 31 binary models (one for each user, where the task is to detect if the current behavior window belongs to a given user) with each multi-class classifier cited before. To evaluate the binary models we used different negative users in the training and test sets, representing a typical continuous authentication system. For the one-class models, we also adopted the default settings in Scikit Learn (Pedregosa et al., 2011) (Isolation Forest with 100 estimators and two one-class SVM, the first with RBF kernel and the second with linear kernel). For each user, we created an outlier set containing the instances of all the other 30 users, excluding the first seven days—that is, the same instances taken as test set in the binary experiments, for a fairer comparison among one-class and binary results.

Online Classification. For the online experiments, we selected four classifiers that support online learning: three multi-class in the binary classification (Adaptive Random Forest; Stochastic Gradient Descent, SGD; and Perceptron (Pedregosa et al., 2011)) and one one-class (Half-Space Trees (Tan et al., 2011)). We leveraged the default parameters in Scikit Multiflow (Montiel et al., 2018) and River (Montiel et al., 2021) for both Adaptive Random Forest and Half-Space Trees (Tan et al., 2011) (which was the only online one-class classifier available at the time of this writing), and the default parameters in Scikit Learn for SGD and Perceptron. The same training and test sets from the offline experiments were used. The training set was used to train the initial model and the test set to create a data stream, which was considered in a test-then-train

evaluation, where each sample is tested by the model (generating a prediction used to compute the F-score) and then used to updated it (Bifet et al., 2018).

Evaluation. As per the evaluation metrics, we included recall, precision, and F-score (the harmonic mean of precision and recall) in both offline and online classification experiments. These metrics provide a more realistic measure of a model's performance for unbalanced datasets than more popular metrics, such as accuracy (Han et al., 2011). For each classifier, we obtained the recall, precision, and F-score values per user using the average of ten runs with different random states (for the classifiers and users in the training and test set in the binary classification experiments) (Breiman, 2001). The only exception was the Adaptive Random Forest, in which we calculated the results for a single run due to time constraints (as it exhibited an excessively high training time in our experiments). We then computed the average recall, precision, and F-score results among the 31 users for each classifier, along with the respective 95% confidence intervals.

Top Features Analysis (RQ3)

To analyze the most important features in identifying each user, we first divided their data in weeks to observe which are the most important over time. We trained a binary Random Forest classifier for each week of each user with a time window of 10 minutes (the optimal value found in RQ1), resulting in 248 models trained. As we found that most of the profiles repeat on a daily basis (see Sec. 4.1.8), we performed weekly training to consider several repetitions of computer usage habits to identify the most relevant features. Moreover, we chose the Random Forest for two reasons: it provides the importance of each feature in the classification process, and it outperformed the other tested binary models in both offline and online settings. We then used the Gini importance property from scikit-learn's Random Forest implementation (Pedregosa et al., 2011) to explain our models' predictions for each instance of the corresponding week and user. Finally, we computed these values (the higher the value, the more important the feature is) for each feature and obtained their average, where we collected the top 10 for each user.

4.1.8 Experimental Results

Profile Temporal Consistency (RQ1). In our surrogate data testing analysis, the sample entropy results obtained from the time series representing the computer usage profiles were statistically higher ($p < .001$) than the results obtained from the surrogates for all 31 users in the two assessed conditions (with and without background activity). The difference in the sample entropy results between time series vs. surrogates was more noticeable in the first condition (0.222 ± 0 vs. 0.714 ± 0.025) compared to the second (0.277 ± 0 vs. 0.499 ± 0.021). Regarding the Hurst exponent results, statistically different ($p < .001$) values for time series vs. surrogates were observed for 29 profiles (all users except User 5 and User 22) in the first condition (with background activity), and for all 31 profiles in the second condition (without background activity).

> All 31 profiles differed from unstructured data in terms of sample entropy in the two assessed conditions ($p < .001$). In terms of Hurst exponent results, the number of profiles that differed from unstructured data was 29 and 31 in the first and second conditions.

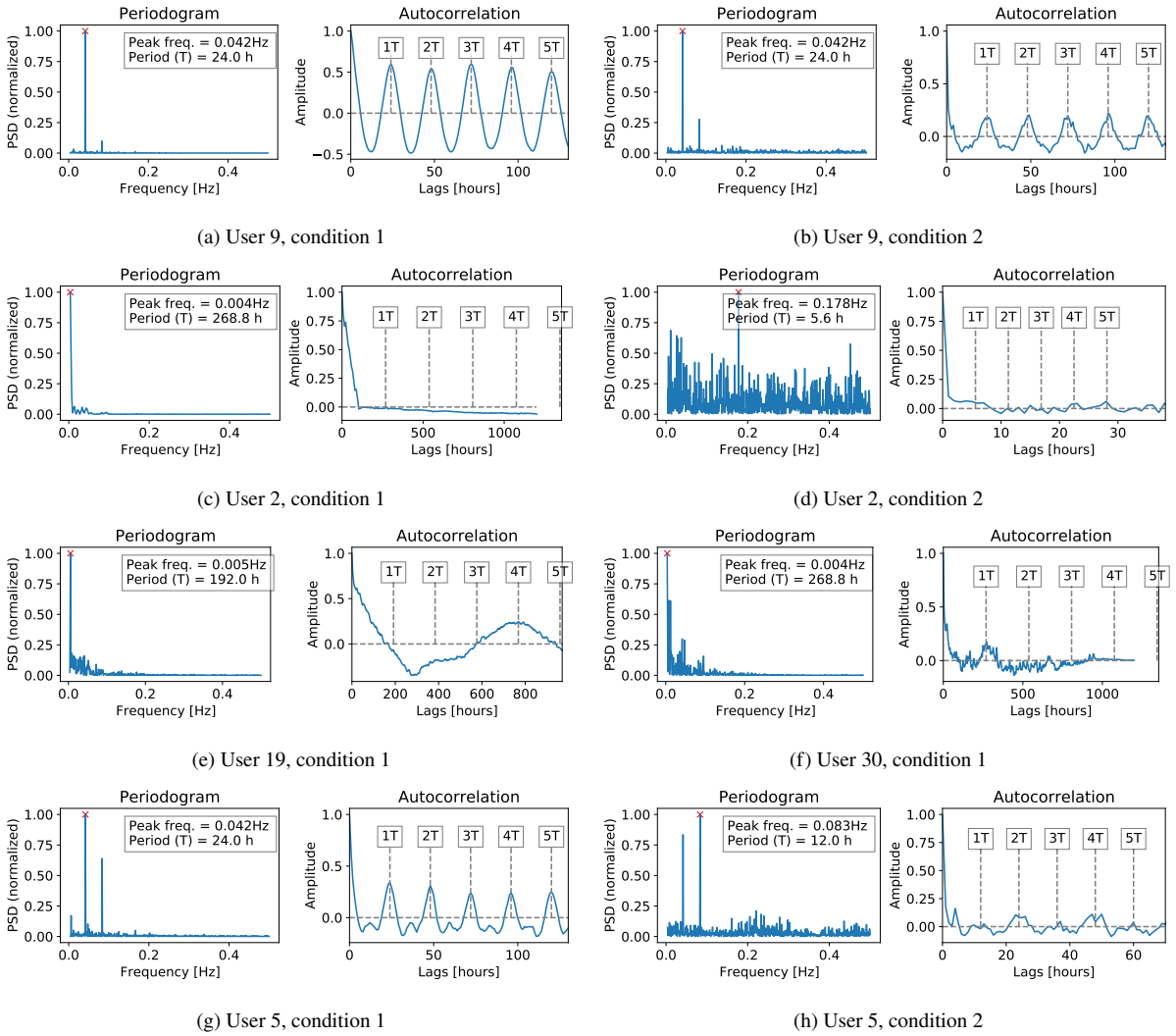
We identified a period for 28 profiles (90.3% of our sample) in the first condition (with background activity) and for 29 profiles (93.5% of our sample) in the second condition (without background activity). In both conditions, 26 profiles exhibited a period equal to 24 hours, as

exemplified in Fig. 4.3 for User 9. In other words, 83.9% of the study participants repeated computer usage patterns daily. The periodogram and autocorrelation for all users and conditions are exhibited in Appendix B.4 (Fig. B.2).

Figure 4.3 contrasts results between the periodogram and autocorrelation for Users 2, 19, and 30 in the first condition, and Users 2 and 13 in the second condition, preventing us from identifying a period in these cases. User 2's periodogram and autocorrelation were similar to the expected for unstructured data, especially in the second condition. For User 19, the periodogram exhibited a period of 192 hrs while the autocorrelation suggested 750 hrs. In the remaining cases, the autocorrelation did not exhibit peaks consistent with the period found with the periodogram.

Twenty-five profiles exhibited results consistent between the two conditions, with the period being equal to 24 hours in all cases. In the cases where the results changed between conditions (Users 5 and 8), the period was found to be lower in the second condition (i.e., after removing background activity), as illustrated in Fig. 4.3 for User 5.

A period was found for 90% of the profiles in Condition 1, and for 94% in Condition 2. 84% of the profiles exhibited a period of 24 hours.



(a) User 9, condition 1

(b) User 9, condition 2

(c) User 2, condition 1

(d) User 2, condition 2

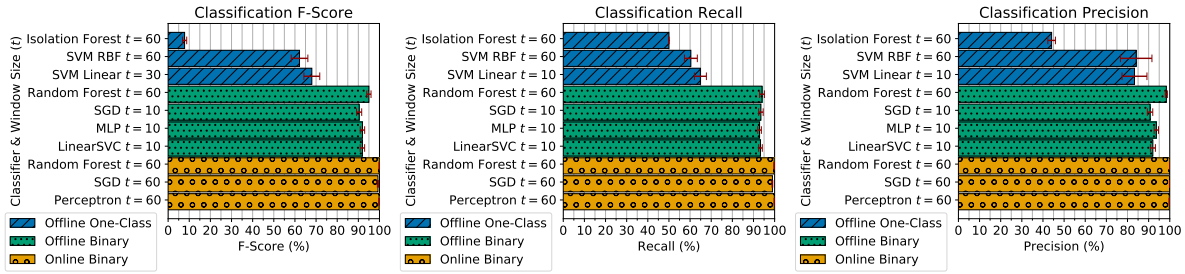
(e) User 19, condition 1

(f) User 30, condition 1

(g) User 5, condition 1

(h) User 5, condition 2

Figure 4.3: **Periodogram and autocorrelation.** Periodogram and autocorrelation results for different users and conditions.



(a) F-Score. (b) Recall. (c) Precision.

Figure 4.4: **Results.** Top average F-score (a), recall (b), and precision (c) values in distinguishing among the computer usage profiles. The error bars denote the 95% confidence intervals.

Profile Uniqueness (RQ2). Figure 4.4 shows the top results obtained to distinguish among the profiles. The complete results are exhibited in Appendix B.5 (Tables B.3, B.4, and B.5). The online one-class classifier yielded the poorest results: 0% precision, recall, and F-score; the Half-Space Trees model, the only model available, works better when anomalous data are rare, which is the opposite of our scenario (Tan et al., 2011).

In summary, the best results were reached with the online binary models (top F-score of 99.90%), followed by the offline binary models (top F-score of 95.00%). We observed that evaluation metrics improved with the increase of the window size. Most of the binary classifiers (our top performers) achieved optimal results for $t = 10$ min given that the increase of t to 30 min and 60 min did not cause the evaluation metrics to improve substantially.

Profile Top Features (RQ3). For 29/31 users, the top 10 features were all web domains. After unifying the top 10 features from each user and removing duplicates, we obtained a set of 186 unique features in which 177 were *domains* and only five were *processes*. The four remaining features refer to *background traffic* (which only appeared for two users), *number of mouse clicks*, and *number of keystrokes* (both appearing for a single user). From these 186 top features, 45 were observed for two or more users.

4.1.9 Discussion

Computer Usage Profiles Consistency. Our analysis of the temporal consistency of the profiles (**RQ1**) revealed that the majority of the users enrolled in our study did exhibit computer usage patterns consistent over time in the two assessed conditions (with and without background process activity). Interestingly, most of the users repeated computer usage on a daily basis over the 8-week study period. As the profiles also repeated for the integer multiples of their main period (as observed in the autocorrelation results), it is plausible to assume that such profiles also repeated every j days where $j = \{1, 2, ..., 56\}$, every k weeks where $k = \{1, 2, ..., 8\}$, and every l months where $l = \{1, 2\}$, within the study period. In other words, the profiles repeated not only on a daily basis, but also on a weekly and monthly basis, for example.

We acknowledge that these results are likely to be sensitive to changes in the user's routine that alter the way they use the computer (e.g., change of role, vacation, and travel in corporate environments), which might be the reason for the lack of periodicity observed for User 2. We therefore assume that most of our users did not have significant changes in their main occupation/tasks while in study.

Our analysis involving the surrogate data revealed that all profiles differed from completely unstructured data in the two tested conditions, i.e., they are unlikely to be purely stochastic. However, our periodogram and autocorrelation results (Fig. B.2) were not identical to those observed for purely structured data (Fig. 4.2). Our results look similar to the expected

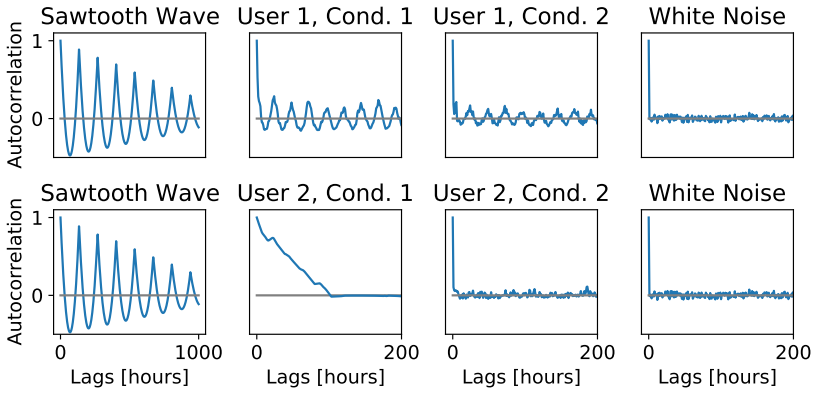


Figure 4.5: **Comparison.** Comparison between the autocorrelation for purely structured data (sawtooth wave) and purely stochastic data (white noise); results from User 1 (top) and User 2 (bottom).

for structured time series contaminated with random noise. In both conditions, most profiles exhibited a periodogram and autocorrelation more similar to the expected for structured data (e.g., User 1), suggesting lower presence of noise. On the other hand, results more similar to the expected for unstructured data were also observed for some users (e.g., User 2), suggesting higher contamination by noise (Fig. 4.5). In other words, users exhibited different levels of correlation in time in their use patterns.

These findings suggest that profiles can be described as a combination of two components, one *deterministic* (most likely nonlinear) and one *stochastic*, with the latter being stronger for some users than others. In other words, it seems reasonable to infer that profiles are consistent over time, but yet subject to random factors that decrease their temporal correlations and make them less structured.

Another factor that may have caused random drifts in the computer usage behaviors of our users is the current COVID-19 pandemic (our data collection happened during the months of the pandemic). The pandemic has greatly altered how users in general work, learn, socialize, and entertain themselves using their personal devices (Koeze and Popper, 2020), likely causing them to use the same computer for personal and work tasks. Several activities of daily life started to take place virtually due to safety concerns, such as grocery shopping, bank services, classes in general (e.g., schools, universities, gyms), watching movies or concerts etc., and many of them may not happen on a regular basis (e.g., someone might buy groceries twice a week for two weeks and then spend three weeks without buying anything). This has the potential to strengthen the stochastic component in the profiles.

Importantly, when moving from the first condition (with background activity) to the second one (without background activity), we noticed a decrease in the peaks of the autocorrelation function (Fig. B.2). We also observed an increase in the average sample entropy result (from 0.222 ± 0.114 to 0.277 ± 0.124), as well as a decrease in the average Hurst exponent result towards 0.5 (from 0.608 ± 0.104 to 0.564 ± 0.084). All these results indicate that the removal of the background activity caused the profiles to be less correlated in time. This finding is aligned with our hypothesis that the background activity contains periodic events that can strengthen the deterministic component in the profiles. However, some of these events may be produced by the OS without the knowledge of the user and according to a predefined schedule (e.g., automatic software update), which might make them easy to predict/mimic. Thus, it might be possible to increase the robustness of CA tools by leaving background process activity out of the computer usage profiles.

> The profiles were mostly consistent, with users repeating habits on a daily basis, but subject to random factors decreasing their temporal consistency.

Profiles Uniqueness. Our results from **RQ2** partially supported the hypothesis that profiles comprised of process-, network-, mouse-, and keystrokes-related events can uniquely characterize computer users. While most of the tested binary classifiers yielded promising results, the one-class models achieved poor results (in both offline and online settings). In other words, our profiles can be considered unique depending on the learning model used. Importantly, we trained our classifiers using one week of data, which potentially contained seven repetitions of computer usage habits given that we found most of the profiles to repeat on a daily basis (**RQ1**). This may explain the promising results achieved with the binary classifiers. Using less training data can allow the CA solution to be implemented in a more timely manner, but may also affect its performance since less usage patterns would be available for the learning model.

Our top feature analysis **(RQ2)** revealed that the set of domains accessed by the users was significantly more relevant to distinguish profiles, than other process- and keystrokes-related features. This finding suggests that collecting only network-related events can potentially lead to unique profiles. However, it is important to remark that our data was collected with users who experienced no restrictions on network activity, contrary to some corporate environment that restrict network activity, such as social media sites. Restricting the list of domains that users can access will potentially decrease profile uniqueness and impose challenges to profile-based CA systems. In this case, enriching a profile with process-, mouse-, and/or keystroke-related events may help to increase profile uniqueness and improve user recognition with CA.

> Computer usage profiles can uniquely characterize computer users when analyzed with binary classifiers. Network-related events constituted the most important feature to construct unique profiles.

Challenges and Opportunities of Using Computer Usage Profiling for Applications in CA. Regarding the feasibility of leveraging computer usage profiles for CA applications, our data suggests that binary models are better suited for profile-based CA tools than one-class models; while both allow the training on a per-user basis, which is advantageous for corporations due to employee turnover/hiring, the binary models were able to identify the profiles more accurately. Moreover, our results suggest that online binary models should be preferred over offline ones due to better performance and robustness to temporal changes in profiles.

One limitation of leveraging profiles for CA is that a certain amount of data is necessary to start recognizing users (a.k.a. cold-start problem). We found that a time window of 10 minutes was the optimal trade-off to recognize our dataset of profiles with the binary models (our top performers), but this optimal value might change for other datasets. This may be a challenge for profile-based CA tools given that the longer this optimal time window, the more susceptible the system is to a cold-start problem, causing the device to be initially vulnerable for a longer period of time. Moreover, the longer it takes for the system to recognize an anomaly, the less value the CA system might have for corporate security, because the anomaly could correspond to a malicious activity whose effects persist after detection.

4.1.9.1 Limitations and Future Work

One limitation of our study lies in our sample, which is small (N = 31) and comprised mainly of young adults from a large university (i.e., not representative of all possible environments).

Although a large university of nearly 50K students may be somewhat similar to an organization, corporate users likely present different computer usage profiles compared to university students. These factors thus limit the generalizability of our findings. Future studies with more diverse and representative samples are warranted.

Importantly, we collected our data during the COVID-19 pandemic, which might have affected user behavior and, consequently, influenced our conclusions towards profiles' uniqueness and consistency. The yet unforeseeable changes to how our society may shift after the pandemic is unresolved and how this may impact the user profiling analyses presented in this paper will require investigations from future work (e.g., will there be more sparse personal computer usage as users return to work in offices and classrooms?).

Another limitation of our study concerns the integrity of the systems in which computer usage data was recorded: we assumed that the users' systems were not compromised by malware during the study period. The presence of malware could have adversely affected the learning models, as malicious behaviors would be considered part of the usage profile. In addition, although all participants confirmed that they did not share their computers while in study, some self-reported responses might not be accurate.

Regarding our analysis of profiles' temporal consistency (RQ1), we focused on finding a period for each profile using well known techniques. Future research is needed to investigate whether profiles indeed posses a deterministic component combined with a stochastic one, as we concluded based on our results. Given the evidence of nonlinear dynamics in the profiles that we found via surrogate data testing, futures studies might benefit from using other non-linear metrics/techniques to better understand the behavior of computer usage patterns. For instance, one may try to reconstruct the phase space and investigate its properties.

In our classification experiments (RQ2 and RQ3), we employed a time window. This may constitute a limitation because, in some cases, when the behavior is about to change, the classifier may have difficult in detecting the shift because the window will gradually change with time, especially when its size is large (60 minutes, for instance). Future work addressing this issue is needed to improve classification performance. Moreover, we only considered traditional metrics (precision, recall, and F-score) to evaluate the performance of the tested classifiers. Future studies focusing on CA systems are advised to consider other evaluation metrics that are better suited for this case, such as receiver operating characteristics (ROC) curve (Sugrim et al., 2019).

4.1.10 Conclusions

We conducted an ecologically-valid user study with 31 computer users to systematically investigate whether computer usage profiles comprised of process, mouse, keystrokes, and network events are consistent over time and unique. Though we found evidence of temporal consistency for most of the profiles within the study period—with most of them reoccurring every 24 hours—our results suggest that these profiles experienced variations over time, due to factors such as days off. This suggests that online ML models (which allow for periodical retraining) might be better suited for a profile-based CA tool than offline models (where training occurs only once). In our comprehensive set of experiments, binary classifiers (online and offline) were able to accurately recognize the profiles, indicating that computer usage profiling can uniquely characterize computer users. Network domains accessed by users were more relevant in recognizing them than the keyboard and mouse activity.

4.2 FAST & FURIOUS: MODELLING MALWARE DETECTION AS EVOLVING DATA STREAMS

This paper was published in the Expert Systems with Applications journal, 2022 (Ceschin et al., 2022).

Fabrício Ceschin[1], Marcus Botacin[1], Heitor Murilo Gomes[2], Felipe Pinagé[1], Luiz S. Oliveira[1], André Grégio[1]
[1]Federal University of Paraná, Brazil
{fjoceschin, mfbotacin, fapinage, lesoliveira, gregio}@inf.ufpr.br
[2]Victoria University of Wellington, Wellington, New Zealand
heitor.gomes@vuw.ac.nz

4.2.1 Abstract

Malware is a major threat to computer systems and imposes many challenges to cyber security. Targeted threats, such as ransomware, cause millions of dollars in losses every year. The constant increase of malware infections has been motivating popular antiviruses (AVs) to develop dedicated detection strategies, which include meticulously crafted machine learning (ML) pipelines. However, malware developers unceasingly change their samples' features to bypass detection. This constant evolution of malware samples causes changes to the data distribution (i.e., concept drifts) that directly affect ML model detection rates, something not considered in the majority of the literature work. In this work, we evaluate the impact of concept drift on malware classifiers for two Android datasets: DREBIN (≈130K apps) and a subset of AndroZoo (≈285K apps). We used these datasets to train an Adaptive Random Forest (ARF) classifier, as well as a Stochastic Gradient Descent (SGD) classifier. We also ordered all datasets samples using their VirusTotal submission timestamp and then extracted features from their textual attributes using two algorithms (Word2Vec and TF-IDF). Then, we conducted experiments comparing both feature extractors, classifiers, as well as four drift detectors (Drift Detection Method, Early Drift Detection Method, ADaptive WINdowing, and Kolmogorov-Smirnov WINdowing) to determine the best approach for real environments. Finally, we compare some possible approaches to mitigate concept drift and propose a novel data stream pipeline that updates both the classifier and the feature extractor. To do so, we conducted a longitudinal evaluation by (i) classifying malware samples collected over nine years (2009-2018), (ii) reviewing concept drift detection algorithms to attest its pervasiveness, (iii) comparing distinct ML approaches to mitigate the issue, and (iv) proposing an ML data stream pipeline that outperformed literature approaches, achieving an improvement of 22.05 percentage points of F1Score in the DREBIN dataset, and 8.77 in the AndroZoo dataset.

4.2.2 Introduction

Countering malware is a major concern for most networked systems, since they can cause billions of dollars in loss (SecurityVentures, 2018). The growth of malware infections (CTONetworks, 2017) enables the development of multiple detection strategies, including machine learning (ML) classifiers tailored for malware, which have been adopted by the most popular AntiViruses (AVs) (Gandotra et al., 2014; Kantchelian et al., 2013; Jordaney et al., 2017). However, malware samples are very dynamic pieces of code – usually distributed over the Web (Chang et al., 2013)

and constantly evolving to survive – quickly turning AVs into outdated mechanisms that present lower detection rates over time. This phenomenon is known as concept drift (Gama et al., 2014b), and requires AVs to periodically update their ML classifiers (Ceschin et al., 2018). Malware concept drift is an understudied problem in the literature, with the few works that address it usually focusing on achieving high accuracy rates for a temporally localized dataset, instead of aiming for long-term detection due to malware evolution. Moreover, the community considers ML a powerful ally for malware detection, given its ability to respond faster to new threats (Gibert et al., 2020).

Being the most used operating system worldwide, Android has more than 2 billion monthly active devices (Fergus Halliday, 2018), with almost 40% of prevalence in the operating system market, surpassing Microsoft Windows in 2018 (StatCounter, 2018). As a widespread platform, Android is more affected by malware evolution and distribution, rendering its AVs vulnerable to concept drift effects, which makes the need to adapt existing solutions.

In this work, we evaluate the impact of concept drift on malware classifiers for two Android datasets: DREBIN (Arp et al., 2014) (≈130K apps) and a subset of AndroZoo (Allix et al., 2016b) (≈285K apps). For our longitudinal evaluation, we collected malware samples over nine years (2009-2018) and used them to train an Adaptive Random Forest (ARF) classifier (Gomes et al., 2017c), as well as a Stochastic Gradient Descent (SGD) classifier (Pedregosa et al., 2011). Our goal is to answer the following questions: (i) is concept drift a generalized phenomenon and not an issue of a particular dataset? (ii) is it important to update the feature extractor (and not only the classifier) when a concept drift is detected? (iii) how can we consider the feature extractor in the malware detection pipeline? (iv) which drift detector is the best for this task? (v) is concept drift somehow related to changes in the Android ecosystem? To do so, we ordered all datasets samples using their VirusTotal (Total, 2019) submission timestamp and then extracted features from their textual attributes using two algorithms (Word2Vec (Mikolov et al., 2013b) and TF-IDF (Salton et al., 1975)). After that, we conducted experiments comparing both feature extractors, classifiers, as well as four drift detectors (Drift Detection Method (Gama et al., 2014b), Early Drift Detection Method (Baena-Garcıa et al., 2006), ADaptive WINdowing (Bifet and Gavaldà, 2007), and Kolmogorov-Smirnov WINdowing (Raab et al., 2020)) to determine the best approach for real environments. We also compare some possible approaches to mitigate concept drift and propose a novel method based on the data stream pipeline that outperformed current solutions by updating both the classifier and the feature extractor. We highlight the need for also updating the feature extractor, given that it is as essential as the classifier itself to achieve increased detection rates due to new features appearing over time. Therefore, we propose a realistic data stream learning pipeline, including the feature extractor in the loop. We also show with our experiments that concept drift is a generalized phenomenon in Android malware. Finally, we discuss the impact of changes on the Android ecosystem in our classifiers by comparing feature changes detected over time, and the implications of our findings. It is worth notice that all code and datasets used will be publicly available[3], as well as the implementation of our data stream learning pipeline using `scikit-multiflow` (Montiel et al., 2018)[4], an open-source ML framework for stream data.

This article is organized as follows: we compare our work with the literature in Section 4.2.3; we introduce our methodology in Section 4.2.4; we describe the machine learning background in Section 4.2.5; we present the experiments results in Section 4.2.6; we discuss our findings in Section 4.2.7, and draw our conclusions in Section 4.2.8.

[3] `https://www.kaggle.com/datasets/fabriciojoc/fast-furious-malware-data-stream`

[4] `https://github.com/fabriciojoc/scikit-multiflow`

4.2.3 Related Work

The literature on malware detection using machine learning is extensive (Gandotra et al., 2014). Usually, the primary concern of most works is to achieve 100% of accuracy using different representations and models, ignoring the fact that malware samples evolve as time goes by. Few papers consider concept drift in this context, such as Masud et al., which were (at the best of our knowledge) the first to treat malware detection as a data stream classification problem like us, but proposing an ensemble of classifiers trained from consecutive chunks of data using v-fold partitioning of them and reducing classification error compared to other ensembles (Masud et al., 2008). They also presented a feature extraction and selection technique for data streams that do not have any fixed feature set (the opposite of our approach) based on information gain. Bach et al. combined two models: a stable one, based on all data, and a reactive one, based on a short window of recent data, to determine when to replace the current stable model by computing the difference in their accuracy, assuming that the stable one performs worse than the reactive when the concept changes (Bach and Maloof, 2008), very similar to what we do when the drift detector reaches a warning level.

Singh et al. proposed two measures to track concept drift in static features of malware families (which we do by using concept drift detectors, but in a malware detection task): relative temporal similarity (based on the similarity score between two time-ordered pairs of samples and are used to infer the direction of concept drift) and meta-features (based on summarization of information from a large number of features) (Singh et al., 2012), claiming to provide paths to further exploration of drift in malware detection models. Narayanan et al. presented an online machine learning based framework, named DroidOL to handle concept drift and detect malware (Narayanan et al., 2016) using control-flow sub-graph features in an online classifier, which adapts to the malware drift by updating the model more aggressively when the error is large and less aggressively, otherwise, which can also be done by drift detectors. Deo et al. proposed the use of Venn-Abers predictors to measure the quality of classification tasks and identify concept drift (Deo et al., 2016).

Jordaney et al. presented Transcend, a framework to identify concept drift in classification models which compares the samples used to train the models with those seen during deployment, computing algorithm credibility and confidence to measure the quality of the produced results and detect concept drift (Jordaney et al., 2017). Anderson et al. have shown that, by using reinforcement learning to generate adversarial samples, it is possible to retrain a model and make these attacks less effective, also protecting it against possible drifts, given that this technique hardens a model against worst-case inputs (Anderson et al., 2018), something that can be improved even more by retraining the feature extractor as we do. Pendlebury et al. reported that some results are inflated by spatial bias, caused by the wrong distribution of training and testing sets, and temporal bias, caused by incorrect time splits into these same sets (Pendlebury et al., 2019), which is one of the reasons why we collected the timestamps in our datasets and used data streams. They introduced an evaluation framework called TESSERACT to compare malware classifiers in a realistic setting and confirmed that earlier published results are biased. We reinforce that in our findings by comparing different experiments with a real-world simulation using data streams. Ceschin et al. compared a set of experiments that use batch machine-learning models with ones that take into account the change of concept (data streams), emphasizing the need to update the decision model immediately after a concept drift occurs (Ceschin et al., 2018). In contrast, we do not update only the decision model, but also the feature extractor when drift occurs. The authors also show that the malware concept drift is related to their concept evolution due to the appearance of new malware families.

Mariconti et al. created MaMaDroid, an Android malware detection system that uses a behavioral model, in the form of a Markov chain, to extract features from the API calls performed by an app (Onwuzurike et al., 2019). The solution proposed was tested with a dataset containing 44K samples, collected over six years, and presented a good detection rate and the ability to keep its performance for long periods (at least five years according to their experiments). We increased the length of the observation window by using malware samples collected over nine years. Cai et al. compared their approach (Cai, 2018; Cai and Jenkins, 2018), which uses 52 selected metrics concerning sensitive data accesses via APIs (using dynamic analysis) as features, with MaMaDroid. According to their experiments, their approach managed to remain stable for five years, while MaMaDroid only kept the same performance for two years. A very similar approach was compared with other four methods in literature and all of them presented an overall f1score bellow 72% (Fu and Cai, 2019).

Xu et al. proposed DroidEvolver, an Android malware detection system that can be automatically updated without any human involvement, requiring neither retraining nor true labels to update itself (Xu et al., 2019). The authors use online learning techniques with evolving feature sets and pseudo labels, keeping a pool of different detection models and calculating a juvenilization indicator that determines when to update its feature set and each detection model. We compared our approach with DroidEvolver and showed that it outperformed it. Finally, Gibert et al. presented research challenges of state-of-the-art techniques for malware classification, exemplifying the concept drift as one of them (Gibert et al., 2020).

Zhang et al. designed APIGraph, a framework to detect evolved Android malware that groups similar API calls into clusters based on the Android API official documentation (Zhang et al., 2020). Applying the aforementioned technique to other Android malware classifiers, the authors reduced the labeling efforts when combined with Tesseract (Pendlebury et al., 2019).

Finally, different from other approaches listed, Massimo Ficco presented an ensemble detector that exploits diversity in the detection algorithms by using both generic and specialized detectors (trained to detect certain malware types). The author also presents a mechanism that explores how the length of the observation time window can affect the detection accuracy and speed of different combinations of detectors during the detection (Ficco, 2022).

Our main contribution in this work is to apply data stream based machine learning algorithms to malware classification, proposing an important improvement in the pipeline that makes feature vectors correspond to the actual concept. The proposed solution, to the best of our knowledge, was not considered before and is as important as updating the classifier itself. To test and validate our proposal, we use two datasets containing almost 415K android apps, showing that it outperforms traditional data stream solutions. We also include an analysis of how certain features change over time, correlating it with a cybersecurity background.

4.2.4 Methodology

4.2.4.1 Threat Model and Assumptions

Our threat model considers an antivirus engine (AV) for the Android platform since it is the market share leader. Thus, successful malware infections affect a large number of users. For scientific purposes, we considered an AV entirely based on Machine Learning (ML) as implemented by many of the malware detectors cited in Section 4.2.3. In practice, however, it does not imply that an AV must use only this detection method: it can be complemented with any other detection approach envisioned by the AV vendor. Our detection model is completely static (features are retrieved directly from the APK files) because static malware detection is the most popular and fastest way to triage malware samples.Similarly to the above discussion, the use of static detectors

do not imply that an AV should not use dynamic components, but that our research focuses on improving the static detection component. In our proposed AV model, the APK files are inspected as soon as they are placed in the Android filesystem, i.e., it does not rely on any other system information except the APK files themselves. It is worth emphasizing that our goal is not to implement an actual AV but to highlight the need for updating ML models based on classifiers. Therefore, we simulated the behavior of an online AV using offline experiments that use data streams, simplifying our solution implementation. The details of the simulated ML model are presented below.

4.2.4.2 Data Stream

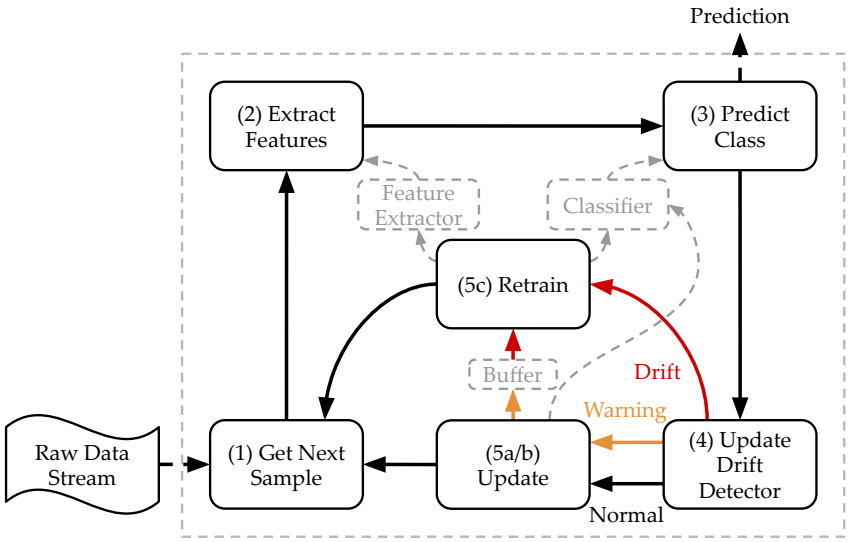


Figure 4.6: **Data Stream Pipeline.** Every time a new sample is obtained from the data stream, its features are extracted and presented to a classifier, generating a prediction, which is used by a drift detector that defines the next step: update the classifier or retrain both classifier and feature extractor.

Since our goal is to evaluate the occurrence of concept drift in malware classifiers, we analyzed malware detection evolution over time using a data stream abstraction for the input data. In a traditional data stream learning problem that includes concept drift, the classifier is updated with new samples when a change occurs—usually the ones that caused the drift (Gama et al., 2014b). Our data stream pipeline also considers the feature extractor under changes, according to the following five steps shown in Figure 4.6:

1. Obtain a new sample X from the raw data stream;
2. Extract features from X using a feature extractor E, trained with previous data;
3. Predict the class of the new sample X using the classifier C, trained with previous data;
4. With the prediction from C, update the drift detector D to check the drift level (defined by authors (Montiel et al., 2018));
5. According to the drift level, three paths can be followed, all of them making the pipeline restarts at **Step 1**:
    a **Normal:** incrementally **update** C with X;

b **Warning:** incrementally **update** C with X and add X to a buffer;

c **Drift: retrain** both E and C using only the data collected during the warning level (from the buffer build during this level), creating a new extractor and classifier.

4.2.4.3 Datasets

A challenge to detect concept drift in malware classifiers is to properly identify the sample's date to allow temporal evaluations. Since malware samples are collected in the wild and they may be spreading for some time, no actual creation date is available. As an approximation for it, we considered each sample's first appearance in VirusTotal (Total, 2019), a website that analyzes thousands of samples every day. Malware samples were ordered by their "first seen" dates, which allows us to create a data stream representing a real-world scenario, where new samples are released daily, thus requiring malware classifiers to be updated (Pendlebury et al., 2019).

In our experiments, we considered attributes vectors provided by the authors of DREBIN (Arp et al., 2014), composed of ten textual attributes (API calls, permissions, URLs, etc), which are publicly available to download and contain $123,453$ benign and $5,560$ malicious Android applications. We show DREBIN's distribution in Figure 4.7(a). We also considered a subset of Android applications reports provided by AndroZoo API (Allix et al., 2016b), composed of eight textual attributes (resources names, source code classes and methods, manifest permissions etc.) and contains $213,928$ benign and $70,340$ malicious applications. We show the distribution of our AndroZoo subset in Figure 4.7(b): it keeps the same goodware and malware distribution as the original dataset, which originally is composed by most of 10 million apps.

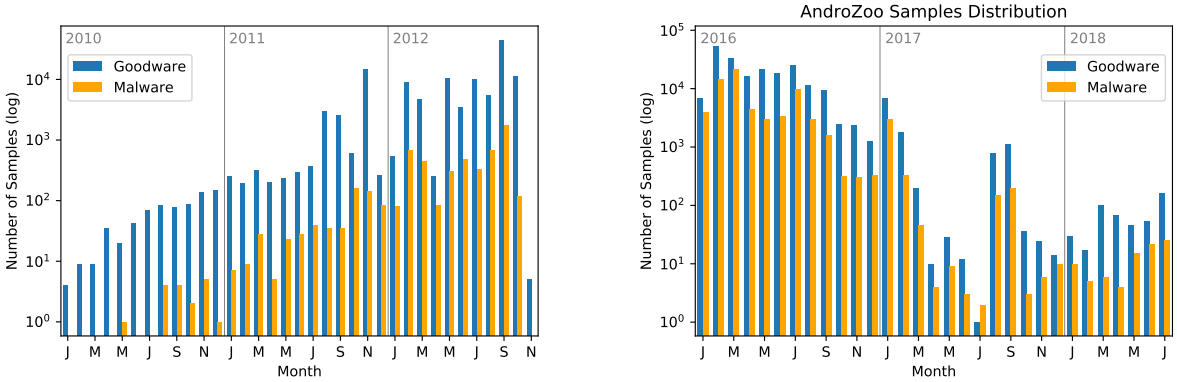
It is important to notice that we are using the attributes that were already extracted statically from them (Arp et al., 2014; Allix et al., 2016b), since we do not have access to applications binaries, packages, or source codes. In addition, the timing information provided by the authors of the DREBIN (Arp et al., 2014) and VirusTotal differs. According to Arp et al. (Arp et al., 2014), their samples were collected from August 2010 to October 2012. However, our version is based on VirusTotal appearance date, which showed us that very few samples were already analyzed by VirusTotal (48 in 2009) and some of them were just analyzed after the establishment of the collection (37 in 2013 and 2014), probably following the dataset publicly release. We do not show these samples in Figure 4.7 for better visualization.

Furthermore, both datasets reflect two characteristics of the real world that challenge the development of efficient ML malware detectors: (i) long-term variations (2009-2014, for DREBIN and 2016-2018 for AndroZoo); and (ii) class imbalance. For example, in DREBIN, whereas more than 40K goodware were collected in Sep/2009,only 1,750 malware samples were collected in the same period. Whereas class imbalance is out of this work's scope, we considered both datasets as suitable for our evaluations due to the long-term characteristic, which challenges ML classifiers to learn multiple concepts over time. Finally, to evidence the difference of both datasets, we created heat maps containing the prevalence of a subset of malware families created using the intersection of families from them (54 families), as shown in Figures 4.8(a) and 4.8(b).

4.2.5 Machine Learning Algorithms

4.2.5.1 Representation

A typical way to represent malware for ML is to create a vector using the sample textual attributes, such as API calls, permissions, URLs, providers, intents activities, service receivers, and others. In our work, we represented them using Word2Vec (Mikolov et al., 2013b) and TF-IDF (Salton et al., 1975), since they are widely used representations for text classification. Besides, both



(a) **DREBIN dataset distribution by month.** Very few samples are shown in 2009 (not shown), 2010, 2013 (not shown) and 2014 (not shown). The majority of them are in 2011 and 2012.

(b) **AndroZoo dataset distribution by month.** The majority of samples are in 2016, with fewer malware showing in 2017. February 2016 and March 2016 are the months with more prevalence of goodware and malware, repectively.

Figure 4.7: **Distribution.** Datasets distribution over time.

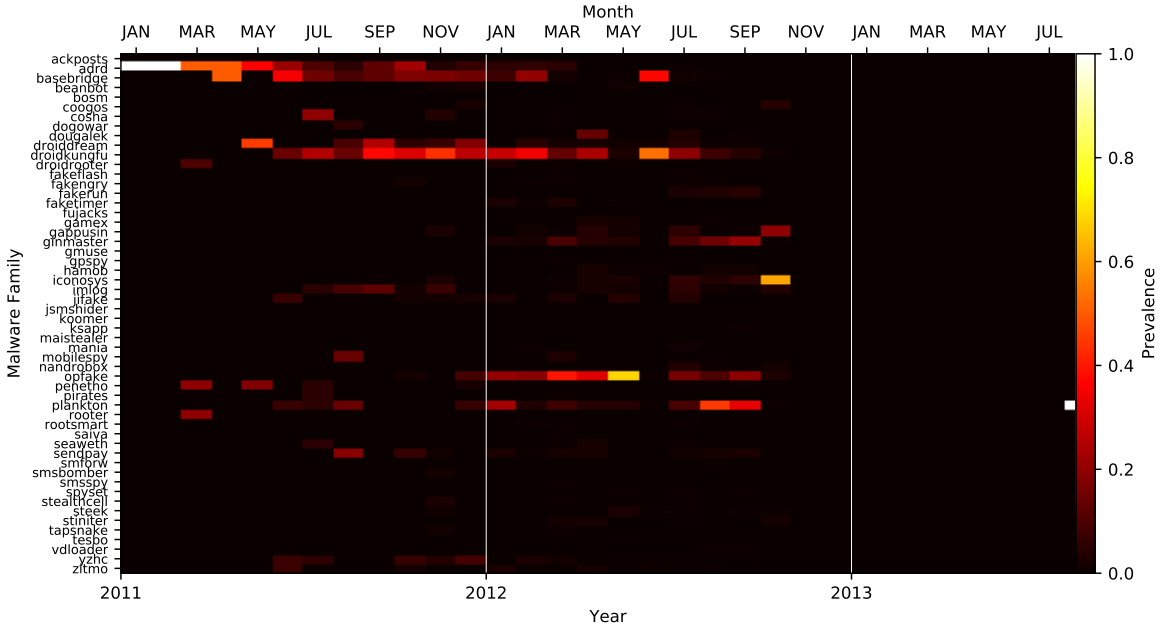
use the training data to generate the representation of all samples, which allows us to test our hypothesis that the drift affects not only the classifier but also the feature extractor.

4.2.5.2 Concept Drift Detectors

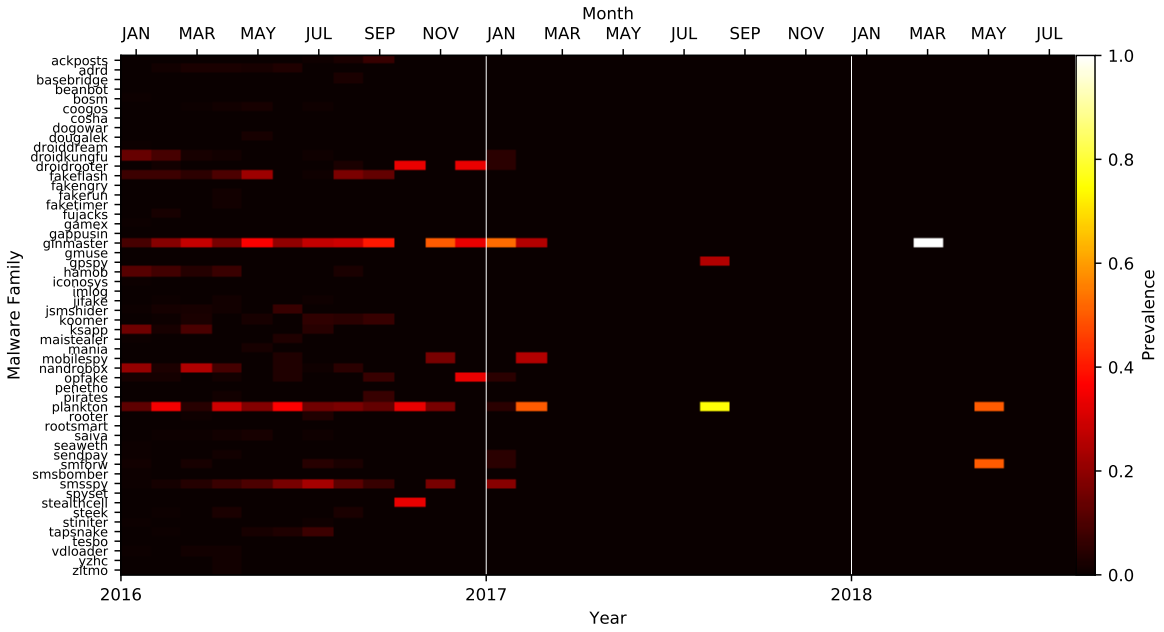
We used four concept drift detection algorithms in our work: DDM (Drift Detection Method (Gama et al., 2014b)), EDDM (Early Drift Detection Method (Baena-Garcıa et al., 2006)), ADWIN (ADaptive WINdowing (Bifet and Gavaldà, 2007)), and KSWIN (Kolmogorov-Smirnov WINdowing) (Raab et al., 2020). Both DDM and EDDM are online supervised methods based on sequential error monitoring, i.e., each incoming example is processed separately estimating the sequential error rate. Therefore, they assume that the increase of consecutive error rate suggests the occurrence of concept drift. DDM directly uses the error rate, while EDDM uses the distance-error rate, which measures the number of examples between two errors (Baena-Garcıa et al., 2006). These errors trigger two levels: warning and drift. The warning level suggests that the concept starts to drift, then an alternative classifier is updated using the samples which rely on this level. The drift level suggests that the concept drift occurred, and the alternative classifier build during the warning level replaces the current classifier. ADWIN keeps statistics from sliding windows of variable size, used to compute the average of the change observed by cutting them in different points. If the difference between two windows is greater than a pre-defined threshold, a concept drift is detected, and the data from the first window is discarded (Bifet and Gavaldà, 2007). KSWIN uses a sliding window of fixed size to compare the most recent samples of the window with the remaining ones by using Kolmogorov-Smirnov (KS) statistical test (Raab et al., 2020). Different from the other two methods, ADWIN and KSWIN has no warning level, which made it necessary to adapt our data stream ML pipeline. For instance, when a change occurs in ADWIN, out of the window data is discarded and the remaining samples are used to retrain the both the classifier and feature extractor. Finally, when a change occurs in KSWIN, only the most recent samples of the window are kept and used to retrain the classifier and feature extractor.

4.2.5.3 Classifiers

In all evaluations, we developed classification models leveraging Word2Vec and TF-IDF representations and normalized using a MinMax technique (Pedregosa et al., 2011). For all



(a) DREBIN dataset.



(b) AndroZoo dataset.

Figure 4.8: **Malware family distribution.** Intersection of families from both datasets shows evidences of class evolution, given that they are very different.

cases, we used as classifiers Adaptive Random Forest (Gomes et al., 2017c) without its internal drift detectors (working as a Random Forest for data streams), since its widespread use in the malware detection literature and has the best overall performance (Ceschin et al., 2018), and Stochastic Gradient Descent (SGD) classifier (Pedregosa et al., 2011), which is one of the fastest online classifiers in scikit-learn (Pedregosa et al., 2011). Both the classifier and drift detectors were configured using the same hyper-parameters proposed by the authors (Gomes et al., 2017c; Montiel et al., 2018).

Classifier	Algorithm	DREBIN Dataset				AndroZoo Dataset			
		Accuracy	F1Score	Recall	Precision	Accuracy	F1Score	Recall	Precision
Random Forest	Word2Vec	99.09%	88.73%	82.12%	**96.48%**	90.52%	76.27%	66.03%	90.29%
	TF-IDF	**99.23%**	**90.63%**	**85.85%**	96.31%	**91.54%**	**79.30%**	**70.25%**	**91.03%**
SGD	Word2Vec	98.29%	78.28%	70.90%	87.36%	85.41%	60.05%	47.52%	81.54%
	TF-IDF	**98.63%**	**83.26%**	**78.49%**	**88.66%**	**88.74%**	**71.57%**	**61.43%**	**85.73%**

Table 4.1: **Cross-Validation.** Mixing past and future threats is the best scenario for AVs in both datasets.

4.2.6 Experiments

4.2.6.1 The Best-Case Scenario for AVs (ML Cross-Validation)

In the first experiment, we classify all samples together to compare which feature extraction algorithm is the best and report baseline results. We tested several parameters for both algorithms and fixed the vocabulary size in 100 for TF-IDF (top-100 features ordered by term frequency) and created projections with 100 dimensions for Word2Vec, resulting in $1,000$ and 800 features for each app in both cases, for DREBIN and AndroZoo, respectively. All results are reported after 10-fold cross-validation procedures, a method commonly used in ML to evaluate models because its results are less prone to biases (note that we are training new classifiers and feature extractors at every iteration of the cross-validation process). In practice, folding the dataset implies that the AV company has a mixed view of both past and future threats, despite temporal effects, which is the best scenario for AV operation and ML evaluation.

Table 4.1 presents the results obtained in this experiment for both DREBIN and AndroZoo datasets using Adaptive Random Forest (ARF) and Stochastic Gradient Descent (SGD), highlighting the performance of TF-IDF, which was better than Word2Vec in all metrics, except in precision when classifying the DREBIN dataset. It means that Word2Vec is slightly better to detect goodware (particularly in DREBIN dataset) since its precision is higher (i.e., fewer FPs) and TF-IDF is better to detect malware due to its higher recall (i.e., less FNs). In general, we conclude that TF-IDF is better than Word2Vec since its accuracy and f1score are higher. Moreover, we notice that its f1score is not as high as the accuracy, which is near 100%, indicating that one of the classes (malware) is more difficult to predict than the other (goodware). Regardless of small differences, we observe that ML classifiers perform significantly well when samples of all periods are mixed, even in a more complex dataset such as AndroZoo, since they can learn features from all periods.

4.2.6.2 On Classification Failure (Temporal Classification)

Although currently used classification methodology helps reducing dataset biases, it would demand knowledge about future threats to work properly. AV companies train their classifiers using data from past samples and leverage them to predict future threats, expecting to present the same characteristics as past ones. However, malware samples are very dynamic, thus this strategy is the worst-case scenario for AV companies. To demonstrate the effects of predicting future threats based on past data, we split our datasets in two: we used the first half (oldest samples) to train our classifiers, which were then used to predict the newest samples from the second half. The results of Table 4.2 indicate a drop in all metrics when compared to the 10-fold experiment in both DREBIN and AndroZoo datasets and also sugest the occurrence of concept drift on malware samples, given that the recall is much smaller than the previous experiment.

Due to dataset imbalance in both datasets, we notice a bias toward goodware detection (very small accuracy decrease) and significant qualitative differences for f1score and recall (much

Table 4.2: **Temporal Evaluation.** Predicting future threats based on data from the past is the worst-case for AVs.

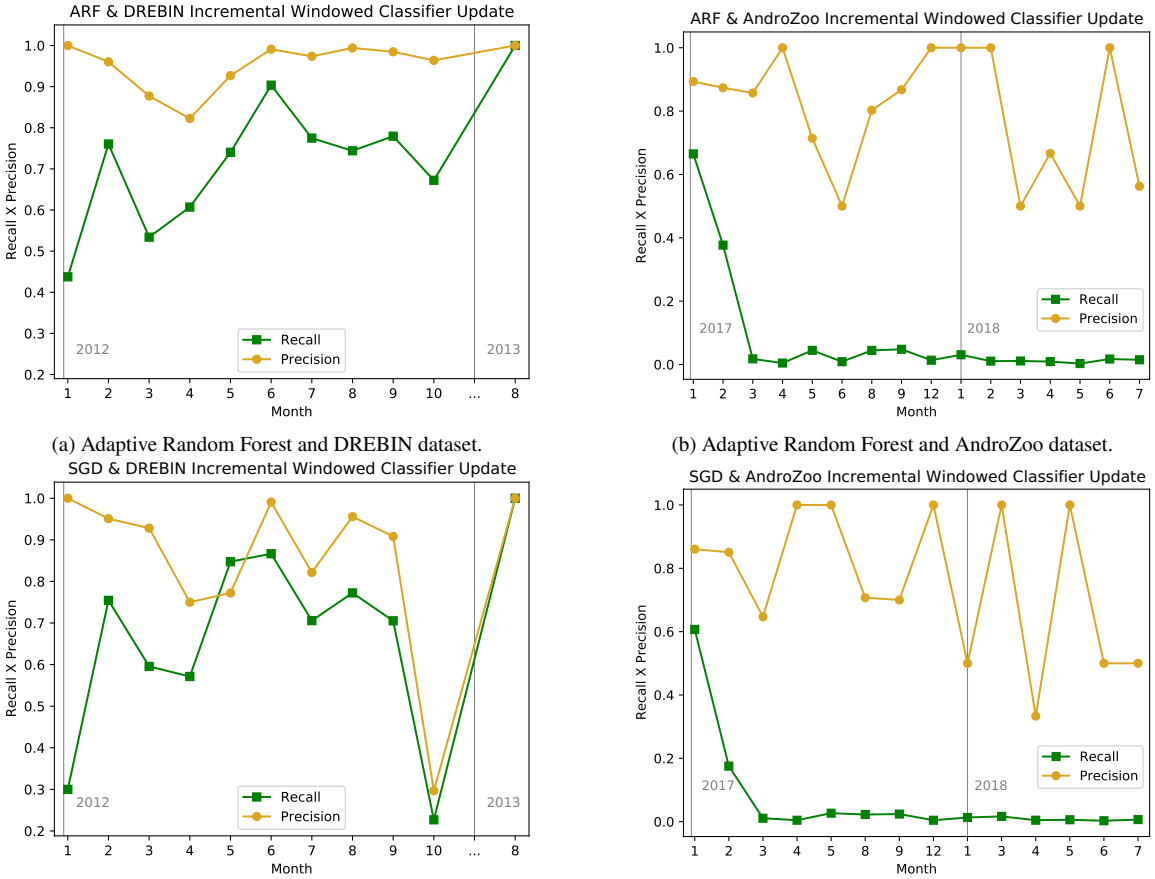
Classifier	Algorithm	DREBIN Dataset				AndroZoo Dataset			
		Accuracy	F1Score	Recall	Precision	Accuracy	F1Score	Recall	Precision
Random Forest	Word2Vec	97.66%	62.58%	46.31%	96.47%	87.55%	53.95%	38.71%	88.96%
	TF-IDF	**98.20%**	**73.26%**	**58.36%**	**98.39%**	**88.20%**	**57.13%**	**41.71%**	**90.63%**
SGD	Word2Vec	97.52%	65.14%	55.08%	79.70%	85.81%	47.04%	33.44%	79.28%
	TF-IDF	**98.15%**	**75.18%**	**66.42%**	**86.61%**	**86.96%**	**52.79%**	**38.68%**	**83.07%**

worse than before). The precision score was very similar to the cross-validation experiment, and better in the case of TF-IDF when classifying DREBIN, showing that goodware samples remain similar in the "future" while malware samples evolved. Word2Vec and TF-IDF lost, in average, about 16 percentage points of their f1score in DREBIN and about 19 percentage points for this same metric in AndroZoo. In addition, the model's recall rates significantly dropped in both Word2Vec and TF-IDF when classifying DREBIN and AndroZoo, regardless of the models used. This indicates that detecting unforeseen malware only from a single set of past data is a hard task. These results highlight the need of developing better continuous learning approaches for effective malware detection.

4.2.6.3 Real-World Scenario (Windowed Classifier)

Since static classifiers are a bad strategy, AV companies adopt continuous updating procedures as samples are collected and identified. From a ML perspective, they adopt an incremental stream learning method (Pinage et al., 2016), which we call Incremental Windowed Classifier (IWC). Notice that this is the same approach used by other authors in the literature, but they propose the use of distinct attributes and features only, instead of a new stream pipeline (Zhang et al., 2020; Cai, 2020). To evaluate the impact of this approach, we divided the datasets into two groups, one containing samples released until a certain month of a year for training, and the other with samples released one month after that said month for testing. For example, considering Jan/2012, the training set contained samples that were created until Jan/2012 (cumulative) and the validation set contained samples created only in Feb/2012 (a month later). We tested both Adaptive Random Forest (ARF) (Gomes et al., 2017c) and Stochastic Gradient Descent (SGD) classifier (Pedregosa et al., 2011) with TF-IDF (100 features for each textual attribute), given its better performance in previous experiments, and excluded months with no samples. Every month, both classifier and feature extractor were retrained, generating new classifiers and feature extractors.

The results for the DREBIN dataset shown in Figures 4.9(a) and 4.9(c) indicate a drop in both precision and recall rates. For Adaptive Random Forest, precision drops to about 85% in April/2012 and increases in the following months to almost 100% in August/2013. The Recall starts with the worst result (about 40%), in January/2012 and reaches almost 100% in August/2013. The average recall was 70.3% and average precision, 95.36%. For SGD, the worst precision is reported in October/2012 (about 30%), increasing to almost 100% in August/2013. The same happen with the recall, which drops to almost 20% in October/2012 and increases to almost 100% in August/2013. Finally, the average recall was 66.78% and average precision, 85.22%. On the one hand, the growth of precision and recall in some periods indicate that continuously updating the classifier might increase classification performance in comparison to using a single classifier tested with samples from a past period because the classifier learns new patterns for the existing concepts (features) that may have been introduced over time. On the other hand, the drop in the precision and recall rates in some periods indicate the presence of



(a) Adaptive Random Forest and DREBIN dataset.

(b) Adaptive Random Forest and AndroZoo dataset.

(c) SGD and DREBIN dataset.

(d) SGD and AndroZoo dataset.

Figure 4.9: **Continuous Learning.** Recall and precision while incrementally retraining both classifier (Adaptive Random Forest and Stochastic Gradient Descent classifier) and feature extractor.

new concepts (e.g., new malware features) that were not fully handled by the classifiers. When looking at the AndroZoo dataset results, as shown in Figures 4.9(b) and 4.9(d), it is possible to note a drastic fall in recall as time goes by in both cases, dropping from almost 70% (ARF) and 60% (SGD) in January/2017 to less than 5% in March/2019, not exceeding 10% by the end of the stream. This indicates that AndroZoo is a much more complex dataset with very different malware in distinct periods, given that the recall did not behave as this same experiment using the DREBIN dataset. In contrast, the precision remained unstable. For Adaptive Random Forest, it reaches 100% in some periods, such as May/2017, and drops to almost 50%, in June/2017, with quite similar behavior from the DREBIN dataset. For SGD, it also reaches 100% in some periods, such as April/2017, and drops to almost 30% in April/2018. These results indicate that more than continuous retraining, AV companies need to develop classifiers fully able to learn new concepts.

4.2.6.4 Concept Drift Detection using Data Stream Pipeline (Fast & Furious – F&F)

Although the previous results indicate that continuously learning from the past is the best strategy for detecting forthcoming threats, concept drift remains challenging, even more when considering results from the AndroZoo dataset. Therefore, we present an evaluation of a drift detector approach that could be deployed by AV companies to automatically update their classifier models. In our experiments, we used both Adaptive Random Forest (Gomes et al., 2017c) and Stochastic Gradient Descent as the classifiers, DDM, EDDM, ADWIN, and KSWIN as drift

detectors (Montiel et al., 2018) (with the same parameters as the authors), and TF-IDF (using 100 features for each textual attribute) as a feature extractor. In the DREBIN dataset, we initialized the base classifier with data from the first year, given that in the first months we have just a few malware showing up. In the AndroZoo dataset, we initialized it with data from the first month. We tested all drift detection algorithms in two circumstances when creating a new classifier, according to our data stream pipeline: (i) **Update:** we just update the classifier with new samples (collected in warning level or ADWIN window), which reflects the fastest approach for AV companies to react to new threats; (ii) **Retrain:** we extract all features from the raw data (also collected in warning level or ADWIN window) and all models are built from scratch again, i.e., both classifier and feature extractor are retrained and a new vocabulary is built based on the words in the training set for each textual attribute (more complete approach, but time-consuming for AV companies). To compare our solution with another similar in the literature, we implemented our version of DroidEvolver (Xu et al., 2019), replicating their approach using the same feature representation as ours. It is important to report here that we tried to use the authors' source code, but they were not working properly due to dependencies that could not be installed. Thus, we implemented an approximation of their method by analyzing their paper and code, using $\tau_0 = 0.3$ and $\tau_1 = 0.7$ with three compatible online learning methods from scikit-learn (Pedregosa et al., 2011): `SGDClassifier`, `PassiveAggressiveClassifier`, and `Perceptron`. Finally, during the execution of DroidEvolver, we noticed that it was detecting at least one drift in one of its classifiers each iteration, making it necessary to create a new parameter that checks for drift in an interval of *steps* (iterations), in our case 500 steps was selected according to our analysis.

Table 4.3 presents the results for DroidEvolver and our methods for both datasets. For DREBIN, when using Adaptive Random Forest, EDDM outperforms DDM methods' classification performance in all scenarios, which was the opposite when using SGD classifier. For both classifiers, ADWIN outperforms EDDM, DDM, and KSWIN, providing the best overall performance for retraining as well as for learning new features. However, only when using Adaptive Random Forest, ADWIN with the update is better for precision, making it slightly better in reducing false positives, and KSWIN is better for recall, making it better in reducing false negatives. Moreover, retraining both the feature extractor and classifier makes it detect fewer drift points than the updating approach (18 vs. 20 when using ARF, 13 vs 14 when using SGD). Overall, SGD outperformed Adaptive Random Forest when classifying DREBIN, presenting an improvement in f1score of almost 16 percentage points. In comparison to DroidEvolver, our approach outperformed it in all metrics, with a relatively large margin. In Figure 4.10, it is possible to observe the prequential error of Adaptive Random Forest with ADWIN using Update (Figure 4.10(a)) and Retrain (Figure 4.10(b)) strategies. Despite the Retrain strategy is similar to the update approach in some points, it has less drift points, a lower prequential error and, consequently, a better classification performance.

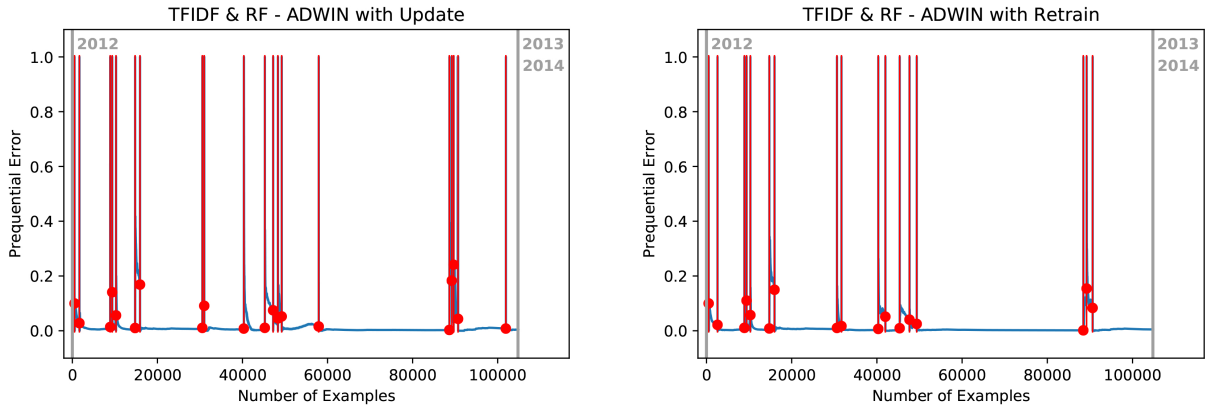
When classifying AndroZoo, DDM outperforms EDDM, which was not a good drift detector for this specific dataset. ADWIN with retraining was the best method again, detecting 50 and 33 drift points versus 78 and 30 when using the update approach with Adaptive Random Forest and SGD classifier, respectively, outperforming all other methods, including DroidEvolver again. Finally, these results suggest that AV companies should keep investigating specific samples to increase overall detection.

4.2.6.5 Multiple Time Spans

In the previous experiment, we showed that the use of drift detectors with the Retrain strategy improved the classification performance. Although the experiment simulates a real-world stream,

Table 4.3: **Overall results.** Considering IWC, DroidEvolver (Xu et al., 2019), F&F (U)pdate and (R)etrain strategies with multiple drift detectors and classifiers (Random Forest and SGD).

Classifier	Method	DREBIN Dataset					AndroZoo Dataset				
		Accuracy	F1Score	Recall	Precision	Drifts	Accuracy	F1Score	Recall	Precision	Drifts
Model Pool	**DroidEvolver (Xu et al., 2019)**	97.27%	67.14%	59.14%	77.64%	69	87.09%	66.28%	56.17%	80.83%	22
Adaptive Random Forest	**IWC**	96.8%	80%	70.3%	95.36%	N/A	82.99%	11.4%	8.25%	79.61%	N/A
	DDM (U)	98.3%	79.19%	68.38%	94.04%	8	88.19%	70.84%	63.5%	80.11%	14
	DDM (R)	98.4%	80.54%	70.27%	94.32%	9	87.96%	69.92%	61.94%	80.27%	24
	EDDM (U)	98.53%	82.27%	71.84%	96.26%	27	78.28%	39.52%	37.23%	42.11%	118
	EDDM (R)	98.57%	82.85%	73.09%	95.6%	14	77.91%	39.31%	37.52%	41.28%	17
	ADWIN (U)	98.58%	82.66%	71.35%	**98.21%**	20	86.41%	65.86%	58.02%	76.15%	78
	ADWIN (R)	**98.71%**	**84.44%**	74.17%	98.02%	18	89.6%	**75.05%**	**69.23%**	81.93%	50
	KSWIN (U)	98.38%	80.82%	72.19%	91.80%	10	89.22%	71.78%	60.66%	87.88%	70
	KSWIN (R)	98.55%	83.01%	**74.96%**	93.00%	9	**89.66%**	73.22%	62.56%	**88.26%**	50
SGD	**IWC**	96.33%	73.40%	66.78%	85.22%	N/A	82.63%	9.16%	6.61%	75.71%	N/A
	DDM (U)	98.84%	87.56%	86.29%	88.88%	3	88.05%	71.17%	65.30%	78.20%	7
	DDM (R)	98.85%	87.67%	86.69%	88.66%	3	89.10%	73.81%	67.98%	80.73%	4
	EDDM (U)	98.85%	87.63%	86.43%	88.87%	11	82.99%	42.50%	32.97%	59.75%	85
	EDDM (R)	98.78%	87.05%	86.61%	87.49%	20	82.77%	41.80%	32.45%	58.72%	73
	ADWIN (U)	98.95%	88.68%	87.03%	90.38%	14	89.39%	73.98%	66.74%	82.97%	30
	ADWIN (R)	**99.00%**	**89.19%**	**87.38%**	**91.08%**	13	**89.49%**	**74.31%**	67.25%	**83.01%**	33
	KSWIN (U)	98.93%	88.45%	86.83%	90.13%	1	88.23%	72.41%	**68.34%**	77.00%	48
	KSWIN (R)	98.85%	87.71%	87.28%	88.15%	2	85.34%	67.05%	65.99%	68.14%	70



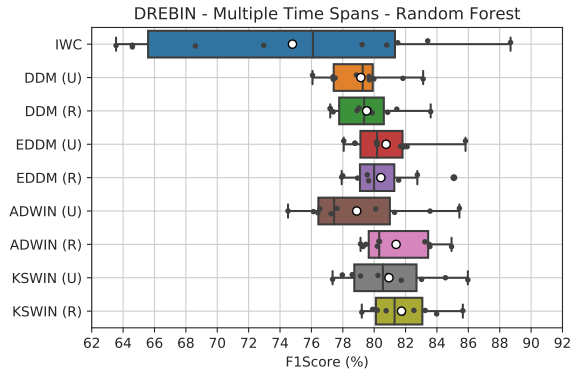
(a) Adaptive Random Forest with ADWIN and Update.

(b) Adaptive Random Forest with ADWIN and Retrain.

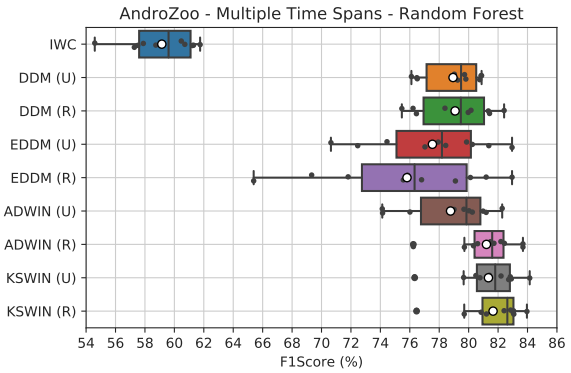
Figure 4.10: **F&F (U)pdate and (R)etrain prequential error and drift points as time goes by when using Adaptive Random Forest.** Despite being similar in some points, fewer drift points are detected when retraining the feature extractor, reducing the prequential error and increasing classification performance.

there might be potential biases from the use of the very same training and test sets. By proposing a new experiment, we mitigated the risk of biasing evaluations, since we have tested our solutions under different circumstances by using multiple time spans and reporting the average result.

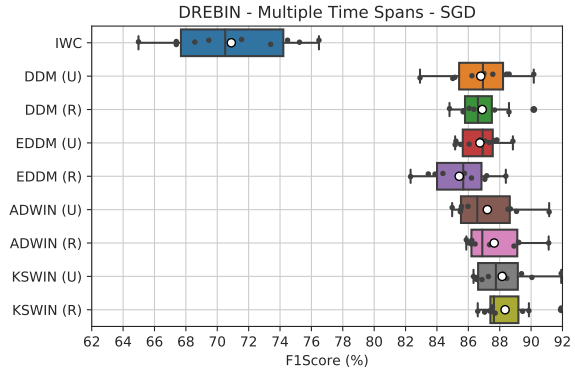
To evaluate the effectiveness of our approach in multiple conditions, we applied our data stream pipeline to different training and test sets of our dataset: we splitted the two datasets (sorted by the samples' timestamps) into eleven folds (thus 10 consecutive epochs), with every fold containing the same amount of data, similar to a k-fold cross-validation scheme. However, instead of using a single set for training as done in each iteration of k-fold cross-validation, we increment the training set i with the fold $i+1$, and remove it from the test set at every iteration. This way, we create a cumulative training set and simulate the same scenario as the previous experiment, but starting the stream in different parts. In the end, we produced ten distinct results that present the effectiveness of our method under varied conditions, which worked as a k-fold cross-validation. However, we focused on collecting the F1Score of each iteration).



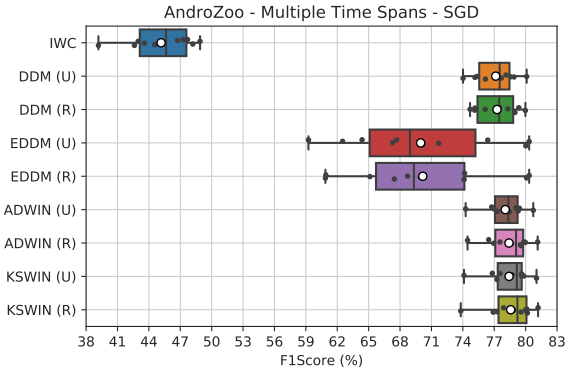
(a) Adaptive Random Forest and DREBIN dataset.



(b) Adaptive Random Forest and AndroZoo dataset.



(c) SGD and DREBIN dataset.



(d) SGD and AndroZoo dataset.

Figure 4.11: **Multiple Time Spans.** F1Score distribution when experimenting with ten different sets of training and test samples. Black dots represent the distribution of all F1Scores, whereas white dots represent their average for each applied method.

In the current experiment, we used all the methods presented in the previous section: both classifiers (Adaptive Random Forest and SGD), all drift detectors (DDM, EDDM, ADWIN, and KSWIN), and both methods ((U)pdate and (R)etrain). To accomplish a better presentation of the results, we chose a boxplot visualization containing the distribution of all F1Scores (black dots) and their average for each method (white dots), as shown in Figure 4.11.

In Figures 4.11(a) and 4.11(c), we present the results for the DREBIN dataset using Adaptive Random Forest and SGD as classifiers, respectively. The IWC method performed much better than the others with Random Forest; in the case of using ten folds to predict the last one, it reaches a better performance than the other methods, which indicates that the last fold may not be affected by concept drift. However, when we look at the other methods, KSWIN with retrain performed best, achieving a higher average F1Score and a low standard deviation. Also, as we saw in the previous section, SGD performed better than the Adaptive Random Forest applied to this dataset.

In Figures 4.11(b) and 4.11(d), we present the results for the AndroZoo dataset using Adaptive Random Forest and SGD as classifiers, respectively. In this scenario, IWC performance is much worse than any other method, which we believe is evidenced due to the complexity of this dataset (almost 285K samples). Again, in both classifiers, KSWIN with retrain method performed best, achieving the higher average F1Score, despite Adaptive Random Forest presenting better overall results and being almost 4 percentage points higher than SGD.

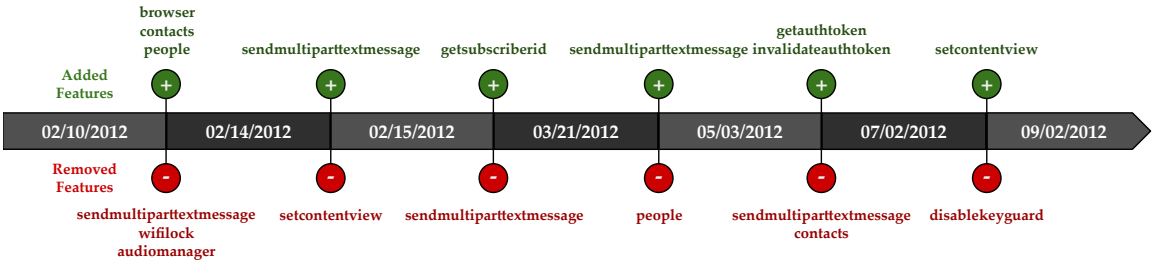
Finally, the analysis of the results allows us to observe that (i) the more data we have in the data stream, the most difficult it becomes for IWC to keep a good performance; (ii) using KSWIN drift detector with retrain is the most recommended method for static android malware detection data streams; and (iii) we ensure that our results do not have potential biases.

4.2.6.6 Understanding Malware Evolution

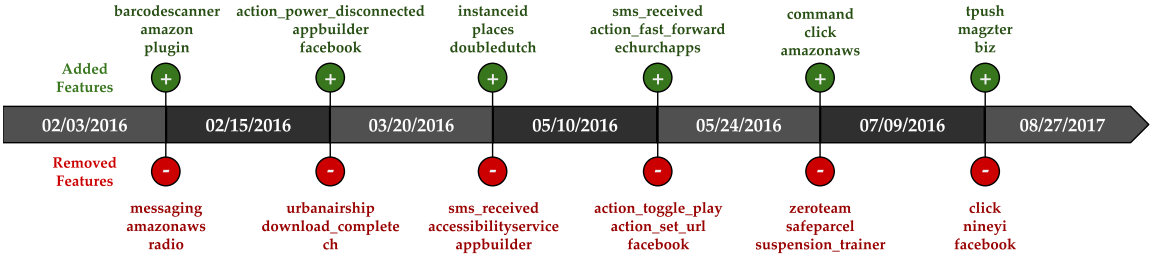
After we confirmed that concept drift is prevalent in these malware datasets, we delved into evolution details to understand the reasons behind it and the lessons that mining a malware dataset might teach us. To do so, we analyzed vocabulary changes over time for both datasets and correlated our findings of them.

DREBIN Dataset Evolution. Figure 4.12(a) shows API calls' vocabulary change for the first six detected drift points that actually presented changes (green and red words mean introduced and removed from the vocabulary, respectively). We identified periodic trends and system evolution as the two main reasons for the constant change of malware samples. The first occurs because malware creators compromise systems according to the available infection vectors (e.g., periodic events exploitable via social engineering). Also, attackers usually shift their operations to distinct targets when their infection strategies become so popular that AVs detect them. Therefore, some features periodically enter and leave the vocabulary (e.g., `setcontentview`). The second occurs due to changes in the Android platform, causing reactions from malware creators to handle the new scenario, either by supporting newly introduced APIs (e.g., `getauthtoken` (Android, 2018a)), or by unsupporting deprecated and modified APIs (e.g., deleted `keyguard` (Android, 2018b) or the deprecated `fbreader` (FBReader, 2018) intent). Handling platform evolution is required to keep malware samples working on newer systems, whereas ensuring maliciousness. In this sense, the removal of a feature like `DELETE_PACKAGE` permission from the vocabulary can be explained by the fact that Android changed permission's transparency behavior to an explicit confirmation request (Android, 2016), which makes malware abusing those features less effective. The most noticeable case of malware evolution involves SMS sending, a feature known to be abused by plenty of samples (Sarma et al., 2012; Hamandi et al., 2012; Luo et al., 2013). We identified that APIs were periodically added and removed from the vocabulary (e.g., `sendmultiparttextmessage`, part of the Android SMS subsystem (Android, 2018c), had its use restricted by Android over time, until being finally blocked (Cimpanu, 2018)). This will certainly cause modifications in newer malware and probably incur in classifiers' drifting once again. Therefore, considering concept drift is essential to define any realistic threat model and malware detector.

AndroZoo Dataset Evolution. We conducted the same experiments performed on the Drebin dataset on the AndroZoo dataset to evaluate dataset influence on concept drift occurrence, as shown in Figure 4.12(b). We highlight that developing a malware classifier for the AndroZoo dataset is a distinct challenge than developing a malware classifier for the Drebin dataset because the AndroZoo dataset presents more malware families (1.166 distinct families according to Euphony (Hurier et al., 2017) labels) than the Drebin dataset (178 distinct families according to Euphony labels). The difference in the dataset complexity is reflected in the number of drift points identified for each experiment. Whereas we identified 18 drift points for the Drebin dataset, the AndroZoo dataset presented 50 drift points. Despite the difference in the number of drifting points, the reasons behind such drifts have been revealed very similar when we delve into vocabulary change details. When we look to the `manifest_action` vocabulary, we notice that the `sms_received` feature leaves and returns to the vocabulary many times. This behavior reflects the arms race between Google and attackers for SMS sending permission (Cimpanu, 2018). This same occurrence has been observed in the DREBIN dataset. Similarly, when we look to the `manifest_category` vocabulary, we notice that the `facebook` feature also leaves and returns to the vocabulary many times. This happens because the attackers often rely on Facebook name for distributing "Trojanized" applications that resemble that original Facebook app. Although Google often removes these apps from the official store, attackers periodically come up with new ways of bypassing AppStore's verification routines, thus resulting in the



(a) Vocabulary change timeline for API calls using DREBIN dataset.



(b) Vocabulary change timeline for manifest and resources using AndroZoo dataset.

Figure 4.12: **Vocabulary changes for both datasets.** Many significant features are removed and added as time goes by.

presented infection waves regarding the Facebook name. Finally, this same behavior is observed in the `resource_entry` vocabulary regarding the `amazonaws` feature. Attackers often store their malicious payloads on popular cloud servers to avoid detection (Rossow et al., 2013). Even though Amazon sinkhole malicious domains when detected, attackers periodically find new ways to survive security scans, thus also resulting in the observed vocabulary waves.

The results of our experiments show that despite presenting distinct complexities, both datasets have undergone drift occurrences. This shows that concept drift is not a dataset-exclusive issue, but a characteristic inherently associated with the malware classification problem. Therefore, overall malware classification research work (and primarily the ones leveraging these two popular datasets) should consider concept drift occurrence in their evaluations to gather more accurate samples information.

4.2.7 Discussion

We here discuss our findings and their implications.

What the metrics say. While the accuracy of distinct classifiers is very similar in some experiments, we highlight that malware detectors must be evaluated using the correct metrics, as any binary classification. Therefore, we rely on f1score, recall, and precision to gain further knowledge on malware detection. AV companies should adopt the same reasoning when evaluating their classifiers. Experiments 4.2.6.2 and 4.2.6.3 indicate Android malware evolution, which imposes difficulties for established classifiers and corroborates the fact that samples changed. However, goodware generally kept the same concept over time, suggesting that creating benign profiles may be better than modeling everchanging malicious activities.

Feature drift, concept drift and evolution. Using a fixed ML model is not enough to predict future threats. In experiment 4.2.6.2, we notice a significant drop in f1score and recall (a metric that indicates the ability to correctly classify malware). Hence, a fixed ML model is prone to misclassify novel threats. Besides, continuous learning helps increasing detection but is still subject to drift, as shown by AndroZoo in experiment 4.2.6.3, when recall remains below 10% for a long period. If we compare experiments 4.2.6.3 and 4.2.6.4 when classifying DREBIN, it

is possible to observe that the Incremental Windowed Classifier update still outperforms DDM with the update when using Adaptive Random Forest, but is not better than DDM with retraining (except in precision). However, when comparing it with EDDM and ADWIN, we notice that both are significantly better, and it is still highly prone to drift (Figure 4.9). In response to these results, concept drift detectors help to improve classification performance. When comparing DDM with retraining and EDDM and ADWIN (with both update and retrain) to IWC (Table 4.3), we can see that the former outperformed the latter, advocating the need of using drift detectors. This is evident when using SGD, once IWC was outperformced by all other methods, and is even more evident when classifying AndroZoo, given that IWC lost the ability to detect new malware as time goes by. In this case, ADWIN with retraining was able to outperform all the other methods again, even a closely related work (DroidEvolver (Xu et al., 2019)), giving us insight that this is a valid method to be used in practice. In addition, even more important than just using a drift detector, reconsidering the classifier's feature set is required to completely overcome concept drift. According to experiment 4.2.6.4, we can infer that retraining the classifier and feature extraction models every time a drift occurs (using the data stream pipeline we proposed) is better than just updating only the classifier. This implies that not only the representation of malware becomes obsolete as time goes by, but also the vocabulary used to build them. It means that every textual attribute used by applications can change, i.e., new API calls, permissions, URLs, for example, emerge, requiring a vocabulary update, i.e., it indicates that discovering new specific features might help in increasing detection rates. Thus, we conclude that malware detection is not just a concept drift problem, but also in essence a feature drift detection problem (Barddal et al., 2017).

Our solution in practice. To implement our solution in practice, we need to model the installation, update, and removal of applications as a stream. This can be done by considering the initial set of applications installed in a stock phone as ground truth and thus subsequent deployment of applications as the stream. To reduce the delay between the identification of a drift point and the classifier update, ideally, multiple classifiers (the current one and the candidates to replace it) should be running in parallel. However, this parallel execution is too much expensive to be performed on endpoints. Therefore, we consider that the best usage scenario for our approach is its deployment on the App's Stores distributing the applications. Therefore, each time a new application is submitted to an App Store, it is verified according to our cycle. To cover threats distributed by alternative markets, this concept can be extended to any centralized processing entity. For instance, an AV can upload the suspicious application to a cloud server and perform its checks according to our cycle. To speed up the upload process, the AV can make the current feature extractor available to the endpoint so as the endpoint does not need to upload the entire application but only its feature. In this scenario, the AV update is not composed of new signatures, but of new feature extractors.

Drift and evolution are common problems in malware detection. Despite being more complex, (more apps and malware families) all characteristics present in DREBIN are drastically shown in AndroZoo, evidencing that feature drift, concept drift, and evolution are present in malware detection problems in practice and not only in a single dataset. This suggests that AV companies should keep enhancing their models and knowledge database constantly.

Limitations and future work. One of the limitations of our approach is that it relies on ground truth labels that may be available with a certain delay, given that known goodware and malware are needed in order to train and update the classifier (as any AV). Thus, future researches to reduce these delays are an important step to improve any ML solution that does not rely on their own labels, such as DroidEvolver (Xu et al., 2019) (outperformed by our approach).

It is important to note that our work considers only the attributes vectors provided by DREBIN (Arp et al., 2014) and AndroZoo (Allix et al., 2016b) datasets' authors. Due to this fact, we were unable to consider other types of attributes, such as API semantics (Zhang et al., 2020) or behavioral profiling (dynamic analysis) (Cai, 2020). Therefore, we cannot directly compare our work with theirs, due to different threat models. To compare different attributes using our data stream pipeline under the same circumstances, it is necessary to download all the APKs from both datasets. This is left as future work.

Finally, we make our data stream learning pipeline available as an extension of `scikit-multiflow` (Montiel et al., 2018), aiming at encouraging other researchers to contribute with new strategies that address feature and concept drift.

4.2.8 Conclusion

In this article, we evaluated the impact of concept drift on malware classifiers for Android malware samples to understand how fast classifiers expire. We analyzed ≈415K sample Android apps from two datasets (DREBIN and AndroZoo) collected over nine years (2009-2018) using two representations (Word2Vec and TF-IDF), two classifiers (Adaptive Random Forest and Stochastic Gradient Descent classifier) and four drift detectors (DDM, EDDM, ADWIN, and KSWIN). Our results show that resetting the classifier only after changes are detected is better than periodically resetting it based on a fixed window length. We also point the need to update the feature extractor (besides the classifier) to achieve increased detection rates, due to new features that may appear over time. This strategy was the best in all scenarios presented, even using a complex dataset, such as AndroZoo. The implementation is available as an extension to `scikit-multiflow` (Montiel et al., 2018)[5]. Our results highlight the need for developing new strategies to update the classifiers inherent to AVs.

[5] `https://github.com/fabriciojoc/scikit-multiflow`

5 EVALUATION

In this chapter, I show a novel evaluation approach to deal with security data that do not have labels available at the time they were collected, i.e., data that needs to be verified and have a delay to be labeled. In the paper presented here, I show that in a realistic scenario where labels are provided with delay and may never be available, the model performance drops significantly, improving when using just a few labels more than all of them. I consider this an important finding because knowing which data to label reduces processing time spent with model training and improves classification performance over time by selecting the best samples to label for a given time t. I also identify problems with the most popular drift detectors in the literature when applied to security data, given that they work solely based on accuracy and the development of new drift detectors that consider class imbalance problems are needed in these scenarios. Finally, the framework SPICE (Streaming Problems Investigation for Cybersecurity Evaluations) is proposed as an open-source library for data stream evaluation that includes modules to simulate delayed labels, labeling probability, and efficiently compute the AUC-PR by using a red-black tree.

5.1 THE SPICE MUST FLOW: CHALLENGES OF MODELLING SECURITY DATA AS STREAMS

This paper was submitted for publication to the USENIX Security conference.

Fabrício Ceschin[1], David L. P. Gomes[1], Eduardo Victor Lima Barboza[1], Paulo R. Lisboa de Almeida[1], André Grégio[1]
[1]Federal University of Paraná, Brazil
{fjoceschin, dlpg21, evlbarboza, paulo, gregio}@inf.ufpr.br

5.1.1 Abstract

The peculiarities of security data pose several challenges for evaluating classification solutions and to understand how they perform in the real world. When considering intelligent, adaptive solutions, even the best practices of traditional machine learning (ML) may produce erroneous conclusions that do not represent the reality of data. In this paper, we propose to model security data as streams to overcome current classification issues, such as those related to distribution and evolution of sample data. We discuss challenges faced by online approaches and how they differ from offline ones. To do so, we apply our security data stream model to popular Android and MS Windows malware datasets, critically analyzing the results to validate our approach and understand its extent and limitations based on closely related state-of-the-art work. We identify problems of the most popular drift detectors when applied to security data, since they are solely based on accuracy. We show that model performance drops significantly in a realistic scenario where labels are provided with delay and may never be available. Finally, we implement and publicly release the SPICE framework, an open-source library for data stream evaluation that includes modules to simulate these realistic scenarios and efficiently compute evaluation metrics.

5.1.2 Introduction

The constant increase in the availability of interconnected computer devices, mobile or not, creates the motivation for attackers to develop malware and other exploit codes. Malware attacks lead to a massive amount of security data, which has to be collected and analyzed so that useful information could be learned either to update current defense solutions or to support future counter-measures. In parallel, machine learning techniques have been widely applied to security-related classification problems, often without considering the specificities of this type of data. Since malware evolves and attacks never cease, we face a scenario in which sample variants arise continuously, creating the need for constant evolution also for the classification models produced.

Pendlebury et al. (Pendlebury et al., 2019) discuss the identification of space and time experimental bias; the former is related to unrealistic scenarios due to the train/test ratio of malware vs. goodware samples, whereas the latter occurs due to temporally inconsistent evaluation caused by the use of data "from the future" (test) to train models. Both pitfalls are discussed in detail in (Arp et al., 2022) as *sampling bias* and *data snooping*, respectively. However, another major problem of security data classification problems, especially those involving malware, is that the sheer amount of variants or closely related families of samples also leads to biased results. The insertion of lots of variants/related samples in training may boost metric rates during validation and testing, thus inaccurately representing the scenarios in which the model will be applied in the real world.

Moreover, the seasonality of malicious campaigns causes an interesting effect: variants of the same (or closely related) family of samples appear in the wild and rapidly increase in quantity for some time, then become less and less frequent as antiviruses mature their detection signatures/heuristics for those families. During the fade out, new campaigns arise and there is an overlap of the unknown variants of families that bypass current detection schemes with the previous campaigns' samples that are being replaced for effective ones. In addition, "family" labels assigned by antiviruses are not as important (from the users' point of view) as to detect a suspicious sample simply as "malware".

Since malware campaigns come and go with hundreds or thousands of variants released daily, the flow is continuous from the perspective of a detection solution. Therefore, this oscillatory movement of families in and out could be seen as a malware stream. That said, in this paper, we (i) propose to model security data as streams to overcome current classification issues, (ii) discuss challenges faced by online approaches and how they differ from offline ones, and (iii) apply our proposal to popular Android and MS Windows malware datasets, critically analyzing the results either to validate the approach we took and to understand its extent and limitations based on closely related state-of-the-art work. More specifically, our contributions are threefold:

- We identify problems with the most popular drift detectors in the literature when applied to security data, given that they work solely based on accuracy. We experimentally verify that the development of new drift detectors that consider class imbalance problems are needed in realistic scenarios.
- We show that in a realistic scenario where labels are provided with delay and may never be available the model performance drops significantly. Sometimes, the performance is better with just a few labels than all of them. It is an important finding because knowing which data to label reduce processing time spent with model training and improve classification performance over time by selecting the best samples to label for a given time t.
- We implement and publicly release the SPICE (Streaming Problems Investigation for Cybersecurity Evaluations) framework, an open-source library for data stream evaluation that includes modules to simulate delayed labels, labeling probability, and efficiently compute the AUC-PR by using a red-black tree.

5.1.3 Preliminaries

In this section, we motivate this work presenting the threat model and details about datasets used in the evaluation of our proposal, as well as the extracted features and model chosen to be the base classifier.

5.1.3.1 Threat Model

Our threat model considers machine learning classifiers whose aim is to detect security threats. In our experiments, those classification models can be seen as "malware detectors" for Android and MS Windows operating systems. We consider that this type of model is maintained by a centralized server, which is also responsible for continuously updating these models with new labeled information, as well as distributing them to multiple clients that use them to detect threats. It is worth emphasizing that our goal is to show how to correctly simulate a threat detection solution as a stream, considering that: (i) we operate in an adversarial environment; (ii) the data

has a non-stationary distribution, i.e. suffers from concept drift; and (iii) the models do not have their labels available at the same time as the feature vectors. Aside from malware detection, it is worth mentioning other flavors of detection (e.g., spam, intrusion/attacks, and misleading information) as problems that may use this very same threat model.

5.1.3.2 Datasets

To validate our proposal and conduct this paper's experiments, we used three popular datasets consisting of malicious samples for two distinct operating systems (Android and MS Windows). Those datasets present different characteristics and sum up to $1,613,281$ samples altogether. All samples had their static attributes extracted from the original Android Package (APK) and Portable Executable (PE) files, respectively. In Table 5.1, we describe the distribution of each dataset. Notice that although the MS Windows dataset is completely balanced (which is something that probably will not happen in real-world, improving the results for this dataset), both Android ones contain fewer malware samples, reaching a combined rate of 18.36% of malware on average, thus corroborating the Android malware landscape range (6% to 18.8%) reported in (Pendlebury et al., 2019). Finally, DREBIN dataset may be the most challenging dataset, given that it contains less malware information than AndroZoo.

Table 5.1: **Data Distribution.** Distribution of "goodware" and "malware" samples within the three datasets (DREBIN/Android, AndroZoo/Android, and EMBER/MS-Windows).

Dataset	Goodware	Malware	Total
DREBIN (Android)	123,453 (95.7%)	5,560 (4.3%)	129,013
AndroZoo (Android)	213,928 (75.25%)	70,340 (24.75%)	284,268
EMBER (Windows)	600,000 (50%)	600,000 (50%)	1,200,000
Total	937,381 (58.1%)	675,900 (41.9%)	1,613,281

The DREBIN dataset (Arp et al., 2014) consists of $123,453$ goodware and $5,560$ malware ($129,013$ APKs), of which $96K$ of the apps came from Google Play, $19.5K$ from Chinese markets, $2.8K$ from Russian markets, and $13K$ from other sources, including all samples from malgenome (Zhou and Jiang, 2012). The dataset provides ten textual attributes for each sample, which were extracted from Android APKs. Examples of DREBIN's attributes include the list of API calls, permissions, and URLs. In this work, we used a variation of the original dataset that also includes the timestamps of the samples (Ceschin et al., 2022) as an 11^{th} attribute, collected from VirusTotal samples reports (Total, 2019). In Figure 5.1, we present DREBIN's distribution over time, with samples ranging from 2009 to 2014.

The AndroZoo dataset has almost 21 million apps collected from Google Play, Anzhi, AppChina, Imobile, and AnGeeks (Allix et al., 2016a). In this paper, we use a subset of AndroZoo that consists of data collected from reports using the discontinued AndroZoo APK Analysis API (Ceschin et al., 2022). Thus, our version of AndroZoo has $213,928$ benign and $70,340$ malicious applications (totaling up to $284,268$ samples), but we kept the same goodware and malware distribution as the original dataset. The dataset is composed of eight textual attributes extracted from Android APKs, such as resource names, source code classes and methods, manifest permissions, etc. We show its distribution over time in Figure 5.2, which spans from 2016 to 2018 according to the samples' VirusTotal *first seen dates*.

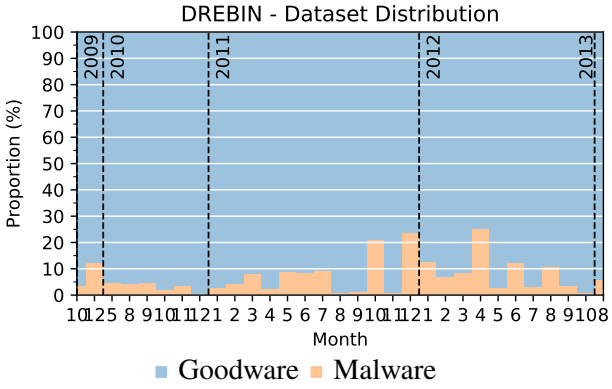


Figure 5.1: **DREBIN Distribution.** Goodware is the majority class; malware reached ≈ 30% only in April 2012.

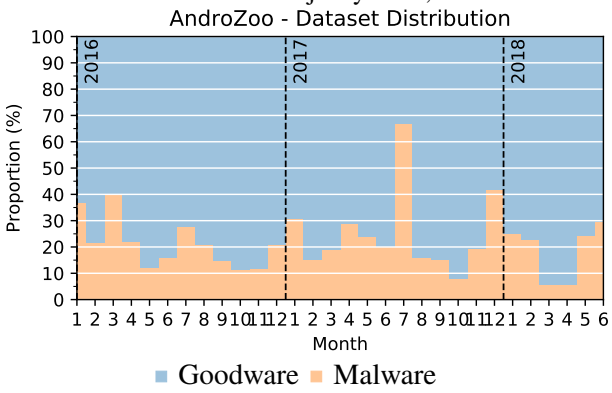


Figure 5.2: **AndroZoo Distribution.** Goodware is again the majority class overall, except for July 2017.

Finally, the EMBER dataset has 1,200,000 MS Windows executable samples: 600,000 malware and 600,000 goodware (Anderson and Roth, 2018). It contains attributes extracted from the PE header, as well as the list of libraries and functions used by the binaries (Ceschin et al., 2020a). Despite the absence of VirusTotal (Total, 2019)-collected timestamps, the EMBER dataset is already ordered by month and year of the samples' release date. We show the distribution of malware and goodware of this dataset over time (2017 to 2018) in Figure 5.3.

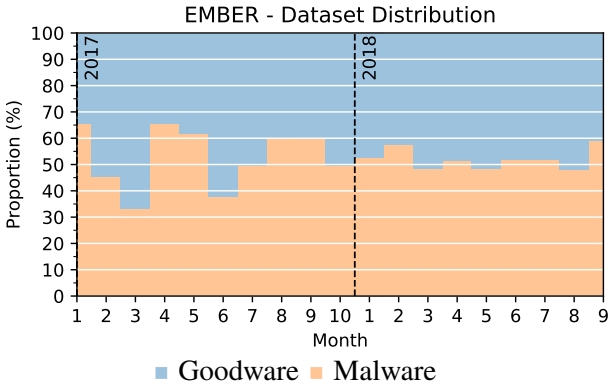


Figure 5.3: **EMBER Distribution.** The total of each dataset class is balanced, but it may present a minor to moderate imbalance on some months.

5.1.3.3 Features and Model

Considering that the aforementioned datasets contain three types of attributes (numerical, categorical, and textual), we used the same strategy for all of them according to the literature (Ceschin

et al., 2018; Ceschin et al., 2022, 2020a), i.e., we extracted features from the categorical and textual attributes and concatenated those with the numerical features. Thus, we transformed all the categorical features into a one-hot encoding array and the textual features were used as input to TF-IDF (term frequency–inverse document frequency) (Jones, 1972), which transforms every text into a sparse array containing the TF-IDF values for the 100 top words in all the texts (the ones with most frequency).

We used the Hoeffding Tree (Hulten et al., 2001) classifier as the base model for all experiments presented in further sections of this paper. Hoeffding Trees have been commonly applied in data streams problems due to its ability to deal with incremental learning, which will be the case in our everchanging scenario of arriving malware variants/families over time. Among its advantages in the data stream scenario, we have its ability to process each instance once, it does not need to store data in the memory and it does not grow in space when new instances arrive, unless there are more information in data (Lu et al., 2020). It is important to note that, in all experiments considering data streams, the models are initially trained with 20% of the stream and then tested with the remaining 80%.

5.1.4 Why Using Standard ML Fails

When evaluating ML solutions for cybersecurity, it is common to see experiments that use a static model and follow ML best practices, such as 10-fold cross-validation. Although it seems straightforward and correct, such an approach may backfire in security contexts (Cavallaro, 2019) and present different results than expected due to the nature of cybersecurity problems, something not expected in most standard ML solutions.

To illustrate these problems, we first applied the standard ML evaluation technique (10-fold cross-validation) to our datasets, similar to what many researchers do in the literature. The results are reported in Table 5.2. In all datasets, we achieved more than 86% of accuracy and more than 80% of FScore. In the Android landscape, recall is the lowest metric, which means that the number of false negatives is relatively high, indicating that more malware are classified as goodware. On the other hand, in the Windows landscape, we have a more balanced scenario, but with a higher recall than precision, which means that more goodware are classified as malware. All these conclusions seem convincing at first, but do they really reflect a solution in the real world?

Dataset	Accuracy	FScore	Recall	Precision
DREBIN	98.66%	81.35%	70.67%	96.91%
AndroZoo	91.03%	80.41%	74.42%	87.47%
EMBER	86.21%	86.37%	87.48%	85.30%

Table 5.2: **10-Fold Cross Validation Results.** Overall results of a 10-Fold Cross Validation evaluation.

The previous question was already explored in the literature (Ceschin et al., 2022; Arp et al., 2022) and the answer is no, it does not reflect a real-world solution due to the data snooping problem, given that 10-fold cross-validation exposes the model to future data during training, something that will never happen in reality. A valid way to evaluate the model without exposing it is to create a static one trained with only the first portion of the data (ordered by their timestamps) and test it with the remaining data, similar to (Pendlebury et al., 2019). This way, we are simulating a real scenario where we collect known attacks, and use them to train a model to detect future threats. We show the results of such approach, using the first 20% of the datasets to train the models, in Table 5.3. As noticed, all metrics are much worse when compared to 10-fold

cross-validation, which shows that using this standard machine learning evaluation technique is not indicated in cybersecurity contexts. For instance, in practice, the recall for the DREBIN dataset was only 11.07%, which means that the number of false negatives is much higher than expected by 10-fold cross-validation and just a few malware will be detected.

Dataset	Accuracy	FScore	Recall	Precision
DREBIN	95.80%	19.79%	11.07%	93.36%
AndroZoo	77.67%	58.94%	66.29%	53.07%
EMBER	81.46%	81.07%	79.71%	82.68%

Table 5.3: **Static Model Results.** Overall results considering samples' timestamps and a static model trained using the first 20% samples of each dataset.

Finally, knowing how to correctly model and evaluate a cybersecurity solution is crucial for its use in real-world. That is why we advocate for solutions that are modeled and evaluated as streams as we do in this work.

5.1.5 Modeling Security Problems as Streams

As already mentioned, in (Arp et al., 2022) the authors cite the data snooping problem, where a model is trained with data that is not available in practice – e.g., train the model using data available at time $t + 1$ to classify data available at t, thus using the future to classify the past. To avoid this problem, the authors in (Arp et al., 2022) suggest considering the temporal dependencies of the data when splitting it between training and test sets, as we did in the static model experiment in the previous section.

We extend this rationale to advocate that machine learning in security must be modeled as stream classification problems, where a large static dataset is not available to train/test the classifier at the beginning of the test. Instead, the instances are provided sequentially in a time-dependent problem, in a possibly infinite data stream (Gomes et al., 2017a).

This imposes several real-world challenges such as (i) new labeled instances may arrive over time, and the model must be updated using these instances; (ii) the instances (train or test) cannot be stored indefinitely in the system, since only limited Memory and CPU resources are available, but the stream size is indefinite. (iii) due to the nature of the arms race in security problems, where attackers evolve their techniques over time, it is expected that the relations between the features and classes may change over time, leading to Concept Drifts (Gomes et al., 2017a; Lisboa de Almeida et al., 2020).

To model the problem more formally, consider a stream $S = \{\mathbf{x_1}, \mathbf{x_2}, \mathbf{x_3}, \dots\}$, where x_t is the feature vector of an instance that arrived and must be classified at time t. In such problems, it is often the case that the true label y of the instance that arrived at time t becomes available at time $t + k$, where $k \in \mathbb{N}^+$. Thus, the model may be updated over time, using previously seen instances.

This modeling of course forces the presence of the timestamps of the instances in the dataset. For malware detection problems, it is highly recommended to use VirusTotal (Total, 2019) first-seen date as a time stamp, as many works in literature do (Pendlebury et al., 2019; Ceschin et al., 2022). This way, it is possible to create a stream where samples are ordered by these timestamps, simulating a real-world solution.

Some works (Xu et al., 2019; Ceschin et al., 2022; Narayanan et al., 2017) consider that the true label of the instance that arrived at time t is available at $t + 1$, in a test-then-train scenario, as shown in Algorithm 1 (Bifet et al., 2018). Considering a model M and a stream S of

samples (a list of tuples (X, y), where X are the feature vectors and y, the labels of the instances, the algorithm initially trains M with the beginning of stream S (ordered by time) and initialize the metrics. Then, we iterate over the remaining samples of S, getting every feature vector X and label y, using X as input to the model M to generate a new prediction $pred$, used to update the metrics. At the end of each iteration, the model is updated using the feature vector X and its label y. We argue that this is unfeasible in the real world for security solutions. How can we classify one instance, and know its true label when the next one arrives (maybe some seconds after the previous one)?

Algorithm 1 Standard Test-Then-Train Stream Evaluation

Require: model M and a stream S of samples (list of tuples (X, y), where X are the feature vectors and y, the labels of the instances)

1: Train M with beginning of S
2: Initialize metrics
3: **while** S has samples **do** ▷ Remaining samples of S
4: $X, y \leftarrow S$ next sample
5: $pred \leftarrow M$ predicts X
6: Update metrics with prediction $pred$
7: Update M with current sample (X,y)
8: **end while**

Considering malware detection, the majority of the works use only a single snapshot of scanning results from platforms like VirusTotal (Total, 2019), without considering a given delay before using the labels, which may vary from ten days to more than two years (Zhu et al., 2020). A recent study from Botacin et al. analyzed the labels provided by VirusTotal for 30 consecutive days from two distinct representative malware datasets (from Windows, Linux, and Android) and showed that solutions took 19 days to detect the majority of the threats (Botacin et al., 2020b).

Thus, we suggest that the true label of the instance that arrived at time t only becomes available for training at time $t + 19\ days$. This configuration may lead to closer to the real-world configurations, and avoid problems such as the instability of the labels of the malware when new malware is discovered (Arp et al., 2022).

Notice that this protocol is more general than the one proposed in (Pendlebury et al., 2019), since it works for streams of data (differently from (Pendlebury et al., 2019), that must deal with streams of batches). Nevertheless, the proposed protocol is compatible with the constraints proposed in (Pendlebury et al., 2019). The code for our protocol is presented in Algorithm 2. Differently from Algorithm 1, here it also requires the timestamp t of the sample in the stream S, as well as the delay k, which is the time it takes for a sample to be labeled, in our case, $k = 19$ days. At the beginning, the model M is also trained with the start of the stream S and the metrics are initialized. The only difference here is that we save the last timestamp seen by the model M to keep track of the time and we initialize a FIFO (First-In-First-Out) queue Q used to save all the instances while iterating over the stream. Thus, once a new sample is obtained from the stream S, we update the $time$ with its timestamp t, and instead of updating the model after updating the metrics as we do in Algorithm 1, we add the current sample to the queue Q. This queue is used at the end of the iteration to check if there are samples with labels available, i.e., if samples in the queue have their timestamps with delay ($t + k$) bigger than the current $time$, if so, they can be unqueued and used to update the model M. This way, we can simulate a real-world scenario where samples' feature vectors X are labeled after a given time and then presented to the model M with their right label y.

Algorithm 2 Test-Then-Train Stream Evaluation with Delayed Updates

Require: model M, a stream S of samples (list of tuples (X, y, t), where X are the feature vector, y, the labels of the instances, and t their timestamps), and a delay k (the time it takes for a sample to be labeled)

```
1: Train M with beginning of S
2: Initialize metrics
3: time ← last timestamp seen by M
4: Q ← [ ]                                   ▷ Initialize queue of instances Q
5: while S has samples do                    ▷ Remaining samples of S
6:     X, y, t ← S next sample
7:     time ← t                              ▷ Update current time
8:     pred ← M prediction for X
9:     Update metrics with prediction pred
10:    Q.put(X, y, t)                        ▷ Add current sample to queue
11:    if Q.size() > 0 then
12:        X, y, t ← Q.get(0)                ▷ Check first in queue
13:        while t + k <= time do            ▷ Label is available?
14:            X, y, t ← Q.pop(0)            ▷ Remove it
15:            Update M with current sample (X,y)
16:            X, y, t ← Q.get(0)            ▷ Check next
17:        end while
18:    end if
19: end while
```

Although the previous evaluation seems realistic, we can still think of a scenario where the labels will never be available, even after a given delay k, where $k = \infty$. In malware detection tasks, using VirusTotal (Total, 2019) may be very expensive due to the API usage cost (Pendlebury et al., 2019). Also, according to (Arp et al., 2022), ground-truth labels may be inaccurate, unstable, or erroneous, which may affect the overall classification performance of ML-based solutions. Thus, it is also very important to consider that, aside from the delay in the labels, they may also never be available. To consider this scenario, we extended previous evaluation algorithm with a probability function that only updates the model $p\%$ of times, where p is the probability of labeling new samples seen while iterating over the stream S, as shown in Algorithm 3, implemented by SPICE.

We advocate the use of Algorithm 3, implemented by SPICE framework, when evaluating security data as streams because it simulates multiple conditions that may happen in a real-world solution.

5.1.6 Metrics of Performance

Selecting suitable metrics to show the results is arguably the most critical task in any scientific research. When considering the classification of streams containing malware and goodware, we must be careful with the unbalanced and stream properties of the data. Otherwise, we will certainly generate biased results that do not accurately describe the real quality of the methods.

Metrics for Unbalanced Data. To give us better insight, let us consider the *prior* probabilities of commonly used datasets (the proportion between goodware and malware). Table 5.1 shows the *prior* probabilities considering the entire datasets, while Figures 5.1, 5.2 and 5.3 show the *prior* probabilities calculated over time (monthly).

Algorithm 3 SPICE: Test-Then-Train Stream Evaluation with Delayed Updates and Labeling Probability

Require: model M, a stream S of samples (list of tuples (X, y, t), where X are the feature vectors, y, the labels of the instances, and t their timestamps), a delay k (the time it takes for the samples to be labeled), and a probability p of labeling samples

```
1:  Train M with beginning of S
2:  Initialize metrics
3:  time ← last timestamp seen by M
4:  Q ← [ ]                                   ▷ Initialize queue of instances Q
5:  while S has samples do                    ▷ Remaining samples of S
6:      X, y, t ← S next sample
7:      time ← t                              ▷ Update current time
8:      pred ← M prediction for X
9:      Update metrics with prediction pred
10:     Q.put(X, y, t)                        ▷ Add current sample to queue
11:     if Q.size() > 0 then
12:         X, y, t ← Q.get(0)                ▷ Check first in queue
13:         while t + k <= time do            ▷ Label is available?
14:             X, y, t ← Q.pop(0)            ▷ Remove it
15:             if true(p) then               ▷ True with p probability
16:                 Update M with current sample (X,y)
17:             end if
18:             X, y, t ← Q.get(0)            ▷ Check next
19:         end while
20:     end if
21: end while
```

As one can observe in Table 5.1 and Figures 5.1, 5.2 and 5.3, the datasets are often highly unbalanced to the goodware class. This indicates that using metrics such as accuracy may lead to a biased view. For instance, considering the DREBIN dataset, an uninformed classifier that always predicts the class of the instance as a goodware will have an accuracy of 95.7% (as shown in Table 5.1), even though that classifier is unable to find any malware.

To enrich this discussion, consider the confusion matrix for classification between goodware and malware problems in Table 5.4. Notice that a classification threshold must be fixed in the *a posteriori* probabilities to generate such a table. In other words, probabilistic classifiers often generate a score $\in [0..1]$ indicating the probability of the tested instance belonging to the positive class. For equiprobable and cost-insensitive problems, a threshold of 0.5 is often used – i.e., if the score value is above or equal to 0.5, the instance is classified as belonging to the positive class, or the negative class otherwise.

Malware detection problems are often biased to one class and may be cost-sensitive (e.g., the cost of misclassifying a malware may be greater than the cost of misclassifying a goodware). Metrics such as the F_1 *score* (Lipton et al., 2014) and the *balanced accuracy* (Brodersen et al., 2010) can overcome some of the shortcomings of accuracy in cost-sensitive problems. Nevertheless, they analyze the performance using a fixed classification threshold.

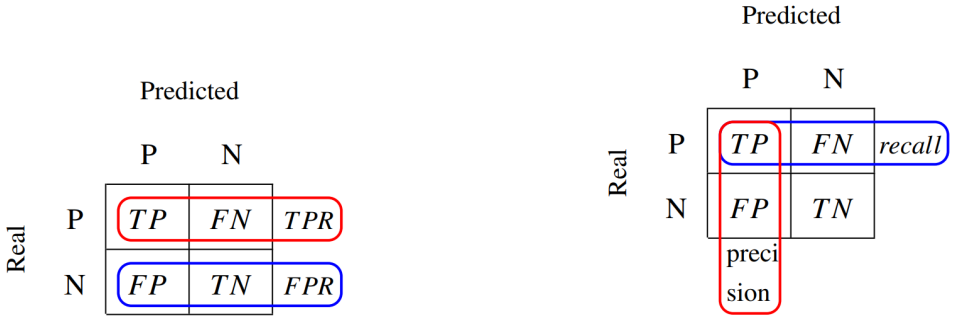
Metrics such as the Area Under the ROC Curve (AUC-ROC) and the Area Under the Precision-Recall Curve (AUC-PR) try to give a better insight into the classifiers' competence over all possible classification thresholds. The AUC-ROC is computed by first generating a point

		Predicted Class: Positive (Malware)	Predicted Class: Negative (Goodware)
Real Class	Positive (Malware)	True Positive (TP) Detected Malware	False Negative (FN) Undetected Malware
	Negative (Goodware)	False Positive (FP) Goodware detected as Malware	True Negative (TN) Goodware detected as Goodware

Table 5.4: **Malware x Goodware.** Confusion Matrix.

in a Cartesian space for every possible classification threshold, giving the ROC curve. In a ROC curve, the x coordinate contains the False Positive Rate = $FPR = \frac{FP}{FP+TN}$, while the y coordinate contains the True Positive Rate = $TPR = recall = \frac{TP}{TP+FN}$ for a given threshold. The AUC-ROC is given by the area (integration) of the ROC curve (Brown and Davis, 2006; Davis and Goadrich, 2006).

The AUC-PR is computed similarly. For the PR Curve, the x coordinate contains the $recall$, while the y coordinate contains $precision = \frac{TP}{TP+FP}$ for every possible threshold (Davis and Goadrich, 2006). The confusion matrix variables involved in the computation of the AUC-ROC and AUC-PR values are shown in Figures 5.4(a) and 5.4(b), respectively.



(a) AUC-ROC variables.

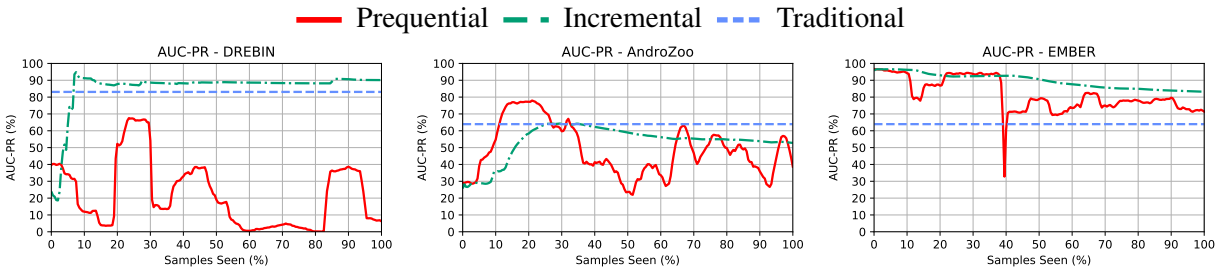
(b) AUC-PR variables.

Figure 5.4: **Confusion matrix variables involved for the ROC and PR curves.** Blue for the x coordinate and red for the y coordinate.

Notice that the ROC curve uses the true negatives (i.e., goodware correctly detected as goodware). As discussed in (Davis and Goadrich, 2006), this can create an optimistic view of an algorithm's performance if there is a significant skew (i.e., much more examples of goodware when compared to malware). To overcome this issue, we advocate the use of AUC-PR as a more effective metric since it does not take into account the true negatives (see Table 5.4(b)), and can accommodate in a single value the $precision$ and $recall$.

Adapting the Metrics for Streams. The AUC-PR can be a good metric to show the final result in the stream. Nevertheless, some precautions must be taken if we want to check the classifier's performance over time. Otherwise, we will be blind to sudden changes and take a long time to react. To illustrate this, consider the experiment inspired in (Brzezinski and Stefanowski, 2017), showing a stream containing a concept drift at time K. The classifier performance at time t is measured using the AUC-PR considering: (i) incrementally, using all instances that arrived between times 0 and t; (ii) prequentially, using a window of size $w = 10.000$, i.e., using the instances that arrived between $t - 999$ and t; and (iii) traditionally as any batch problem, computing a unique metric using the true labels and scores of all instances after processing the

stream. We show the results of this experiment, considering a static model with no updates over time, for all datasets in Figure 5.5.



(a) **DREBIN.** Drastic drops are seen in AUC-PR in interval 55%-80%, which may indicate concept drifts.

(b) **AndroZoo.** Major drop in AUC-PR is seen during 45%-50%, indicating a possible concept drift.

(c) **EMBER.** A huge drop in AUC-PR is seen near 40%, also indicating concept drifts.

Figure 5.5: **AUC-PR.** Prequential, Incremental, and Traditional AUC-PR comparison. The Prequential metric can capture temporal changes and classifier's performance over time, different from the other metrics.

As one can observe, besides the concept drift that severely affected the classifier performance (as noticed by the prequential AUC-PR), the change was not readily perceptible when considering all instances (Incremental AUC-PR). In this case, errors being made just after the drift were diluted by the good performance of the classifier before the drift. When considering the entire stream, the change in the classifier performance only becomes perceptible when the period after the drift becomes similar in size to the period before the drift. In addition, the traditional metric cannot give us any insight of the classifier performance over time, only an overview.

We advocate using prequential metrics to generate a plot containing the classifier performance over time. Of course, this can be computationally expensive. In such scenarios, an alternative is to compute the metric in batches of w instances, as shown in (Brzezinski and Stefanowski, 2017). Nevertheless, whenever possible, the true prequential metric should be used. In (Brzezinski and Stefanowski, 2017) it is presented an efficient algorithm to compute the prequential AUC-ROC. In this paper, we use and extend this strategy to compute the prequential AUC-PR.

To calculate the recall and precision of a score (when it is the threshold) the necessary variables are the true positives (TP), false positives (FP), and false negatives (FN). However, true positives plus false negatives equals the total number of positive samples, thus, the recall can be calculated as $\frac{TP}{Total positives}$. With this in mind and aiming at fewer operations, we do not need to calculate true positives and false positives in every iteration for a given threshold: those rates only have to be updated if a higher score is added (if positive, increment the TPs, else increment the FPs) or removed (if positive, decrement the TPs, else decrement the FPs). The use of a red-black tree is justified in (Brzezinski and Stefanowski, 2017), and we maintain the tree to efficiently calculate the true positives and false positives of a score when it is the threshold, since it is needed to iterate on the scores higher than the threshold. The prequential AUC-PR implemented by SPICE is presented in Algorithm 4.

For each incoming sample of the stream, the related score is added to the sliding window and to a red-black tree (lines 6-7), its TPs and FPs are calculated and the TPs and FPs of lower scores are updated (line 8). If the sliding window length is equal to the determined window size, the area under the Precision-Recall curve is calculated and the first element of the sliding window is removed (lines 31-32), updating the TPs and FPs of lower scores (line 33).

To calculate the AUC-PR, for each sample in the sliding window, it is calculated the recall and precision with the TPs and FPs related to the example (only if their sum is different

	SK-Learn AUC-PR	Our Prequential AUC-PR
Average by Iteration (s)	0.001	0.0002
Total (s)	107.01	21.76

Table 5.5: **AUC-PR Performance Comparison.** Comparison between scikit-learn AUC-PR and our prequential AUC-PR implementation.

than zero, since it is used as the denominator of the precision) and added to a points list (lines 16-25). Then, the points list is sorted by the recall of each point in descending order (line 26), and additional points are added in order to correctly build the curve (lines 27-28). Finally, the AUC-PR is calculated by integrating the curve using the trapezoidal rule (line 29).

As shown in (Brzezinski and Stefanowski, 2017), the prequential AUC-ROC performed fast, with a low computational cost. Since our prequential AUC-PR algorithm does not calculate the true positives and false positives in every iteration, i.e., it is updated when inserting or removing a score, it performs fewer operations to calculate the area under the curve. Therefore, when considering a stream of data, the calculation is made faster than computing the TPs and FPs in every iteration, as implemented in scikit-learn (Pedregosa et al., 2011) (using `precision-recall_curve` and `auc`).

To demonstrate the difference in time, we tested the Drebin dataset 10 times (with 103,211 instances) and calculated the average of the sum of each AUC-PR calculation method. The results are presented in Table 5.5. The scikit-learn method took 107.01 seconds to calculate all AUC-PRs, and our prequential AUC-PR, 21.76 seconds. Hence, to calculate the AUC-PR of a single iteration, scikit-learn method took, on average, 0.001 seconds, and our implementation, 0.0002 seconds. Thus, our method performed nearly five times faster.

Our efficient prequential area under the Precision-Recall curve calculation method is not implemented on the most used machine learning stream libraries, such as `River` (Montiel et al., 2021),`scikit-multiflow` (Montiel et al., 2018), and `Massive Online Analysis (MOA)` (Bifet et al., 2010). Thus, we present this algorithm as a contribution of this paper.

5.2 RESULTS

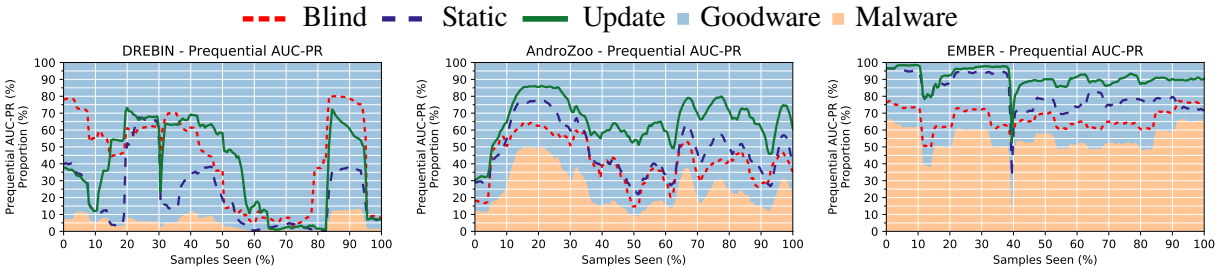
In this section, we show the results of applying the SPICE framework to the three selected datasets, considering (i) incremental learning, when we update the classifier with new samples, (ii) drift detection, when we update the classifier and use a drift detection strategy to handle concept drifts, and (iii) label probability, when we update the classifier in an iteration only with a given probability. In all three scenarios we consider updates without delay ($k = 0$) and with delay ($k = 19$ days).

5.2.1 Incremental Learning

In the incremental learning experimental scenario, in addition to the regular update strategy, we also compared it to the static model (with no updates while iterating over the stream) presented in Section 5.1.4, and a blind classifier that labels an instance according to the label of the previously seen instance. According to (Lisboa de Almeida et al., 2020), the blind classifier should be used as a baseline when employing the test-then-train protocol.

The results considering no delay ($k = 0$) and delay ($k = 19$ days) are presented in Figures 5.6 and 5.7, respectively, and the overall results, obtained with the real labels and scores

of all samples after processing all the streams, are presented in Table 5.6. When not considering delays, in all cases, updating the classifier outperformed the static classifier, which indicates that new samples are helping to detect new threats. Surprisingly, Using a blind classifier in DREBIN (61.97%) was even better than both updated (59.78%) and static (54.29%) models, which indicates that this dataset may have time dependencies. In contrast, when we consider delays, we have different scenarios: the blind classifier fails drastically and the static one is the best for the DREBIN dataset. In all datasets, the updated classifier had a significant drop in its performance over time, having its function closer to the static one. When comparing to the results with no delay, we can conclude that the sooner the samples are labeled, the better the model predicts new threats overall.

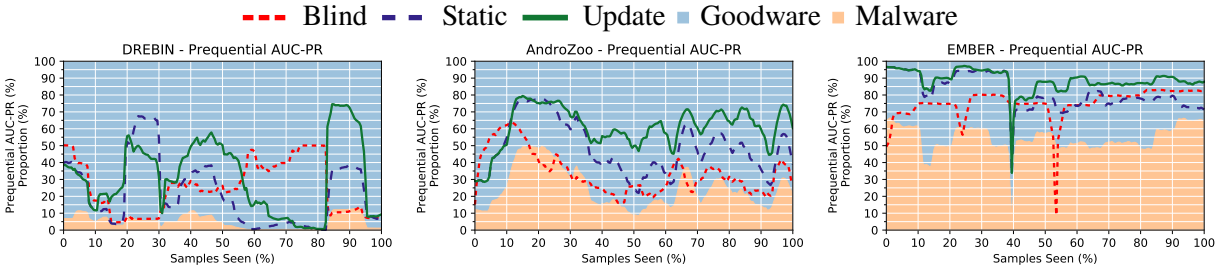


(a) **DREBIN.** The blind and Updated classifiers share the top results over time, with blind achieving the best overall.

(b) **AndroZoo.** The updated classifier is always the best option over time and has the best overall results.

(c) **EMBER.** The updated classifier is also the best option again over time and has the best overall results.

Figure 5.6: **Incremental Learning Results with no Delay.** Results considering no delay in the stream, i.e., labels are available at time $t + 1$. Prequential distribution of goodware and malware is also reported.



(a) **DREBIN.** Both blind and updated classifiers, which were the best without delay, had a significant performance drop.

(b) **AndroZoo.** The updated classifier is still the best, despite its performance function being closer to the static one.

(c) **EMBER.** The updated classifier is also the best again, but now the blind classifier is closer at some points to the static one.

Figure 5.7: **Incremental Learning Results with Delay.** Results considering a delay in the stream, i.e., labels are available only at time $t + 19$ days. Prequential distribution of goodware and malware is also reported.

5.2.2 Drift Detection

In the drift detection experiments, we included drift detectors in the stream pipeline to check their effectiveness over time when considering and not considering delays. We considered four drift detectors in our experiments. The first drift detector is the Random Trigger (RT), which signals a drift with a probability p for every new instance received, in our case $p = 1\%$ following experiments (Lisboa de Almeida et al., 2020). According to (Lisboa de Almeida et al., 2020) this method should also be used as a naive method for comparison with other since a trigger that fires at random should perform worse than a well-conceived one. Aside from this method, we also included the drift detectors DDM (Drift Detection Method) (Gama et al., 2014b), EDDM (Early Drift Detection Method), and ADWIN (ADaptive WINdowing) (Bifet and Gavaldà, 2007).

Dataset	Method	AUC-PR	
		No Delay	Delay
DREBIN	**Blind**	**61.97%**	17.52%
	Static	54.29%	**54.29%**
	Update	59.78%	49.20%
AndroZoo	**Blind**	45.99%	34.89%
	Static	63.75%	63.75%
	Update	**79.23%**	**70.94%**
EMBER	**Blind**	67.30%	61.15%
	Static	86.18%	86.18%
	Update	**92.01%**	**89.39%**

Table 5.6: **Overall Incremental Learning Results.** Best results are highlighted in bold. Overall, the update strategy was the best method.

The Random Trigger, DDM, and EDDM detectors were used following measures that trigger two levels: warning and drift. The warning level suggests that the concept starts to drift, then a background classifier is updated using the samples which rely on this level. The drift level suggests that the concept drift occurred, and the background classifier built during the warning level replaces the current classifier (Bifet et al., 2018). The ADWIN drift detector was used internally in the Hoeffding Tree (Hoeffding Adaptive Tree) to monitor the performance of the branches and to replace them with new branches when their accuracy decreases if the new ones are more accurate (Bifet and Gavaldà, 2009; Montiel et al., 2021). In addition, we also considered the standard updated classifier to compare it with the drift detection strategy. Figures 5.8 and 5.9 present the results over time when considering no delays ($k = 0$) and considering delays ($k = 19$ days), respectively, and Table 5.7 shows the overall results.

In the DREBIN dataset, ADWIN managed to start better than the update strategy (from 0% to 15%) and kept a very similar performance in the remaining time, presenting the best overall performance when not considering the delay (Figure 5.8(a)). In contrast, the drift detectors did not work well when considering the delays (Figure 5.9(a)) in the DREBIN dataset, all of them were outperformed by the traditional classifier update strategy.

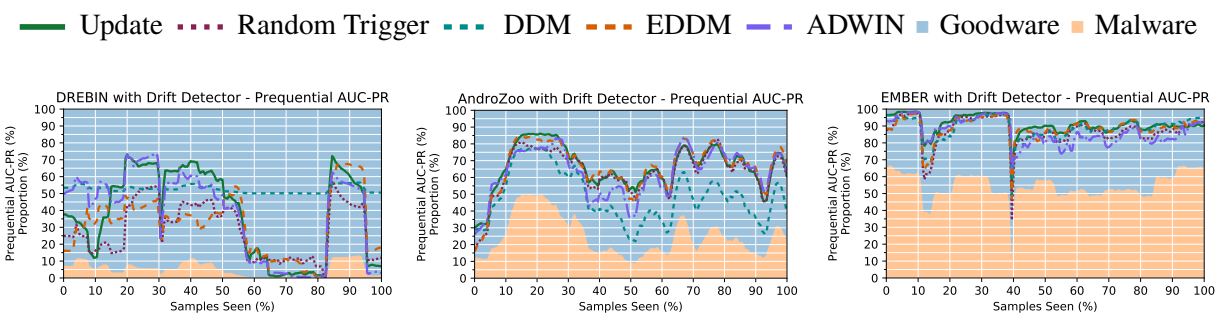
In the AndroZoo dataset, the updated classifier and EDDM performance were very close over time but, overall, the updated classifier still outperformed all drift detectors in both scenarios (with and without delays – Figures 5.9(b) and 5.8(b), respectively). Overall, the DDM was the worst strategy for this dataset, except for the random trigger when considering delays.

In the EMBER dataset, all methods were very close when not considering delays (Figure 5.8(c)), but again with a small overall advantage for the updated classifier, which presented the best result, and EDDM. When we consider the delay (FigureFigure 5.9(c)), a significant drop is seen in all methods again, but even more in DDM, which was the worst strategy. The best strategy when considering delays was updating the classifier.

In addition, the random trigger drift detector only outperformed other drift detectors in a few scenarios, which means that, on average, the drift detectors are working as intended, even though they are still performing worse than the standard update strategy. Also, in all datasets, the models were highly impacted by the delays, specially in scenarios where the class imbalance is extremely high as in the DREBIN dataset, which means that new strategies are needed to overcome this problem.

Dataset	Method	AUC-PR	
		No Delay	Delay
DREBIN	**Update**	59.78%	**49.20%**
	RT	53.38%	33.31%
	DDM	52.33%	47.01%
	EDDM	56.26%	35.28%
	ADWIN	**64.34%**	39.49%
AndroZoo	**Update**	**79.23%**	**70.94%**
	RT	74.83%	59.32%
	DDM	63.85%	63.85%
	EDDM	77.26%	64.26%
	ADWIN	74.97%	63.82%
EMBER	**Update**	**92.01%**	**89.39%**
	RT	89.81%	87.49%
	DDM	90.49%	85.34%
	EDDM	91.11%	87.79%
	ADWIN	89.50%	86.26%

Table 5.7: **Overall Drift Results**. Best results are highlighted in bold. Overall, the update strategy was the best.

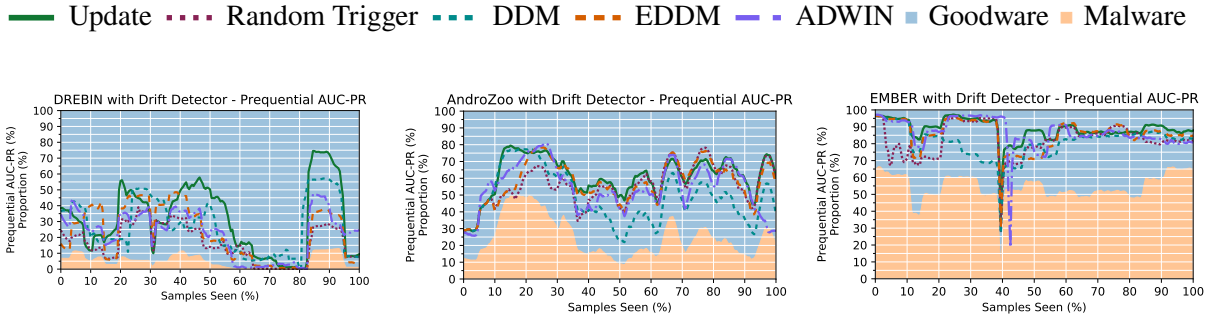


(a) **DREBIN.** Update and ADWIN were the best strategies, with the latter outperforming the former.

(b) **AndroZoo.** Update and EDDM were very close overtime, but the former outperforming the latter.

(c) **EMBER.** Update and EDDM were the best methods, with the former outperforming the latter.

Figure 5.8: **Drift Detection Results with no Delay.** Results considering no delay in the stream, i.e., labels are available at time $t + 1$. Prequential distribution of goodware and malware are also reported.



(a) **DREBIN.** When considering delays, the drift detectors did not perform well and were outperformed by the update strategy.

(b) **AndroZoo.** EDDM and Update performance were very close over time, with Update outperforming the drift detector.

(c) **EMBER.** Update and EDDM were the best methods, but the former was the best again.

Figure 5.9: **Drift Detection Results with Delay.** Results considering a delay in the stream, i.e., labels are available only at time $t + 19$ days. Prequential distribution of goodware and malware are also reported.

5.2.3 Label Probability

Considering that in the majority of the experiments using drift detectors the update strategy was the best, in the label probability experiment scenario we considered only this strategy and compared it with the static model. Theoretically, let $perf(p = k)$ be the function that measures the Prequential AUC-PR of a given model with label probability $p = k$, i.e., that the labels will be provided to the model in the stream with a probability k. Thus, our hypothesis is that it follows the inequation 5.1, i.e., that the static model ($p = 0\%$) will have the worst performance and the updated model ($p = 100\%$) will have the best performance, with any other probability ($p = k, 0\% < k < 100\%$) having its performance between both of them.

$$perf(p = 0) \leq perf(p = k) \leq perf(p = 1), \forall\ k \in (0, 1) \tag{5.1}$$

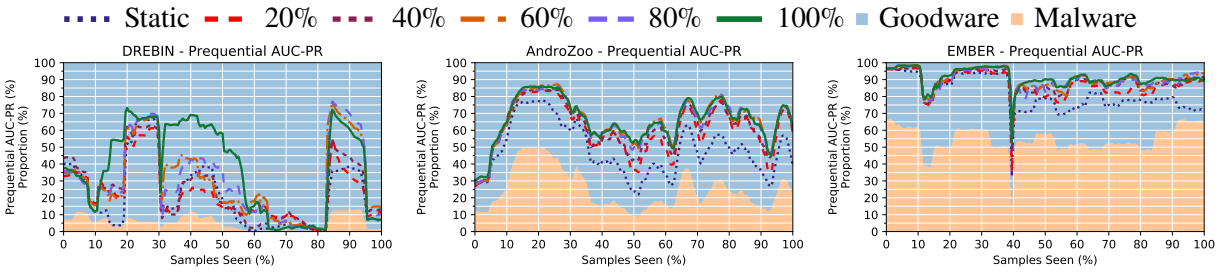
Besides the static ($p = 0\%$) and update ($p = 100\%$) strategies, we considered four values of label probabilities p: $p = 20\%$, $p = 40\%$, $p = 60\%$, and $p = 80\%$. The results over time considering no delays and delays are presented in Figures 5.10 and 5.11 and the overall results are presented in Table 5.8.

For the DREBIN dataset, our hypothesis seems to hold when not considering delays (Figure 5.10(a)) only in some intervals (for instance, from 10% to 30%), where the static model has the worst performance and the updated one, the best. However, only the update strategy ($p = 100\%$) keeps on top in the majority of time, while the static ($p = 0\$$) is still better than using $p = 20\%$, $p = 40\%$, and $p = 60\%$, as shown by the overall results. In contrast, when considering delays, the update strategy ($p = 100\%$) and using $p = 100\%$ are also outperformed by the static model ($p = 0\%$) as seen by the overall results. Surprisingly, in this scenario, when $p = 20\%$ we achieved better overall results than using more labels, so our hypothesis was not true in this scenario.

In the AndroZoo dataset, the results seem more aligned with our hypothesis when not considering delays (Figure 5.10(b)): over time, the static model has the worst result and the updated one, the best, with a significant difference (63.75% vs 79.23%). As shown by the overall results, the more labeled data we provide to the model, the better. When considering delays (Figure 5.11(b)), using only 20% of the labels ($p = 20\%$) was better than using more labels again, showing that our hypothesis was not true in this scenario too.

Finally, in the EMBER dataset, the results also seem aligned with our hypothesis when not considering delays (Figure 5.10(c)), the static model has the worst AUC-PR over time, with 86.18%, and the most updated one ($p = 100\%$), 92.02%, as we can see in the overall results as

well. Different from the other datasets, when we consider delays, we achieved the best overall results when using only 80% of the labels, even better than using 100% of them. Thus, our hypothesis was not true in this scenario again.

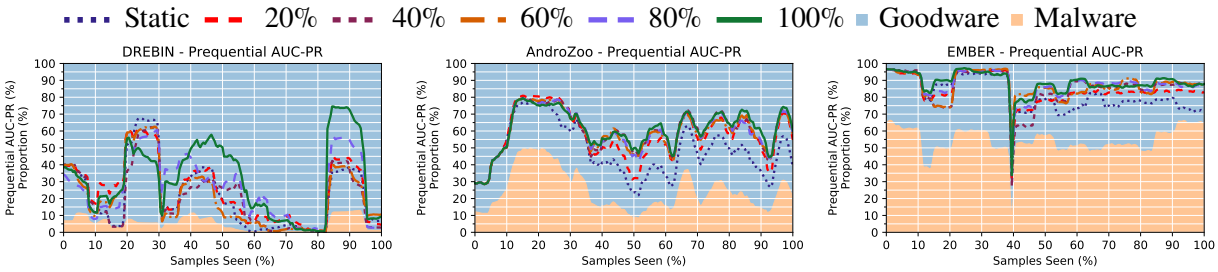


(a) **DREBIN.** Updating the model with all data (p = 100%) was the best method overall, but our hypothesis was not true.

(b) **AndroZoo.** Updating the model with all data (p = 100%) was the best method and our hypothesis was true.

(c) **EMBER.** Using all data to update the model (p = 100%) was the best strategy and our hypothesis was also true.

Figure 5.10: **Label Probability Results with no Delay.** Results considering no delay in the stream, i.e., labels are available at time $t + 1$. Prequential distribution of goodware and malware are also reported.



(a) **DREBIN.** Surprisingly, the static model (p = 0%) was the best overall strategy, i.e., not updating the model was better.

(b) **AndroZoo.** Using only 20% of the labels (p = 20%) was the best method overall.

(c) **EMBER.** Updating the model with 80% of the data (p = 80%) was the best method overall.

Figure 5.11: **Label Probability Results with Delay.** Results considering a delay in the stream, i.e., labels are available only at time $t + 19$ days. Prequential distribution of goodware and malware are also reported.

5.2.4 Lessons Learned

Whenever possible, use the AUC-PR metric prequentially to measure the classifier competence over time. Our implementation of AUC-PR improves the processing speed of the stream when compared to the standard scikit-learn implementation. At the end of the stream, compute the AUC-PR considering all tested instances or, if it is not possible, show the final AUC-PR as an average of each AUC-PR computed prequentially.

The updated models always had a significant drop in their performance over time when we consider delays, being closer to the static models. Thus, we advocate for more efforts to reduce the gap between data arrival and its effective labeling.

Drift detectors definitely work better than the random trigger detector, but they still have room for improvement. The reason behind that is that the most used drift detectors in literature such as DDM, EDDM, and ADWIN are looking only at error rates, i.e., accuracy, which is not the ideal metric for unbalanced data as much security data, such as malware detection. We advocate for more development on drift detectors that consider different metrics for unbalanced data.

Finally, we advocate for more strategies such as active learning (Pendlebury et al., 2019) to deal with scenarios where not all labels are available after a given delay k, querying the labels of just a few samples that are especially informative to build a better classifier (Krempl

Dataset	Probability	AUC-PR	
		No Delay	Delay
DREBIN	p = 0%	54.29%	**54.29%**
	p = 20%	47.77%	50.47%
	p = 40%	48.76%	39.35%
	p = 60%	54.41%	40.16%
	p = 80%	54.74%	44.71%
	p = 100%	**59.78%**	49.20%
AndroZoo	p = 0%	63.75%	63.75%
	p = 20%	76.12%	**71.21%**
	p = 40%	76.60%	70.62%
	p = 60%	78.07%	70.11%
	p = 80%	78.26%	70.76%
	p = 100%	**79.23%**	70.94%
EMBER	p = 0%	86.18%	86.18%
	p = 20%	89.78%	87.82%
	p = 40%	90.55%	88.37%
	p = 60%	91.12%	89.03%
	p = 80%	91.53%	**89.81%**
	p = 100%	**92.01%**	89.39%

Table 5.8: **Overall Label Probability Results**. Best results are highlighted in bold. Overall, when there are no delays, using all labels is the best option. Considering delays, using part of the labels may be the best option.

et al., 2014). In some scenarios, such as in AndroZoo, for instance, using only 20% of the data was better than using more. The hypothesis shown by the Inequation 5.1 was not true in any experiment that considered delays. Thus, selecting the right data to label would (i) reduce the efforts on data labeling, (ii) reduce processing time spent with model training, and (iii) improve classification performance over time by selecting the best samples to label for a given time t.

5.3 RELATED WORK

Machine Learning in Security. There are some methods that consider the non-stationary distribution of data in security applications and try to outline it, like DroidOL (Narayanan et al., 2016) and Casandra (Narayanan et al., 2017). (Yang et al., 2021) presented Cade, a system that aims to detect concept drift and explain why a sample is considered drifting. Their objective was to detect new unknown families based on the distance of a test instance to a centroid. If the new unlabeled instance is an outlier of all known classes, so it is considered a drifting sample.

Transcend (Jordaney et al., 2017) aims to assess the reliability of a model in a dataset by assessing the model's decision. They try to find a cutoff threshold guided by a Conformal Evaluator presented by the authors. The authors also point the importance of the class distribution of an algorithm under evaluation and included *alpha assessment*, which objective is to evaluate if algorithm results suffer from overfitting. Transcendent (Barbero et al., 2020) tries to outline some issues present in Transcend, such as experimental bias, and proposes two new conformal evaluators: Inductive Conformal Evaluator and Cross-conformal Evaluator. According to the authors, they are able to identify and reject drifting examples and require less computational power compared to the original (Jordaney et al., 2017).

DroidEvolver (Xu et al., 2019) creates an ensemble of various online learners and keeps updating them with pseudo-labels generated by the algorithm itself. However, this may cause some issues, like self-poisoning, as showed (Kan et al., 2021), who proposed DroidEvolver++, which tries to improve the idea of pseudo-labeling from DroidEvolver, and to outline *spatial bias*. (Shahraki et al., 2022) evaluate results of known online machine learning algorithms in the network domain and point challenges in modeling approaches to the data stream scenario, such as the *stability-plasticity dilemma* (Mermillod et al., 2013).

In (Andresini et al., 2021), authors extended Tesseract to the intrusion network domain and proposed Insomnia. It uses a Deep Neural Network as a classifier, and as new unlabeled instances arrive, batches are formed to update the model incrementally. It uses the same self-labeling idea from (Xu et al., 2019), and tries to avoid catastrophic forgetting, known for online deep learning approaches. Insomnia also counts with an explanation phase, which objective is to understand how the model has updated to drifting characteristics. (Dai et al., 2021) came with a Generative Adversarial Network approach to generate samples that simulate concept drift, which are used to train detection models.

Comparing to Tesseract and Dos and Don'ts. (Pendlebury et al., 2019) pointed some issues on approaching data streams in security applications, pointing ways to modeling approaches that match better with the real-world, focusing on the time decay of models due to the constant update and new techniques of attackers, which leads to concept drift. Authors define *temporal bias* as the assumption that data distribution does not change over time and *spatial bias* as the change in the class distribution in the training or testing phase. We modeled the problem as streaming data and used some techniques to try to outline concept drift. Data distribution, however, in our tests depends on how it arrives in a timestamp, as we start from the assumption that we have no way how to know data distribution in the future. (Pendlebury et al., 2019) proposed Tesseract, an open-source framework that aims to surpass some constraints to avoid *temporal and spatial bias*, and computes a new metric proposed by them, the *Area Under Time*.

(Arp et al., 2022) also explored issues when modeling approaches of machine learning in the security domain with streaming data, presenting pitfalls that might lead to biased results in this scenario, considering points such as sampling, labeling, evaluation, and the threat model. Authors defended delaying labeling to get reliable labels, as wrong labeling can affect the performance of a model. In this paper, we adopted a test-then-train approach with delayed instances by 19 days, different from (Pendlebury et al., 2019), who considered batches of 1 month of data. We believe that our approach is better and more realistic because (i) our delay k is always the same for all instances, for instance. When you consider 1 month of delay, the last sample of the previous month will have a delay of 1 day, and (ii) our framework reacts to changes faster, since we can update our model as soon as new labels are available, by instance, and not by batches of months.

Both works (Pendlebury et al., 2019; Arp et al., 2022) cite data snooping, which is when we use data "from the future" to train our model, which does not match the reality. We also use this assumption, as we trained our models on older instances, simulating the arrival of new ones in the testing phase.

5.4 CONCLUSION

We have modeled security data as streams to overcome current classification issues. We discussed challenges faced by online approaches and how they differ from offline ones, and apply our proposal to popular Android and MS Windows malware datasets, critically analyzing the results either to validate the approach we took or to understand its extent and limitations based on closely related state-of-the-art work. We identified problems with the most popular drift detectors in

the literature when applied to security data, given that they work solely based on accuracy. We showed that, in a realistic scenario, where labels are provided with delay and may never be available, model performance drops significantly. Furthermore, we implemented and publicly released the SPICE framework, an open-source library for data stream evaluation that includes modules to simulate these realistic scenarios as well as to efficiently compute evaluation metrics. As future work, we consider extending SPICE to include active learning strategies to reduce the efforts on data labeling and processing time spent with model training, and improve classification performance over time by selecting the best samples to label for a given time t. We also intend to propose and evaluate new drift detectors based on different metrics for unbalanced data, such as AUC-PR, to include in SPICE.

6 DISCUSSION

In this section we discuss some of the key aspects I found during the development of my research.

6.1 ADVERSARIAL ATTACKS ARE MORE COMMON IN PRACTICE THAN WE THINK

As noticed by the two papers I presented in Chapter 3, adversarial attacks against ML models are relatively easy to implement and yet very effective, which makes them more common in practice. This is not true only for malware detection, but we can see many other examples in object detection (Goodfellow et al., 2014b), phishing (Panum et al., 2020), and other areas. Thus, cybersecurity players should look at these cases with more attention when creating solutions.

6.2 DEFENDING AGAINST ATTACKS IS A CHALLENGING TASK

It is very difficult to implement defense solutions that are resilient to attacks, even when attackers use old and simple attacks. In some cases, using the adversarial samples to train the model may poison the model and increase the false positives, making its use infeasible (Ceschin et al., 2020a).

6.3 CYBERSECURITY RESEARCH SHOULD FOLLOW "MACHINE LEARNING THAT MATTERS"

Trying to improve the detection rate of the models is important, but trying to solve challenges that are closer to the real world, such as the ones mentioned in this work, is much more. For instance, increasing the accuracy of a model by 1 percentage point will not have the same impact in practice as creating strategies that are able to deal with adversarial attacks, concept drift, or any other challenge stated in this work.

6.4 CONCEPT DRIFT HAPPENS IN DIFFERENT LEVELS IN MANY PROBLEMS OF CYBERSECURITY

As seen in both Sections 4.1 and 4.2, even with two different security data, concept drift is shown, although they are seen on different levels. This may be explained by the difference in the time observed: while the continuous authentication dataset contains 8 weeks of data, the malware detection datasets have 9 years of data, i.e., the probability of having drifts in larger time windows is much bigger than in smaller ones. Thus, the fact of the smaller dataset contains changes indicates that it may happen in other scenarios, even in small periods.

6.5 THE WAY FEATURES ARE EXTRACTED IS VERY IMPORTANT IN NON-STATIONARY DISTRIBUTIONS

According to the feature extractor used, in non-stationary distributions such as any cybersecurity data, it is important to include a strategy to update it and not only the ML model. As shown in Section 4.2, feature extractors that are based on vocabularies, word embeddings, and/or static dictionaries need to be updated as time goes by due to the changes caused by concept drift. Thus, the representation built at a time T may not represent the input data at time $T + k$ and, as a

consequence, may result in sparser feature vectors. In malware detection, for instance, if we consider the libraries used by a binary, new ones may be created as time goes by, and older ones tend to be deprecated and not used, which would invalidate a feature vector trained with older libraries (Ceschin et al., 2022).

6.6 DELAYED EVALUATIONS SHOULD BE ADOPTED BY ANY CYBERSECURITY TASK THAT HAS DELAYS

In many cybersecurity tasks, it is necessary to label the data before actually using it in any ML pipeline. For instance, if we want to detect an attack, we need to verify in collected data if a given sample is actually an attack or not (usually domain-specific specialists are responsible for this task, i.e., it is considered an expensive task). Thus, for all evaluations that use data that need any labeling effort, it is important to consider the delay in the pipeline because, by default, it is not considered by standard ML practices. The SPICE framework presented in Section 5.1 considers the delays in any data stream evaluation, including the probability of not having some labels available, given that they may be inaccurate, unstable, or erroneous, which may affect the overall classification performance of ML-based solutions (Arp et al., 2022).

6.7 METRICS FOR IMBALANCED DATA SHOULD BE USED TO TRACK DRIFTS

The main drift detectors in literature such as DDM (Drift Detection Method (Gama et al., 2004)), EDDM (Early Drift Detection Method (Baena-Garcıa et al., 2006)), and ADWIN (ADaptive WINdowing (Bifet and Gavaldà, 2007)) are based on error rates or data distribution changes. None of them are based on any imbalanced data metric, such as recall, precision, fscore, or precision-recall curve. Thus, developing such drift detectors is important for the future of cybersecurity solutions, given that, in most cases, their data are imbalanced and the standard drift detectors give more importance to the majority class. The results presented in Section 5.1 show that using standard drift detectors in real scenarios that consider delays is worse than not using them, i.e., only updating the model with new data is better.

6.8 ACTIVE LEARNING HAS A PROMISING FUTURE IN CYBERSECURITY

Sometimes, using 100% of the labels to update a model is not good because they may be inaccurate, unstable, or erroneous and affect the overall classification performance of ML solutions (Arp et al., 2022), as already mentioned before. As an example, in malware detection (AndroZoo dataset), using only 20% of the labels to update the model is better than using more data. Thus, selecting the right samples to label, the ones that are especially informative (Krempl et al., 2014) may be a good strategy to handle these cases because it would (i) reduce the efforts on data labeling, (ii) reduce processing time spent with model training, and (iii) improve classification performance over time by selecting the best samples to label for a given time t, as we shown in Section 5.1.

6.9 IN THE END, IT ALL COMES DOWN TO THE ARMS RACE

Many works cited in this thesis mention the arms race created by attackers and defense solutions. In the end, every problem mentioned here is a consequence of this arms race, a cycle that will always exist where any action from one of the sides results in a reaction from the other one, i.e.,

attackers will always try to find a way to overcome the defense solutions, which will respond to them and try to create new strategies to defend.

7 CONCLUSION

In this thesis, I presented a collection of research works related to the challenges of applying ML to the cybersecurity domain, showing the main problems and proposing mitigations when possible. To do so, I first reviewed the literature to understand what are the challenges related to each step of ML in security, as shown in Section 2.1. Then, I explored some adversarial machine learning attacks for ML-based malware detectors, as seen in Sections 3.1 and 3.2. I also studied the effects of concept drift in two security domains: computer usage profiles (Section 4.1) and malware detection (Section 4.2). For computer usage profiles, binary models were the best options to authenticate users (for both online and offline experiments) and I found evidence of temporal consistency for most of the profiles within the study period. In the malware detection problem, I show that concept drift is a major concern and affects the performance of ML-based detectors as time goes by. Thus, I proposed a novel data stream cycle that includes not only the classifier but also the feature extractor in the process, which needs to be updated with time. Moreover, I developed a framework capable of evaluating any data stream considering delays in their labels, even when they may never be available with a given probability, as shown in Section 5.1. For future research, I recommend the following:

- **Think as an attacker to build defense solutions.** Attackers are always trying to find a way to evade defense solutions and build adversarial attacks to evade ML-based detectors. Thus, thinking as an attacker is a good way to build better defense solutions.
- **Commit yourself to the real world:** most of the challenges presented in this thesis are open research problems that are connected to real-world solutions. Thus, it is important to keep the motto "machine learning that matters" (Wagstaff, 2012) in mind, given that it is better to solve the challenges listed here than to improve 1% of accuracy in any detection model.
- **Drift detectors may be explored in different scenarios.** They may be used not only to update the model, but also feature extractors, word embeddings, and any other tool that helps the learning process. Also, new metrics should be considered for imbalanced data, so they can detect drifts with more accuracy in these scenarios.
- **Delays are important for evaluations.** In many cybersecurity contexts, it is essential to consider delays in the evaluation process, once it simulates a real-world scenario. Any evaluation that does not consider it, may be invalidated.
- **Active learning is the future of cybersecurity.** In some scenarios, using all the labels is not ideal, given that they may contain noises that confuse the model. Thus, active learning strategies that select the right samples to label are very important in such cases.

Finally, the main contributions of this thesis were (i) an extensive analysis of the literature regarding ML applied to cybersecurity in a comparative way; (ii) directions for machine learning applied to cybersecurity research considering its particularities, how to improve quality, and allow their effective use in real-world applications; and (iii) a set of modules or frameworks to improve further ML solutions for cybersecurity that can be used by both industry and academy.

REFERENCES

Abri, F., Siami-Namini, S., Khanghah, M. A., Soltani, F. M., and Namin, A. S. (2019). Can machine/deep learning classifiers detect zero-day malware with high accuracy? In *2019 IEEE International Conference on Big Data (Big Data)*, pages 3252–3259.

Aghakhani, H., Gritti, F., Mecca, F., Lindorfer, M., Ortolani, S., Balzarotti, D., Vigna, G., and Kruegel, C. (2020). When malware is packin'heat; limits of machine learning classifiers based on static analysis features. In *NDSS Proceedings*, NDSS, page 1, US. NDSS.

Ahmadi, Z. and Kramer, S. (2017). Modeling recurring concepts in data streams: a graph-based framework. *Knowledge and Information Systems*, 55:15–44.

Ahmed, A. A. and Traore, I. (2014). Biometric recognition based on free-text keystroke dynamics. *IEEE Transactions on Cybernetics*, 44(4):458–472.

Allix, K., Bissyandé, T. F., Klein, J., and Le Traon, Y. (2016a). Androzoo: Collecting millions of android apps for the research community. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 468–471, New York, NY, USA. ACM.

Allix, K., Bissyandé, T. F., Klein, J., and Le Traon, Y. (2016b). Androzoo: Collecting millions of android apps for the research community. In *Int. Conf. on Min. Soft. Rep.*

Anderson, H. (2019). Machine learning static evasion competition. `https://www.elastic.co/pt/blog/machine-learning-static-evasion-competition/`.

Anderson, H. S., Kharkar, A., Filar, B., Evans, D., and Roth, P. (2018). Learning to evade static pe machine learning malware models via reinforcement learning.

Anderson, H. S. and Roth, P. (2018). EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models. *ArXiv e-prints*.

Andresini, G., Pendlebury, F., Pierazzi, F., Loglisci, C., Appice, A., and Cavallaro, L. (2021). Insomnia: Towards concept-drift robustness in network intrusion detection. In *Proceedings of the 14th ACM Workshop on Artificial Intelligence and Security (AISec)*, pages 111–122.

Android (2016). Android 7.0 behavior changes. `https://tinyurl.com/yxlpc4gb`.

Android (2018a). Accountmanager. `https://tinyurl.com/ybsdz76e`.

Android (2018b). Device admin deprecation. `https://tinyurl.com/yygfkc3m`.

Android (2018c). Smsmanager. `https://tinyurl.com/y3sz4zzw`.

Arnaldo, I. and Veeramachaneni, K. (2019). The holy grail of "systems for machine learning": Teaming humans and machine learning for detecting cyber threats. *SIGKDD Explor. Newsl.*, 21(2):39–47.

Arp, D., Quiring, E., Pendlebury, F., Warnecke, A., Pierazzi, F., Wressnegger, C., Cavallaro, L., and Rieck, K. (2022). Dos and don'ts of machine learning in computer security. In *Proc. of the USENIX Security Symposium*.

Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., and Rieck, K. (2014). Drebin: Effective and explainable detection of android malware in your pocket. In *NDSS*.

Athalye, A., Carlini, N., and Wagner, D. A. (2018). Obfuscated gradients give a false sense of security: Circumventing defenses to adversarial examples. *CoRR*, abs/1802.00420.

Ayotte, B., Banavar, M. K., Hou, D., and Schuckers, S. (2021). Study of intra-and inter-user variance in password keystroke dynamics. In *Proceedings of the 7th International Conference on Information Systems Security and Privacy (ICISSP)*, pages 467–474.

Bach, S. H. and Maloof, M. A. (2008). Paired learners for concept drift. In *IEEE Int. Conf. on Data Mining*.

Baena-Garcıa, M., del Campo-Ávila, J., Fidalgo, R., Bifet, A., Gavalda, R., and Morales-Bueno, R. (2006). Early drift detection method. In *Fourth international workshop on knowledge discovery from data streams*, volume 6, pages 77–86.

Bahri, M., Bifet, A., Maniu, S., and Gomes, H. M. (2020). Survey on feature transformation techniques for data streams. In Bessiere, C., editor, *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI-20*, pages 4796–4802. International Joint Conferences on Artificial Intelligence Organization. Survey track.

Balazs, Z. (2020). Cujo ai partners with microsoft for the machine learning security evasion competition 2020. `https://cujo.com/machine-learning-security-evasion-competition-2020/`.

Ballenthin, W. and Raabe, M. (2020). capa: Automatically identify malware capabilities. `https://www.mandiant.com/resources/blog/capa-automatically-identify-malware-capabilities`.

Banon, S. (2020). Introducing elastic endpoint security. `https://www.elastic.co/blog/introducing-elastic-endpoint-security`.

Barbero, F., Pendlebury, F., Pierazzi, F., and Cavallaro, L. (2020). Transcending transcend: Revisiting malware classification with conformal evaluation. *CoRR*, abs/2010.03856.

Barddal, J. P., Gomes, H. M., Enembreck, F., and Pfahringer, B. (2017). A survey on feature drift adaptation: Definition, benchmark, challenges and future directions. *Journal of Systems and Software*, 127:278–294.

Beppler, T., Botacin, M., Ceschin, F., Oliveira, L. E. S., and Grégio, A. (2019). L(a)ying in (test)bed: How biased datasets produce impractical results for actual malware families' classification. In Lin, Z., Papamanthou, C., and Polychronakis, M., editors, *Information Security*, pages 381–401, Cham. Springer International Publishing.

Biallas, S. and Weyergraf, S. (2015). Ht editor. `http://hte.sourceforge.net/index.html`.

Bifet, A. and Gavaldà, R. (2009). Adaptive learning from evolving data streams. In Adams, N. M., Robardet, C., Siebes, A., and Boulicaut, J.-F., editors, *Advances in Intelligent Data Analysis VIII*, pages 249–260, Berlin, Heidelberg. Springer Berlin Heidelberg.

Bifet, A., Gavaldà, R., Holmes, G., and Pfahringer, B. (2018). *Machine Learning for Data Streams with Practical Examples in MOA*. MIT Press.

Bifet, A. and Gavaldà, R. (2007). Learning from time-changing data with adaptive windowing. In *SIAM Int. Conf. on Data Mining*.

Bifet, A., Holmes, G., Kirkby, R., and Pfahringer, B. (2010). Moa: Massive online analysis. *J. Mach. Learn. Res.*, 11:1601–1604.

Bishop, C. M. (2006). *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, Berlin, Heidelberg.

Bisson, D. (2019). Razy trojan installs malicious browser extensions to steal cryptocurrency. `https://securityintelligence.com/news/razy-trojan-installs-malicious-browser-extensions-to-steal-cryptocurrency/`.

BlackBerry - Cylance (2020). Hard on viruses, light on your computer. `https://shop.cylance.com/us/`.

Borello, J.-M. and Mé, L. (2008). Code obfuscation techniques for metamorphic viruses. *JICVHT*.

Botacin, M., Bertão, G., de Geus, P., Grégio, A., Kruegel, C., and Vigna, G. (2020a). On the security of application installer & online software repositories. DIMVA. Springer.

Botacin, M., Ceschin, F., de Geus, P., and Grégio, A. (2020b). We need to talk about antiviruses: Challenges & pitfalls of av evaluations. *Computers & Security*, page 101859.

Botacin, M., Galante, L., Ceschin, F., Santos, P. C., Carro, L., de Geus, P., Grégio, A., and Alves, M. A. Z. (2019). The av says: Your hardware definitions were updated! In *2019 14th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, pages 27–34.

Botacin, M., Kalysch, A., and Grégio, A. (2019). The internet banking [in]security spiral: Past, present, and future of online banking protection mechanisms based on a brazilian case study. In *Proceedings of the 14th International Conference on Availability, Reliability and Security*, ARES '19, pages 49:1–49:10, Canterbury, CA, United Kingdom. ACM.

Botacin, M. F., de Geus, P. L., and Grégio, A. R. A. (2018). The other guys: automated analysis of marginalized malware. *Journal of Computer Virology and Hacking Techniques*, 14(1):87–98.

Box, G. E., Jenkins, G. M., Reinsel, G. C., and Ljung, G. M. (2015). *Time series analysis: forecasting and control*. John Wiley & Sons.

Breiman, L. (1996). Bagging predictors. *Mach. Learn.*, 24(2):123–140.

Breiman, L. (2001). Random forests. *Machine learning*, 45(1):5–32.

Brodersen, K. H., Ong, C. S., Stephan, K. E., and Buhmann, J. M. (2010). The balanced accuracy and its posterior distribution. In *2010 20th international conference on pattern recognition*, pages 3121–3124. IEEE.

Brown, C. D. and Davis, H. T. (2006). Receiver operating characteristics curves and related decision measures: A tutorial. *Chemometrics and Intelligent Laboratory Systems*, 80(1):24–38.

Brzezinski, D. and Stefanowski, J. (2017). Prequential auc: properties of the area under the roc curve for data streams with concept drift. *Knowledge and Information Systems*, 52(2):531–562.

Bursztein, E. and Oliveira, D. (2019). Deconstructing the phishing campaigns that target gmail users. *Black Hat USA 2019*. `https://elie.net/talk/deconstructing-the-phishing-campaigns-that-target-gmail-users/`.

Cai, H. (2018). A preliminary study on the sustainability of android malware detection.

Cai, H. (2020). Assessing and improving malware detection sustainability through app evolution studies. *ACM Trans. Softw. Eng. Methodol.*, 29(2).

Cai, H. and Jenkins, J. (2018). Towards sustainable android malware detection. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ICSE '18, page 350–351, New York, NY, USA. Association for Computing Machinery.

Calleja, A., Martín, A., Menéndez, H. D., Tapiador, J., and Clark, D. (2018). Picking on the family: Disrupting android malware triage by forcing misclassification. *Expert Systems with Applications*, 95:113 – 126.

Carlini, N. and Wagner, D. (2017). Towards evaluating the robustness of neural networks. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 39–57.

Cavallaro, L. (2019). When the magic wears off: Flaws in ML for security evaluations (and what to do about it). Burlingame, CA. USENIX Association.

Ceschin, F., Botacin, M., Gomes, H. M., Oliveira, L. S., and Grégio, A. (2019). Shallow security: On the creation of adversarial variants to evade machine learning-based malware detectors. In *Proceedings of the 3rd Reversing and Offensive-Oriented Trends Symposium*, ROOTS'19, Vienna, Austria. Association for Computing Machinery.

Ceschin, F., Botacin, M., Gomes, H. M., Pinagé, F., Oliveira, L. S., and Grégio, A. (2022). Fast & furious: On the modelling of malware detection as an evolving data stream. *Expert Systems with Applications*, page 118590.

Ceschin, F., Botacin, M., Lüders, G., Gomes, H. M., Oliveira, L., and Gregio, A. (2020a). No need to teach new tricks to old malware: Winning an evasion challenge with xor-based adversarial samples. In *Reversing and Offensive-Oriented Trends Symposium*, ROOTS'20, page 13–22, Vienna, Austria. Association for Computing Machinery.

Ceschin, F., Gomes, H. M., Botacin, M., Bifet, A., Pfahringer, B., Oliveira, L. S., and Grégio, A. (2020b). Machine learning (in) security: A stream of problems.

Ceschin, F., Pinage, F., Castilho, M., Menotti, D., Oliveira, L. S., and Gregio, A. (2018). The need for speed: An analysis of brazilian malware classifers. *IEEE Security Privacy*, 16(6):31–41.

Chang, J., Venkatasubramanian, K. K., West, A. G., and Lee, I. (2013). Analyzing and defending against web-based malware. *ACM Comput. Surv.*, 45(4).

Chawla, N., Bowyer, K., Hall, L., and Kegelmeyer, W. (2002). Smote: Synthetic minority over-sampling technique. *J. Artif. Intell. Res. (JAIR)*, 16:321–357.

Chen, L. (2018). Deep transfer learning for static malware classification. *CoRR*, abs/1812.07606.

Chen, L., Sahita, R., Parikh, J., and Marino, M. (2020). STAMINA Deep Learning for Malware Protection.

Cheron, A. (2017). Code injection with python. https://axcheron.github.io/code-injection-with-python/.

Christodorescu, M., Jha, S., Seshia, S. A., Song, D., and Bryant, R. E. (2005). Semantics-aware malware detection. In *2005 Symposium on Security and Privacy (S&P'05)¨*, pages 32–46.

Chuang, J., Nguyen, H., Wang, C., and Johnson, B. (2013). I think, therefore i am: Usability and security of authentication using brainwaves. In *International conference on financial cryptography and data security*, pages 1–16. Springer.

Cimpanu, C. (2018). Google restricts which android apps can request call log and sms permissions. https://tinyurl.com/y3y5gfhx.

Coleman, C., Kang, D., Narayanan, D., Nardi, L., Zhao, T., Zhang, J., Bailis, P., Olukotun, K., Ré, C., and Zaharia, M. (2019). Analysis of dawnbench, a time-to-accuracy machine learning performance benchmark. *SIGOPS Oper. Syst. Rev.*, 53(1):14–25.

Costa, M., Priplata, A., Lipsitz, L., Wu, Z., Huang, N., Goldberger, A. L., and Peng, C.-K. (2007). Noise and poise: enhancement of postural complexity in the elderly with a stochastic-resonance–based therapy. *Europhysics Letters*, 77(6):68008.

Crane, C. (2020). 42 cyber attack statistics by year: A look at the last decade. https://sectigostore.com/blog/42-cyber-attack-statistics-by-year-a-look-at-the-last-decade/.

CrowdStrike (2020). Falcon prevent: Cloud-native next-generation antivirus (ngav). https://www.crowdstrike.com/endpoint-security-products/falcon-prevent-endpoint-antivirus/.

CTONetworks (2017). Malware infections grow 4x in just one quarter. https://tinyurl.com/yyyugke7.

Cynet (2020). Next-gen antivirus. proactively block zero day attacks. https://www.cynet.com/platform/threat-protection/nextgen-anti-virus/.

D4Vinci (2017). Dr0p1t. https://github.com/D4Vinci/Dr0p1t-Framework.

Dai, Y., Li, H., Qian, Y., Guo, Y., and Zheng, M. (2021). Anticoncept drift method for malware detector based on generative adversarial network. *Security and Communication Networks*, 2021:1–12.

Dalal, N. and Triggs, B. (2005). Histograms of oriented gradients for human detection. In *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, volume 1, pages 886–893 vol. 1.

Dasgupta, D. (2018). Beware steep decline: Understanding model degradation in machine learning models. https://www.endgame.com/blog/technical-blog/beware-steep-decline-understanding-model-degradation-machine-learning-models.

Davis, J. and Goadrich, M. (2006). The relationship between precision-recall and roc curves. In *Proceedings of the 23rd International Conference on Machine Learning*, ICML '06, page 233–240, New York, NY, USA. Association for Computing Machinery.

Dehghan, M., Beigy, H., and Zaremoodi, P. (2016). A novel concept drift detection method in data streams using ensemble classifiers. *Intell. Data Anal.*, 20:1329–1350.

Deo, A., Dash, S. K., Suarez-Tangil, G., Vovk, V., and Cavallaro, L. (2016). Prescience: Probabilistic guidance on the retraining conundrum for malware detection. In *Proceedings of the ACM Workshop on Artificial Intelligence and Security*.

Dev, M., Gupta, H., Mehta, S., and Balamurugan, B. (2016). Cache implementation using collective intelligence on cloud based antivirus architecture. In *2016 International Conference on Advanced Communication Control and Computing Technologies (ICACCCT)*, pages 593–595.

Drolet, M. (2019). What is emotet? and how to guard against this persistent trojan malware. `https://www.csoonline.com/article/3387146/what-is-emotet-and-how-to-guard-against-this-persistent-trojan-malware.html`.

Dwork, C. and Roth, A. (2014). The algorithmic foundations of differential privacy. *Found. Trends Theor. Comput. Sci.*, 9(3–4):211–407.

El Hadj, M. A., Erradi, M., Khoumsi, A., and Benkaouz, Y. (2018). Validation and correction of large security policies: A clustering and access log based approach. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 5330–5332.

Emami-Naeini, P., Francisco, T., Kohno, T., and Roesner, F. (2021). Understanding privacy attitudes and concerns towards remote communications during the covid-19 pandemic. In *Seventeenth Symposium on Usable Privacy and Security (SOUPS)*, pages 695–714.

EndGame (2019). Machine learning static evasion competition. `https://www.endgame.com/blog/technical-blog/machine-learning-static-evasion-competition`.

Epley, N. and Gilovich, T. (2006). The anchoring-and-adjustment heuristic: Why the adjustments are insufficient. *Psychological Science*, 17(4):311–318. PMID: 16623688.

Erocarrera (2019). Pefile is a python module to read and work with pe (portable executable) files. `https://github.com/erocarrera/pefile`.

FBReader (2018). for old android devices. `https://tinyurl.com/y2odlk9w`.

Fergus Halliday (2018). What are the most common kinds of Android malware? `https://tinyurl.com/wcmr9m2`.

Ferreira, D., Kostakos, V., Beresford, A. R., Lindqvist, J., and Dey, A. K. (2015). Securacy: an empirical investigation of android applications' network usage, privacy and security. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, pages 1–11.

Ferri, C., Hernández-Orallo, J., and Modroiu, R. (2009). An experimental comparison of performance measures for classification. *Pattern Recognition Letters*, 30(1):27 – 38.

Ferrie, P. (2008). Anti-unpacker tricks. http://pferrie.tripod.com/papers/unpackers.pdf.

Ficco, M. (2022). Malware analysis by combining multiple detectors and observation windows. *IEEE Trans. Comput.*, 71(6):1276–1290.

Filar, B. (2020). Malware bypass research using reinforcement learning. https://github.com/bfilar/malware_rl.

FindLaw (2016). Email privacy concerns.

FireEye (2014). Xtremerat: Nuisance or threat? https://www.fireeye.com/blog/threat-research/2014/02/xtremerat-nuisance-or-threat.html.

FireEye (2019). Stringsifter. https://github.com/fireeye/stringsifter.

Fleshman, W., Raff, E., Sylvester, J., Forsyth, S., and McLean, M. (2018). Non-negative networks against adversarial attacks. https://arxiv.org/abs/1806.06108.

Freund, Y. and Schapire, R. E. (1997). A decision-theoretic generalization of on-line learning and an application to boosting. *J. Comput. Syst. Sci.*, 55(1):119–139.

Fridman, L., Weber, S., Greenstadt, R., and Kam, M. (2017). Active authentication on mobile devices via stylometry, application usage, web browsing, and gps location. *IEEE Systems Journal*, 11(2):513–521.

Fu, X. and Cai, H. (2019). On the deterioration of learning-based malware detectors for android. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 272–273.

Gade, K., Geyik, S., Kenthapadi, K., Mithal, V., and Taly, A. (2020). Explainable ai in industry: Practical challenges and lessons learned. In *Companion Proceedings of the Web Conference 2020*, WWW '20, page 303–304, New York, NY, USA. Association for Computing Machinery.

Galante, L., Botacin, M., Grégio, A., and de Geus, P. (2019). Machine learning for malware detection: Beyond accuracy rates. Brazilian Security Symposium (SBSeg).

Galar, M., Fernandez, A., Barrenechea, E., Bustince, H., and Herrera, F. (2012). A review on ensembles for the class imbalance problem: Bagging-, boosting-, and hybrid-based approaches. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 42(4):463–484.

Gama, J., Medas, P., Castillo, G., and Rodrigues, P. (2004). Learning with drift detection. In Bazzan, A. L. C. and Labidi, S., editors, *Advances in Artificial Intelligence – SBIA 2004*, pages 286–295, Berlin, Heidelberg. Springer Berlin Heidelberg.

Gama, J., Žliobaitė, I., Bifet, A., Pechenizkiy, M., and Bouchachia, A. (2014a). A survey on concept drift adaptation. *ACM computing surveys (CSUR)*, 46(4):1–37.

Gama, J. a., Žliobaitė, I., Bifet, A., Pechenizkiy, M., and Bouchachia, A. (2014b). A survey on concept drift adaptation. *ACM Comput. Surv.*, 46(4):44:1–44:37.

Gandotra, E., Bansal, D., and Sofat, S. (2014). Malware analysis and classification: A survey. *Journal of Information Security*, 5(2):56–64.

Geiger, R. S., Yu, K., Yang, Y., Dai, M., Qiu, J., Tang, R., and Huang, J. (2020). Garbage in, garbage out? do machine learning application papers in social computing report where human-labeled training data comes from? In *Proceedings of the 2020 Conference on Fairness, Accountability, and Transparency*, FAT* '20, page 325–336, New York, NY, USA. Association for Computing Machinery.

Gibert, D., Mateu, C., and Planes, J. (2020). The rise of machine learning for detection and classification of malware: Research developments, trends and challenges. *Journal of Network and Computer Applications*, 153:102526.

Giovanini, L., Ceschin, F., Silva, M., Chen, A., Kulkarni, R., Banda, S., Lysaght, M., Qiao, H., Sapountzis, N., Sun, R., Matthews, B., Wu, D. O., Grégio, A., and Oliveira, D. (2021). Online binary models are promising for distinguishing temporally consistent computer usage profiles.

Giovanini, L., Ceschin, F., Silva, M., Chen, A., Kulkarni, R., Banda, S., Lysaght, M., Qiao, H., Sapountzis, N., Sun, R., Matthews, B., Wu, D. O., Grégio, A., and Oliveira, D. (2022). Online binary models are promising for distinguishing temporally consistent computer usage profiles. *IEEE Transactions on Biometrics, Behavior, and Identity Science*, pages 1–1.

Goh, J. (2018). Machine learning: The ideal ally for security analysts. `https://www.paloaltonetworks.com/blog/security-operations/machine-learning-the-ideal-ally-for-security-analysts/`.

Gomes, H. M., Barddal, J. P., Enembreck, F., and Bifet, A. (2017a). A survey on ensemble learning for data stream classification. *ACM Computing Surveys (CSUR)*, 50(2):1–36.

Gomes, H. M., Bifet, A., Read, J., Barddal, J. P., Enembreck, F., Pfahringer, B., Holmes, G., and Abdessalem, T. (2017b). Adaptive random forests for evolving data stream classification. *Machine Learning*, pages 1–27.

Gomes, H. M., Bifet, A., Read, J., Barddal, J. P., Enembreck, F., Pfharinger, B., Holmes, G., and Abdessalem, T. (2017c). Adaptive random forests for evolving data stream classification. *Mach. Learn.*, 106(9–10):1469–1495.

Gomes, H. M., Read, J., Bifet, A., Barddal, J. P., and Gama, J. a. (2019). Machine learning for streaming data: State of the art, challenges, and opportunities. *SIGKDD Explor. Newsl.*, 21(2):6–22.

Goodfellow, I. J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. (2014a). Generative adversarial networks.

Goodfellow, I. J., Shlens, J., and Szegedy, C. (2014b). Explaining and harnessing adversarial examples.

Gritzalis, D., Iseppi, G., Mylonas, A., and Stavrou, V. (2018). Exiting the risk assessment maze: A meta-survey. *ACM Comput. Surv.*, 51(1).

Gron, A. (2017). *Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly Media, Inc., 1st edition.

Grosse, K., Papernot, N., Manoharan, P., Backes, M., and McDaniel, P. D. (2017). Adversarial examples for malware detection. In *ESORICS*.

Guo, C., Gardner, J. R., You, Y., Wilson, A. G., and Weinberger, K. Q. (2019). Simple black-box adversarial attacks. *CoRR*, abs/1905.07121:1.

Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., and Witten, I. H. (2009). The weka data mining software: An update. *SIGKDD Explor. Newsl.*, 11(1):10–18.

Hamandi, K., Elhajj, I. H., Chehab, A., and Kayssi, A. (2012). Android sms botnet: A new perspective. In *Int. Symp. on Mobility Management and Wireless Access*.

Han, J., Pei, J., and Kamber, M. (2011). *Data mining: concepts and techniques*. Elsevier.

Hick, P., Aben, E., Claffy, K., and Polterock, J. (2007). the caida ddos attack 2007 dataset. `https://www.caida.org/data/passive/ddos-20070804_dataset.xml`.

Hosseini, M. J., Gholipour, A., and Beigy, H. (2015). An ensemble of cluster-based classifiers for semi-supervised classification of non-stationary data streams. *Knowledge and Information Systems*, 46.

Howard, J. and Ruder, S. (2018). Fine-tuned language models for text classification. *CoRR*, abs/1801.06146.

Hu, W. and Tan, Y. (2017). Generating adversarial malware examples for black-box attacks based on GAN. *CoRR*, abs/1702.05983.

Huang, J., Hou, D., and Schuckers, S. (2017). A practical evaluation of free-text keystroke dynamics. In *2017 IEEE International Conference on Identity, Security and Behavior Analysis (ISBA)*, pages 1–8.

Huang, K., Siegel, M., and Madnick, S. (2018). Systematically understanding the cyber attack business: A survey. *ACM Comput. Surv.*, 51(4).

Hulten, G., Spencer, L., and Domingos, P. (2001). Mining time-changing data streams. In *ACM SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining*, pages 97–106. ACM Press.

Hurier, M., Suarez-Tangil, G., Dash, S. K., Bissyande, T. F., Le Traon, Y., Klein, J., and Cavallaro, L. (2017). Euphony: Harmonious Unification of Cacophonous Anti-Virus Vendor Labels for Android Malware. In *IEEE International Working Conference on Mining Software Repositories*, pages 425–435. IEEE Computer Society.

Hussain, M., Bird, J. J., and Faria, D. R. (2019). A study on cnn transfer learning for image classification. In Lotfi, A., Bouchachia, H., Gegov, A., Langensiepen, C., and McGinnity, M., editors, *Advances in Computational Intelligence Systems*, pages 191–202, Cham. Springer International Publishing.

Inc, T. (2020). Tanium.

Jarabek, C., Barrera, D., and Aycock, J. (2012). Thinav: Truly lightweight mobile cloud-based anti-malware. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, page 209–218, New York, NY, USA. Association for Computing Machinery.

Jiang, H., Nagra, J., and Ahammad, P. (2016). Sok: Applying machine learning in security - A survey. *CoRR*, abs/1611.03186.

Jones, K. S. (1972). A statistical interpretation of term specificity and its application in retrieval. *Journal of Documentation*, 28:11–21.

Jordaney, R., Sharad, K., Dash, S. K., Wang, Z., Papini, D., Nouretdinov, I., and Cavallaro, L. (2017). Transcend: Detecting concept drift in malware classification models. In *USENIX Security Symposium*.

Kan, Z., Pendlebury, F., Pierazzi, F., and Cavallaro, L. (2021). Investigating labelless drift adaptation for malware detection. In *Proceedings of the 14th ACM Workshop on Artificial Intelligence and Security*, AISec '21, page 123–134, New York, NY, USA. Association for Computing Machinery.

Kang, M. G., Poosankam, P., and Yin, H. (2007). Renovo: A hidden code extractor for packed executables. In *Proceedings of the 2007 ACM Workshop on Recurring Malcode*, WORM '07, pages 46–53, New York, NY, USA. ACM.

Kang, P. and Cho, S. (2015). Keystroke dynamics-based user authentication using long and free text strings from various input devices. *Information Sciences*, 308:72 – 93.

Kantchelian, A., Afroz, S., Huang, L., Islam, A. C., Miller, B., Tschantz, M. C., Greenstadt, R., Joseph, A. D., and Tygar, J. D. (2013). Approaches to adversarial drift. In *Proceedings of the 2013 ACM Workshop on Artificial Intelligence and Security*, AISec '13, page 99–110, New York, NY, USA. Association for Computing Machinery.

Kantchelian, A., Tschantz, M. C., Afroz, S., Miller, B., Shankar, V., Bachwani, R., Joseph, A. D., and Tygar, J. D. (2015). Better malware ground truth: Techniques for weighting anti-virus vendor labels. In *AISec 2015 - Proceedings of the 8th ACM Workshop on Artificial Intelligence and Security, co-located with CCS 2015*, pages 45–56, New York, New York, USA. Association for Computing Machinery, Inc.

Kaufman, S., Rosset, S., and Perlich, C. (2011). Leakage in data mining: Formulation, detection, and avoidance. volume 6, pages 556–563.

Kaur, H., Pannu, H. S., and Malhi, A. K. (2019). A systematic review on imbalanced data challenges in machine learning: Applications and solutions. *ACM Computing Surveys (CSUR)*, 52(4):1–36.

Keser, R. K. and Töreyin, B. U. (2019). Autoencoder based dimensionality reduction of feature vectors for object recognition. In *2019 15th International Conference on Signal-Image Technology Internet-Based Systems (SITIS)*, pages 577–584.

Kim, D., Dunphy, P., Briggs, P., Hook, J., Nicholson, J. W., Nicholson, J., and Olivier, P. (2010). Multi-touch authentication on tabletops. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pages 1093–1102, New York, NY, USA. ACM.

Koeze, E. and Popper, N. (2020). The virus changed the way we internet.

Korzybski, A. (1931). *A Non-Aristotelian System and Its Necessity for Rigour in Mathematics and Physics: Abstract*.

Krempl, G., Žliobaite, I., Brzeziński, D., Hüllermeier, E., Last, M., Lemaire, V., Noack, T., Shaker, A., Sievi, S., Spiliopoulou, M., and Stefanowski, J. (2014). Open challenges for data stream mining research. *SIGKDD Explor. Newsl.*, 16(1):1–10.

Krizhevsky, A. (2012). Learning multiple layers of features from tiny images. *University of Toronto*.

Kulesza, T., Amershi, S., Caruana, R., Fisher, D., and Charles, D. (2014). Structured labeling to facilitate concept evolution in machine learning.

Kumar, M. and Mathur, R. (2014). Unsupervised outlier detection technique for intrusion detection in cloud computing. In *International Conference for Convergence for Technology-2014*, pages 1–4.

Kumar, S. and Bhim Bhan Singh, C. (2018). A zero-day resistant malware detection method for securing cloud using svm and sandboxing techniques. In *2018 Second International Conference on Inventive Communication and Computational Technologies (ICICCT)*, pages 1397–1402.

Kwon, B. J., Mondal, J., Jang, J., Bilge, L., and Dumitraş, T. (2015). The dropper effect: Insights into malware distribution with downloader graph analytics. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 1118–1129, New York, NY, USA. ACM.

Lazarevic, A., Kumar, V., and Srivastava, J. (2005). Intrusion detection: A survey. In *Managing Cyber Threats*, pages 19–78. Springer.

Lemaire, V., Salperwyck, C., and Bondu, A. (2015). *A Survey on Supervised Classification on Data Streams*, pages 88–125. Springer International Publishing, Cham.

Levin, B., Gorzelany, A. M., Simpson, D., and Halfin, D. (2019). Malware names. `https://docs.microsoft.com/en-us/windows/security/threat-protection/intelligence/malware-naming`.

Li, L., Qi, X., Xie, T., and Li, B. (2020). Sok: Certified robustness for deep neural networks.

Liao, Y. and Vemuri, V. (2002). Use of k-nearest neighbor classifier for intrusion detection11an earlier version of this paper is to appear in the proceedings of the 11th usenix security symposium, san francisco, ca, august 2002. *Computers & Security*, 21(5):439 – 448.

LightGBM (2018). Lightgbm. `https://lightgbm.readthedocs.io/en/latest/`.

Lipton, Z. C., Elkan, C., and Naryanaswamy, B. (2014). Optimal thresholding of classifiers to maximize f1 measure. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 225–239. Springer.

Lisboa de Almeida, P. R., Oliveira, L. S., Souza Britto, A. d., and Paul Barddal, J. (2020). Naïve approaches to deal with concept drifts. In *2020 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 1052–1059.

Liu, F. T., Ting, K. M., and Zhou, Z.-H. (2012). Isolation-based anomaly detection. *ACM Trans. Knowl. Discov. Data*, 6(1).

López, A. U., Mateo, F., Navío-Marco, J., Martínez-Martínez, J. M., Gómez-Sanchís, J., Vila-Francés, J., and Serrano-López, A. J. (2019). Analysis of computer user behavior, security incidents and fraud using self-organizing maps. *Computers & Security*, 83:38–51.

Lu, J., Liu, A., Dong, F., Gu, F., Gama, J., and Zhang, G. (2020). Learning under concept drift: A review. *CoRR*, abs/2004.05785.

lucasg (2017). The sad state of pe parsing. `https://lucasg.github.io/2017/04/28/the-sad-state-of-pe-parsing/`.

Luo, W., Xu, S., and Jiang, X. (2013). Real-time detection and prevention of android sms permission abuses. In *Int. W. on Sec. in Embedded Sys. and Smartphones*.

Machiry, A., Redini, N., Gustafson, E., Fratantonio, Y., Choe, Y. R., Kruegel, C., and Vigna, G. (2018). Using loops for malware classification resilient to feature-unaware perturbations. In *Proceedings of the 34th Annual Computer Security Applications Conference*, ACSAC '18, page 112–123, New York, NY, USA. Association for Computing Machinery.

Mahbub, U., Komulainen, J., Ferreira, D., and Chellappa, R. (2019). Continuous Authentication of Smartphones Based on Application Usage. *IEEE Transactions on Biometrics, Behavior, and Identity Science*, 1(3):165–180.

Mahbub, U., Sarkar, S., Patel, V. M., and Chellappa, R. (2016). Active User Authentication for Smartphones: A Challenge Data Set and Benchmark Results. *IEEE International Conference on Biometrics Theory,Applications and Systems (BTAS)*.

Maiorca, D., Biggio, B., and Giacinto, G. (2019). Towards adversarial malware detection: Lessons learned from pdf-based attacks. *ACM Computing Surveys (CSUR)*, 52(4):1–36.

Malpedia (2019). Loki password stealer (pws). `https://malpedia.caad.fkie.fraunhofer.de/details/win.lokipws`.

Malshare (2019). A free malware repository providing researchers access to samples, malicous feeds, and yara results. `https://malshare.com/`.

Malwarebytes Labs (2017). Trojan.remcos. `https://blog.malwarebytes.com/detections/trojan-remcos/`.

Manning, C. D., Raghavan, P., and Schütze, H. (2008). *Introduction to Information Retrieval*. Cambridge University Press, USA.

Masud, M. M., Al-Khateeb, T. M., Hamlen, K. W., Gao, J., Khan, L., Han, J., and Thuraisingham, B. (2008). Cloud-based malware detection for evolving data streams. *ACM Trans. Manage. Inf. Syst.*

Matei Zaharia, Tathagata Das, M. A. and Xin, R. (2016). Spark structured streaming: A new high-level api for streaming.

Melis, M., Demontis, A., Pintor, M., Sotgiu, A., and Biggio, B. (2019). secml: A python library for secure and explainable machine learning. *arXiv preprint arXiv:1912.10013*.

Mermillod, M., Bugaiska, A., and BONIN, P. (2013). The stability-plasticity dilemma: investigating the continuum from catastrophic forgetting to age-limited learning effects. *Frontiers in Psychology*, 4.

Michie, D., Spiegelhalter, D. J., Taylor, C. C., and Campbell, J., editors (1994). *Machine Learning, Neural and Statistical Classification*. Ellis Horwood, Upper Saddle River, NJ, USA.

Microsoft (2017a). Event tracing for windows (etw).

Microsoft (2017b). Peering inside the pe: A tour of the win32 portable executable file format. http://bytepointer.com/resources/pietrek_peering_inside_pe.htm.

Microsoft (2018). Resource files (c++). https://docs.microsoft.com/en-us/cpp/windows/resource-files-visual-studio?view=vs-2019.

Microsoft Security Intelligence (2017). Win32/gamarue. https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=Win32/Gamarue.

Mikolov, T., Chen, K., Corrado, G., and Dean, J. (2013a). Efficient estimation of word representations in vector space. *CoRR*, abs/1301.3781.

Mikolov, T., Sutskever, I., Chen, K., Corrado, G., and Dean, J. (2013b). Distributed representations of words and phrases and their compositionality. *CoRR*.

MLSec (2019). Scores. https://evademalwareml.io/scores/.

Montiel, J., Halford, M., Mastelini, S. M., Bolmier, G., Sourty, R., Vaysse, R., Zouitine, A., Gomes, H. M., Read, J., Abdessalem, T., et al. (2021). River: machine learning for streaming data in python. *The Journal of Machine Learning Research*, 22(1):1–8.

Montiel, J., Read, J., Bifet, A., and Abdessalem, T. (2018). Scikit-multiflow: A multi-output streaming framework. *The Journal of Machine Learning Research*, 19(1):2915–2914.

Murphy, C., Huang, J., Hou, D., and Schuckers, S. (2017). Shared dataset on natural human-computer interaction to support continuous authentication research. In *2017 IEEE International Joint Conference on Biometrics (IJCB)*, pages 525–530.

Narayanan, A., Chandramohan, M., Chen, L., and Liu, Y. (2017). Context-aware, adaptive, and scalable android malware detection through online learning. *IEEE Transactions on Emerging Topics in Computational Intelligence*, 1(3):157–175.

Narayanan, A., Yang, L., Chen, L., and Jinliang, L. (2016). Adaptive and scalable android malware detection through online learning. In *IJCNN*.

NetResec (2020). Publicly available pcap files. https://www.netresec.com/?page=PcapFiles.

Nguyen, A., Zak, R., Richards, L. E., Fuchs, M., Lu, F., Brandon, R., Munoz, G. D. L., Raff, E., Nicholas, C., and Holt, J. (2022). Minimizing compute costs: When should we run more expensive malware analysis?

Oberhumer, M. F., Molnár, L., and Reiser, J. F. (2018). Upx the ultimate packer for executables. https://upx.github.io/.

Onwuzurike, L., Mariconti, E., Andriotis, P., Cristofaro, E. D., Ross, G., and Stringhini, G. (2019). Mamadroid: Detecting android malware by building markov chains of behavioral models (extended version). *ACM Trans. Priv. Secur.*, 22(2).

Palo Alto Networks (2020). Advanced endpoint protection protects you from dated antivirus. https://www.paloaltonetworks.com/cyberpedia/advanced-endpoint-protection-protects-you-from-dated-antivirus.

Pang, R., Allman, M., Bennett, M., Lee, J., Paxson, V., and Tierney, B. (2005). A first look at modern enterprise traffic. In *Proceedings of the 5th ACM SIGCOMM conference on Internet Measurement*, pages 2–2.

Panigrahi, P. K. (2012). A comparative study of supervised machine learning techniques for spam e-mail filtering. In *2012 Fourth International Conference on Computational Intelligence and Communication Networks*, pages 506–512.

Panum, T. K., Hageman, K., Hansen, R. R., and Pedersen, J. M. (2020). Towards adversarial phishing detection. In *13th USENIX Workshop on Cyber Security Experimentation and Test (CSET 20)*. USENIX Association.

Papernot, N., Faghri, F., Carlini, N., Goodfellow, I., Feinman, R., Kurakin, A., Xie, C., Sharma, Y., Brown, T., Roy, A., Matyasko, A., Behzadan, V., Hambardzumyan, K., Zhang, Z., Juang, Y.-L., Li, Z., Sheatsley, R., Garg, A., Uesato, J., Gierke, W., Dong, Y., Berthelot, D., Hendricks, P., Rauber, J., and Long, R. (2018a). Technical report on the cleverhans v2.1.0 adversarial examples library. *arXiv preprint arXiv:1610.00768*.

Papernot, N., McDaniel, P., Sinha, A., and Wellman, M. P. (2018b). Sok: Security and privacy in machine learning. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 399–414.

Papernot, N., McDaniel, P. D., Goodfellow, I. J., Jha, S., Celik, Z. B., and Swami, A. (2016). Practical black-box attacks against deep learning systems using adversarial examples. *CoRR*, abs/1602.02697.

Papernot, N., McDaniel, P. D., Jha, S., Fredrikson, M., Celik, Z. B., and Swami, A. (2015a). The limitations of deep learning in adversarial settings. *CoRR*, abs/1511.07528.

Papernot, N., McDaniel, P. D., Wu, X., Jha, S., and Swami, A. (2015b). Distillation as a defense to adversarial perturbations against deep neural networks.

Payne, J., Fenner, M., and Kauffman, J. (2013). System event monitoring for active authentication. *IT Professional*, 15(4):34–37.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830.

PELock (2016). Pelock. https://www.pelock.com/.

Pendlebury, F., Pierazzi, F., Jordaney, R., Kinder, J., and Cavallaro, L. (2019). TESSERACT: Eliminating experimental bias in malware classification across space and time. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 729–746, Santa Clara, CA. USENIX Association.

Permeh, R. (2017). True ai/ml vs. glorified signature-based solutions. https://threatvector.cylance.com/en_us/home/true-ai-ml-vs-glorified-signature-based-solutions.html.

Phong, L. T., Aono, Y., Hayashi, T., Wang, L., and Moriai, S. (2018). Privacy-preserving deep learning via additively homomorphic encryption. *IEEE Transactions on Information Forensics and Security*, 13(5):1333–1345.

Pinage, F. A., dos Santos, E. M., and da Gama, J. M. P. (2016). Classification systems in dynamic environments. *WIREs: Data Mining and Knowledge Discovery*.

Quarkslab (2019). Lief. https://lief.quarkslab.com/doc/stable/api/python/pe.html.

Quiring, E., Pirch, L., Reimsbach, M., Arp, D., and Rieck, K. (2020). Against all odds: Winning the defense challenge in an evasion competition with diversification.

Raab, C., Heusinger, M., and Schleif, F.-M. (2020). Reactive soft prototype computing for concept drift streams. *Neurocomputing*, 416:340–351.

Radford, A. (2018). Improving language understanding by generative pre-training.

Raff, E., Barker, J., Sylvester, J., Brandon, R., Catanzaro, B., and Nicholas, C. (2017). Malware detection by eating a whole exe. https://arxiv.org/abs/1710.09435.

Raghu, M., Zhang, C., Kleinberg, J. M., and Bengio, S. (2019). Transfusion: Understanding transfer learning with applications to medical imaging. *CoRR*, abs/1902.07208.

Rezaei, S. and Liu, X. (2019). A target-agnostic attack on deep models: Exploiting security vulnerabilities of transfer learning. *CoRR*, abs/1904.04334.

Ribeiro, B., Pereira, H., Almeida, R., Ferreira, A., Martins, L., Quaresma, C., and Vieira, P. (2015). Optimization of sitting posture classification based on user identification. In *2015 IEEE 4th Portuguese Meeting on Bioengineering (ENBENG)*, pages 1–6. IEEE.

Richman, J. S. and Moorman, J. R. (2000). Physiological time-series analysis using approximate entropy and sample entropy. *American Journal of Physiology-Heart and Circulatory Physiology*, 278(6):H2039–H2049.

Rieck, K. (2011). Computer security and machine learning: Worst enemies or best friends? In *2011 First SysSec Workshop*, pages 107–110.

Rocha, A., Scheirer, W. J., Forstall, C. W., Cavalcante, T., Theophilo, A., Shen, B., Carvalho, A. R. B., and Stamatatos, E. (2017). Authorship attribution for social media forensics. *IEEE Transactions on Information Forensics and Security*, 12(1):5–33.

Roh, Y., Heo, G., and Whang, S. (2019). A survey on data collection for machine learning: A big data - ai integration perspective. *IEEE Transactions on Knowledge and Data Engineering*, PP:1–1.

Rossow, C., Dietrich, C., and Bos, H. (2013). Large-scale analysis of malware downloaders. In Flegel, U., Markatos, E., and Robertson, W., editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 42–61, Berlin, Heidelberg. Springer Berlin Heidelberg.

Russinovich, M. (2019). Process monitor - windows sysintenals.

Russinovich, M. and Garnier, T. (2019). Sysmon - windows sysintenals.

Salehi, M. and Rashidi, L. (2018). A survey on anomaly detection in evolving data: [with application to forest fire risk prediction]. *ACM SIGKDD Explorations Newsletter*, 20(1):13–23.

Salton, G., Wong, A., and Yang, C. S. (1975). A vector space model for automatic indexing. *Commun. ACM*.

Sarma, B. P., Li, N., Gates, C., Potharaju, R., Nita-Rotaru, C., and Molloy, I. (2012). Android permissions: A perspective combining risks and benefits. In *Symp. on Access Control Models and Technologies*.

Saxe, J. (2020). Sophos ai yaraml rules repository. `https://github.com/sophos-ai/yaraml_rules`.

Saxe, J. and Sanders, H. (2018). *Malware Data Science: Attack Detection and Attribution*. No Starch Press, San Francisco, CA, USA.

Sayed, B., Traore, I., Woungang, I., and Obaidat, M. S. (2013). Biometric authentication using mouse gesture dynamics. *IEEE Systems Journal*, 7(2):262–274.

Schelter, S., Bießmann, F., Januschowski, T., Salinas, D., Seufert, S., and Szarvas, G. (2018). On challenges in machine learning model management. *IEEE Data Eng. Bull.*, 41:5–15.

Schölkopf, B., Williamson, R., Smola, A., Shawe-Taylor, J., and Platt, J. (1999). Support vector method for novelty detection. In *Proceedings of the 12th International Conference on Neural Information Processing Systems*, NIPS'99, page 582–588, Cambridge, MA, USA. MIT Press.

Science Buddies (2020). Steps of the scientific method. `https://www.sciencebuddies.org/science-fair-projects/science-fair/steps-of-the-scientific-method`.

scikit learn (2020a). Minmaxscaler. `https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html`.

scikit learn (2020b). Onehotencoder. `https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html#sklearn.preprocessing.OneHotEncoder`.

scikit learn (2020c). Randomforestclassifier. `https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html`.

scikit learn (2020d). Tfidfvectorizer. `https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html`.

Sebastián, M., Rivera, R., Kotzias, P., and Caballero, J. (2016). Avclass: A tool for massive malware labeling. In Monrose, F., Dacier, M., Blanc, G., and Garcia-Alfaro, J., editors, *Research in Attacks, Intrusions, and Defenses*, pages 230–253, Cham. Springer International Publishing.

SecurityVentures (2018). Global ransomware damage costs predicted to exceed $8 billion in 2018. https://tinyurl.com/y499nvsh.

Services, A. W. (2020). Amazon machine learning key concepts. https://docs.aws.amazon.com/machine-learning/latest/dg/amazon-machine-learning-key-concepts.html.

Shafahi, A., Saadatpanah, P., Zhu, C., Ghiasi, A., Studer, C., Jacobs, D. W., and Goldstein, T. (2019). Adversarially robust transfer learning. *CoRR*, abs/1905.08232.

Shahraki, A., Abbasi, M., Taherkordi, A., and Jurcut, A. D. (2022). A comparative study on online machine learning techniques for network traffic streams analysis. *Computer Networks*, 207:108836.

Shao, J., Ahmadi, Z., and Kramer, S. (2014). Prototype-based learning on concept-drifting data streams. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '14, page 412–421, New York, NY, USA. Association for Computing Machinery.

Shao, P. and Smith, R. K. (2009). Feature location by ir modules and call graph. In *ACM-SE 47*.

Shi, H. and Li, Y. (2018). Discovering periodic patterns for large scale mobile traffic data: method and applications. *IEEE Transactions on Mobile Computing*, 17(10):2266–2278.

Shiravi, A., Shiravi, H., Tavallaee, M., and Ghorbani, A. A. (2012). Toward developing a systematic approach to generate benchmark datasets for intrusion detection. *computers & security*, 31(3):357–374.

Shoesmith, E., Box, G. E. P., and Draper, N. R. (1987). Empirical model-building and response surfaces. *The Statistician*, 37:82.

Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., and Vigna, G. (2016). SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*.

Siahroudi, S. K., Moodi, P. Z., and Beigy, H. (2018). Detection of evolving concepts in non-stationary data streams: A multiple kernel learning approach. *Expert Systems with Applications*, 91:187–197.

Singh, A., Walenstein, A., and Lakhotia, A. (2012). Tracking concept drift in malware families. In *Proceedings of the ACM Workshop on Security and Artificial Intelligence*.

Smirnov, A. (2016). Xor-decrypt. https://github.com/AlexFSmirnov/xor-decrypt.

Spark, A. (2020). Spark streaming.

ssdeep (2017). ssdeep - fuzzy hashing program. https://ssdeep-project.github.io/ssdeep/index.html.

StatCounter (2018). Operating System Market. https://tinyurl.com/tykvtqm.

Steele, J. M. (2006). Models: Masterpieces and lame excuses. http://www-stat.wharton.upenn.edu/~steele/Rants/ModelsMandLE.html.

stevielavern (2017). Cannot add section and rebuild pe on windows 10 #109. https://github.com/lief-project/LIEF/issues/109.

Sugrim, S., Liu, C., McLean, M., and Lindqvist, J. (2019). Robust performance metrics for authentication systems. In *Network and Distributed Systems Security (NDSS) Symposium 2019*.

Sun, R., Botacin, M., Sapountzis, N., Yuan, X., Bishop, M., Porter, D. E., Li, X., Gregio, A., and Oliveira, D. (2020). A praise for defensive programming: Leveraging uncertainty for effective malware mitigation. *IEEE Transactions on Dependable and Secure Computing*, pages 1–1.

Sun, Y., Ceker, H., and Upadhyaya, S. (2016). Shared keystroke dataset for continuous authentication. In *2016 IEEE International Workshop on Information Forensics and Security (WIFS)*, pages 1–6.

Symantec (2015). W32.ramnit. https://www.symantec.com/security-center/writeup/2010-011922-2056-99.

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S. E., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. (2014). Going deeper with convolutions. *CoRR*, abs/1409.4842.

Taheri, R., Javidan, R., Shojafar, M., Pooranian, Z., Miri, A., and Conti, M. (2019). On defending against label flipping attacks on malware detection systems.

Tan, S. C., Ting, K. M., and Liu, T. F. (2011). Fast anomaly detection for streaming data. In *Twenty-Second International Joint Conference on Artificial Intelligence*.

Tasiopoulos, V. G. and Katsikas, S. K. (2014). Bypassing antivirus detection with encryption. In *Proceedings of the 18th Panhellenic Conference on Informatics*, PCI '14, page 1–2, New York, NY, USA. Association for Computing Machinery.

Technologies, O. (2011). Themida. https://www.oreans.com/Themida.php.

Telock (2018). Telock. https://www.softpedia.com/get/Programming/Packers-Crypters-Protectors/Telock.shtml.

Theiler, J., Eubank, S., Longtin, A., Galdrikian, B., and Farmer, J. D. (1992). Testing for nonlinearity in time series: the method of surrogate data. *Physica D: Nonlinear Phenomena*, 58(1-4):77–94.

Torrey, L. and Shavlik, J. (2010). Transfer learning. In *Handbook of research on machine learning applications and trends: algorithms, methods, and techniques*, pages 242–264. IGI global.

Tossell, C., Kortum, P., Rahmati, A., Shepard, C., and Zhong, L. (2012). Characterizing web use on smartphones. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '12, pages 2769–2778, New York, NY, USA. ACM.

Total, V. (2019). Virus total. https://www.virustotal.com.

Ugarte-Pedrero, X., Balzarotti, D., Santos, I., and Bringas, P. G. (2015). Sok: Deep packer inspection: A longitudinal study of the complexity of run-time packers. In *2015 IEEE Symposium on Security and Privacy*, pages 659–673, US. IEEE.

Underwood, T. (2019). All of our ML ideas are bad (and we should feel bad). Dublin. USENIX Association.

VirusShare (2019). Virusshare on twitter. `https://twitter.com/VXShare/status/1095411986949652480`.

Wagstaff, K. (2012). Machine learning that matters. *CoRR*, abs/1206.4656.

Wang, S., Schlobach, S., and Klein, M. (2011). Concept drift and how to identify it. *Web Semantics: Science, Services and Agents on the World Wide Web*, 9(3):247 – 265. Semantic Web Dynamics Semantic Web Challenge, 2010.

Xu, K., Li, Y., Deng, R., Chen, K., and Xu, J. (2019). Droidevolver: Self-evolving android malware detection system. In *2019 IEEE European Symposium on Security and Privacy (EuroS P)*.

Yang, J., Qiao, Y., Zhang, X., He, H., Liu, F., and Cheng, G. (2015). Characterizing user behavior in mobile internet. *IEEE Transactions on Emerging Topics in Computing*, 3(1):95–106.

Yang, L., Guo, W., Hao, Q., Ciptadi, A., Ahmadzadeh, A., Xing, X., and Wang, G. (2021). CADE: Detecting and explaining concept drift samples for security applications. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2327–2344. USENIX Association.

Ye, Y., Li, T., Adjeroh, D., and Iyengar, S. S. (2017). A survey on malware detection using data mining techniques. *ACM Comput. Surv.*, 50(3).

Yuan, J., Shalaby, W., Korayem, M., Lin, D., AlJadda, K., and Luo, J. (2016). Solving cold-start problem in large-scale recommendation engines: A deep learning approach. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 1901–1910. IEEE.

ZareMoodi, P., Beigy, H., and Kamali Siahroudi, S. (2015). Novel class detection in data streams using local patterns and neighborhood graph. *Neurocomputing*, 158:234–245.

ZareMoodi, P., Kamali Siahroudi, S., and Beigy, H. (2019). Concept-evolution detection in non-stationary data streams: a fuzzy clustering approach. *Knowledge and Information Systems*, 60.

Zhang, H., Patel, V. M., Fathy, M., and Chellappa, R. (2015). Touch gesture-based active user authentication using dictionaries. In *2015 IEEE Winter Conference on Applications of Computer Vision (WACV)*, volume 00, pages 207–214.

Zhang, X., Zhang, Y., Zhong, M., Ding, D., Cao, Y., Zhang, Y., Zhang, M., and Yang, M. (2020). Enhancing state-of-the-art classifiers with api semantics to detect evolved android malware. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 757–770, New York, NY, USA. Association for Computing Machinery.

Zhou, Y. and Jiang, X. (2012). Dissecting android malware: Characterization and evolution. In *2012 IEEE Symposium on Security and Privacy*, pages 95–109.

Zhu, S., Shi, J., Yang, L., Qin, B., Zhang, Z., Song, L., and Wang, G. (2020). Measuring and modeling the label dynamics of online anti-malware engines. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2361–2378. USENIX Association.

Žliobaité, I. (2010). Change with delayed labeling: When is it detectable? In *2010 IEEE International Conference on Data Mining Workshops*, pages 843–850.

APPENDIX A – APPENDIX FOR THE "SHALLOW SECURITY: ON THE CREATION OF ADVERSARIAL VARIANTS TO EVADE MACHINE LEARNING-BASED MALWARE DETECTORS" PAPER

A.1 DATASET SAMPLES CLASSIFICATION

Table A.1 shows the labels assigned to each sample by each classifier and their respective confidence levels.

Table A.1: **Class (M for malware and G for goodware) and confidence (%) of each original and adversarial sample when classifying them using the three ML models proposed by the challenge.** All adversarial samples are considered as being a goodware, the majority of them with a high confidence level.

		MalConv				NonNeg. MalConv				LightGDM			
Family	**Sample**	**Original**		**Adversarial**		**Original**		**Adversarial**		**Original**		**Adversarial**	
		Classification	**Confidence**	**Classification**	**Confidence**	**Classification**	**Confidence**	**Classification**	**Confidence**	**Classification**	**Confidence**	**Classification**	**Confidence**
gandcrab	001	M	97.83%	G	65.96%	M	54.30%	G	90.53%	M	100.00%	G	87.83%
cerber	002	M	96.28%	G	85.34%	M	63.96%	G	98.65%	M	99.91%	G	82.83%
xtrat	003	M	99.60%	G	76.84%	M	60.81%	G	98.41%	M	100.00%	G	51.56%
lethic	004	M	95.33%	G	83.61%	M	63.33%	G	98.65%	M	100.00%	G	99.28%
tinyloader	005	M	66.92%	G	65.96%	M	59.90%	G	98.47%	M	100.00%	G	80.86%
tinyloader	006	M	64.22%	G	78.18%	M	94.85%	G	95.35%	M	100.00%	G	90.82%
sality	007	M	99.69%	G	83.25%	M	63.81%	G	98.65%	M	100.00%	G	99.98%
xtrat	008	M	99.90%	G	72.25%	M	50.27%	G	66.01%	M	100.00%	G	63.04%
emotet	009	M	99.81%	G	75.34%	M	59.95%	G	98.41%	M	100.00%	G	62.58%
sality	010	M	87.27%	G	73.72%	M	66.53%	G	98.41%	M	100.00%	G	78.61%
lethic	011	M	61.42%	G	87.97%	M	59.80%	G	98.41%	M	100.00%	G	96.74%
mirai	012	M	53.11%	G	68.05%	M	64.03%	G	98.65%	M	100.00%	G	99.91%
nanocore	013	M	59.90%	G	78.98%	M	50.03%	G	98.41%	M	100.00%	G	94.55%
ramnit	014	M	99.80%	G	87.12%	M	54.61%	G	98.65%	M	100.00%	G	99.84%
emotet	015	M	97.46%	G	73.52%	M	67.72%	G	98.41%	M	99.79%	G	99.70%
emotet	016	M	96.13%	G	81.04%	M	52.18%	G	98.41%	M	99.96%	G	99.72%
ramnit	017	M	88.61%	G	87.86%	M	66.76%	G	95.20%	M	100.00%	G	99.97%
emotet	018	M	95.95%	G	70.34%	M	51.16%	G	98.41%	M	99.73%	G	99.75%
trickbot	019	M	65.76%	G	85.18%	M	50.07%	G	97.98%	M	100.00%	G	51.74%
yakes	020	M	92.88%	G	74.48%	M	63.52%	G	98.41%	M	99.78%	G	99.93%
azorult	021	M	99.17%	G	90.95%	M	83.35%	G	81.54%	M	100.00%	G	56.41%
wapomi	022	M	97.75%	G	84.08%	M	51.15%	G	98.65%	M	100.00%	G	99.91%
loki	023	M	78.93%	G	65.96%	M	64.89%	G	98.79%	M	100.00%	G	60.90%
cutwail	024	M	98.86%	G	89.39%	M	57.83%	G	90.54%	M	99.38%	G	95.07%
trickster	025	M	67.38%	G	75.15%	M	58.97%	G	98.41%	M	100.00%	G	97.77%
ursnif	026	M	99.43%	G	76.26%	M	61.83%	G	98.78%	M	99.93%	G	99.66%
ramnit	027	M	71.84%	G	84.76%	M	66.78%	G	98.65%	M	100.00%	G	99.94%
lethic	028	M	99.71%	G	80.90%	M	51.98%	G	98.41%	M	100.00%	G	99.57%
loki	029	M	96.29%	G	67.93%	M	94.36%	G	96.67%	M	100.00%	G	59.55%
tinyloader	030	M	90.86%	G	76.63%	M	79.99%	G	95.35%	M	100.00%	G	90.60%
kovter	031	M	95.08%	G	94.44%	M	63.90%	G	97.27%	M	99.99%	G	88.78%
cerber	032	M	98.84%	G	71.65%	M	51.51%	G	98.41%	M	100.00%	G	65.99%
wapomi	033	M	74.44%	G	65.96%	M	57.32%	G	97.46%	M	100.00%	G	82.24%
azorult	034	M	61.45%	G	79.49%	M	66.28%	G	98.41%	M	100.00%	G	99.82%
xtrat	035	M	99.99%	G	81.22%	M	75.09%	G	98.65%	M	100.00%	G	99.97%
ursnif	036	M	95.56%	G	78.34%	M	57.14%	G	98.41%	M	100.00%	G	81.82%
loki	037	M	62.48%	G	65.96%	M	77.32%	G	98.41%	M	100.00%	G	64.30%
xtrat	038	M	100.00%	G	87.43%	M	59.22%	G	95.36%	M	100.00%	G	99.95%
loki	039	M	70.72%	G	79.54%	M	62.07%	G	98.41%	M	100.00%	G	50.10%
cutwail	040	M	98.79%	G	65.96%	M	50.89%	G	66.29%	M	100.00%	G	98.94%
ramnit	041	M	97.61%	G	82.01%	M	55.83%	G	98.65%	M	100.00%	G	99.91%
mansabo	042	M	74.98%	G	65.96%	M	77.01%	G	98.41%	M	100.00%	G	81.72%
qbot	043	M	87.64%	G	65.96%	M	51.17%	G	98.87%	M	100.00%	G	99.02%
wapomi	044	M	86.50%	G	82.33%	M	70.80%	G	98.65%	M	100.00%	G	99.98%
sality	045	M	58.42%	G	86.55%	M	81.19%	G	92.57%	M	100.00%	G	82.26%
azorult	046	M	99.52%	G	69.33%	M	98.83%	G	94.59%	M	100.00%	G	73.76%
emotet	047	M	98.84%	G	74.40%	M	51.48%	G	98.41%	M	100.00%	G	65.38%
ursnif	048	M	99.96%	G	93.20%	M	51.36%	G	81.54%	M	99.99%	G	76.67%
nanocore	049	G	95.83%	G	74.79%	M	50.59%	G	98.41%	M	100.00%	G	94.56%
trickbot	050	M	99.92%	G	66.87%	M	58.13%	G	98.41%	M	100.00%	G	80.62%

APPENDIX B – APPENDIX FOR THE "ONLINE BINARY MODELS ARE PROMISING FOR DISTINGUISHING TEMPORALLY CONSISTENT COMPUTER USAGE PROFILES" PAPER

B.1 STUDY PARTICIPANTS' DEMOGRAPHICS

Table B.1 summarizes the demographics of our 31 study participants.

B.2 FEATURE EXTRACTION DETAILS

Figure B.1 illustrates the sliding window strategy used in our ML experiments for a window size of 3 minutes. As shown, we summed the number of clicks, keystrokes, and background traffic activity (all integer numbers), and concatenated the strings of the processes and domains used within consecutive t-minute windows ($t = 3$ in the given example), keeping the last timestamp of the window. For an original matrix containing N lines of active minutes, this process groups computer usage in t-minute windows.

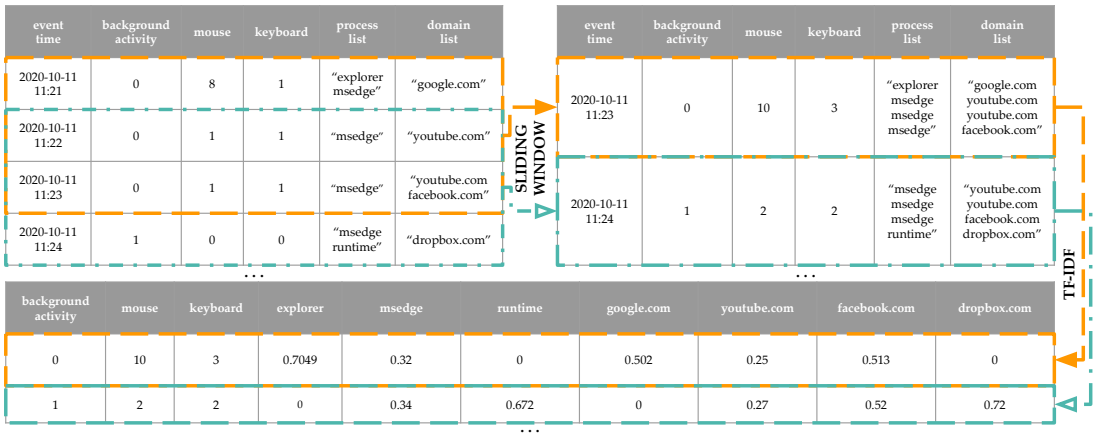


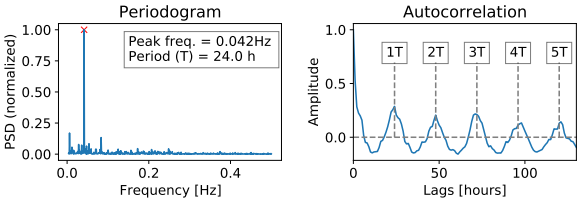
Figure B.1: **Sliding window.** Example of sliding window feature extraction with window size $t = 3$ minutes (illustrative TF-IDF values).

B.3 SURROGATE DATA TESTING RESULTS

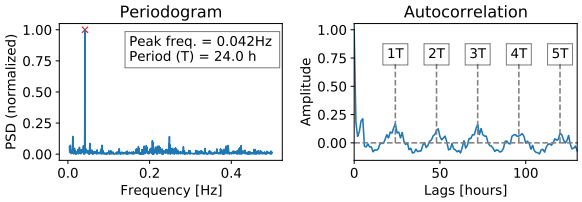
Table B.2 exhibits the results of the statistical comparison for the sample entropy and Hurst exponent obtained from the time series of computer usage profiles vs. their surrogate counterparts in the two assessed conditions: with and without background process activity.

B.4 PROFILES' TEMPORAL CONSISTENCY RESULTS

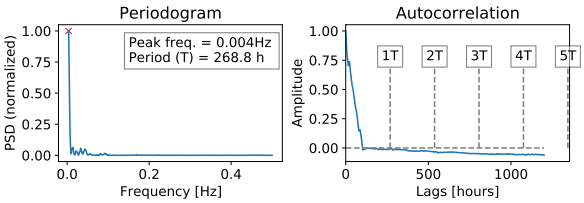
Figure B.2 exhibits the periodogram and autocorrelation results for all 31 study participants in the two assessed conditions: with and without background process activity.



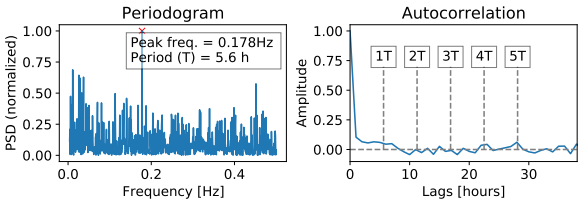
i User 1, condition 1



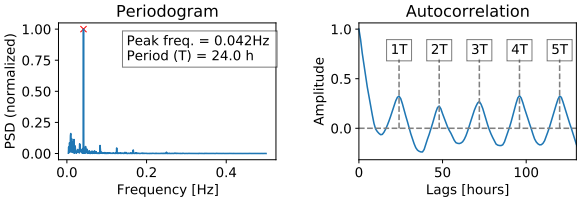
ii User 1, condition 2



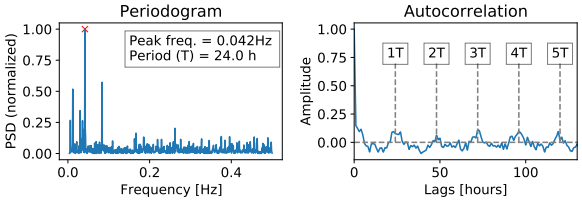
iii User 2, condition 1



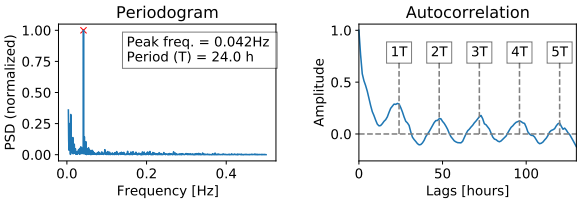
iv User 2, condition 2



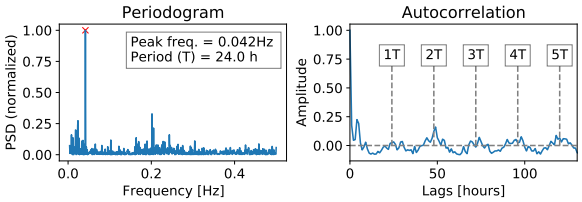
v User 3, condition 1



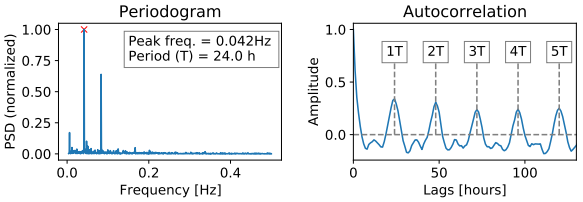
vi User 3, condition 2



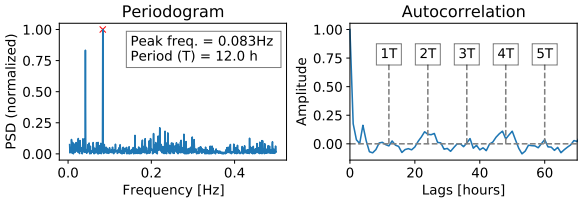
vii User 4, condition 1



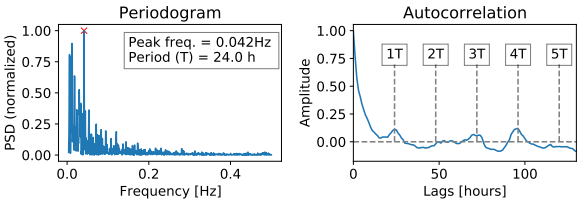
viii User 4, condition 2



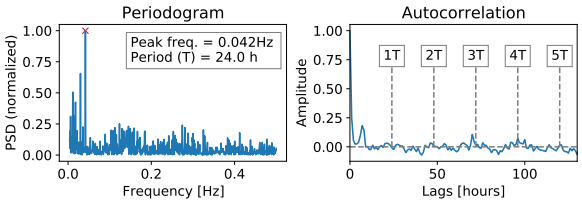
ix User 5, condition 1



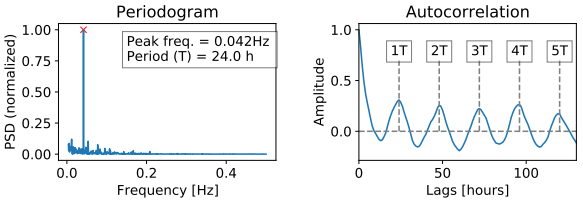
x User 5, condition 2



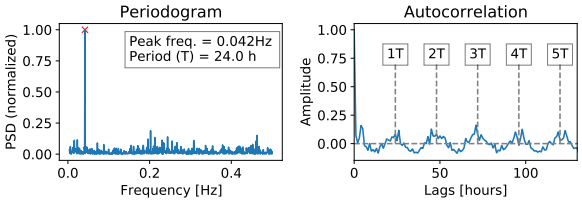
xi User 6, condition 1



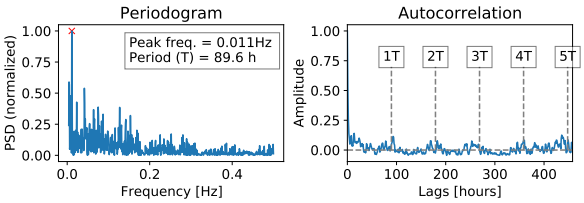
xii User 6, condition 2



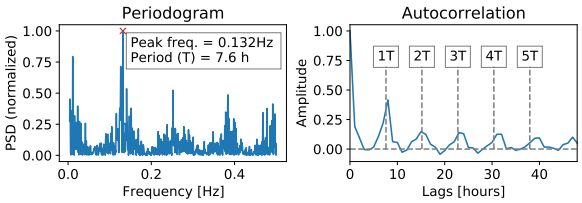
xiii User 7, condition 1



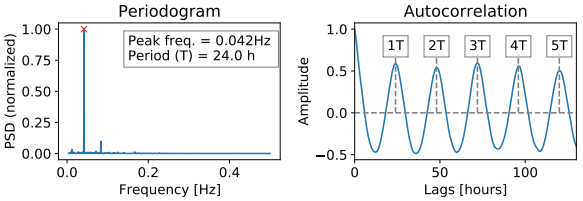
xiv User 7, condition 2



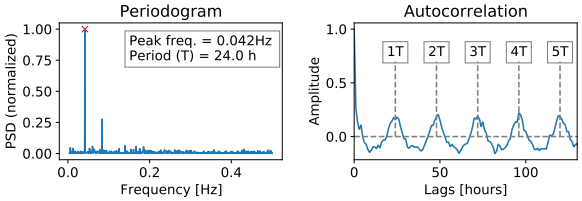
xv User 8, condition 1



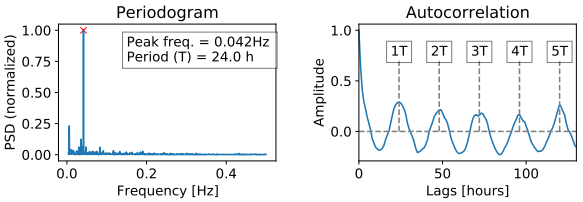
xvi User 8, condition 2



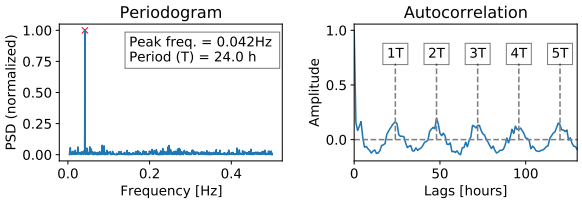
xvii User 9, condition 1



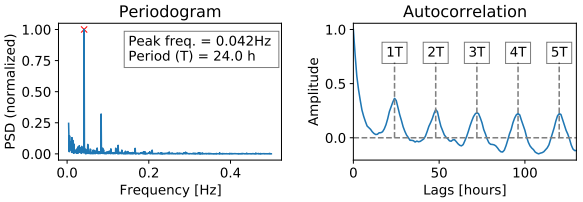
xviii User 9, condition 2



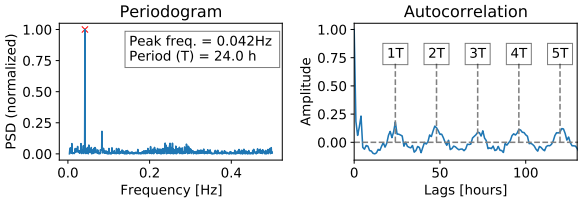
xix User 10, condition 1



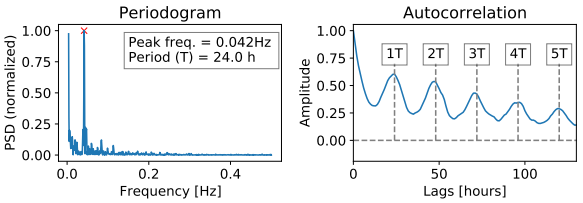
xx User 10, condition 2



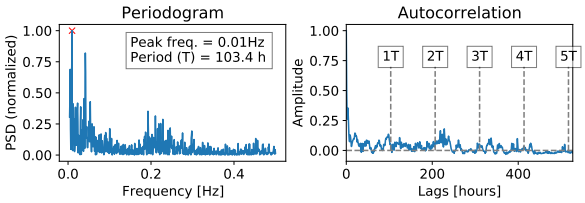
xxi User 11, condition 1



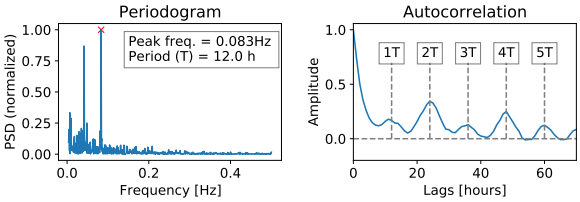
xxii User 11, condition 2



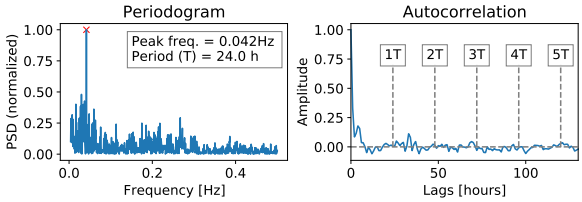
xxiii User 12, condition 1



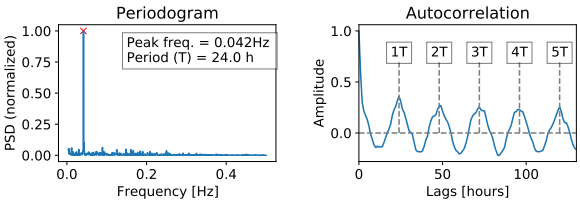
xxiv User 12, condition 2



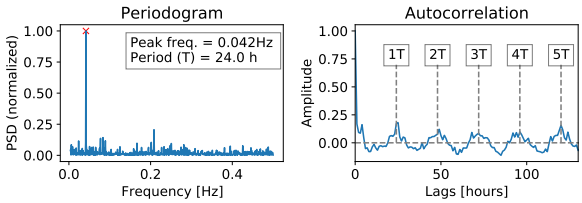
xxv User 13, condition 1



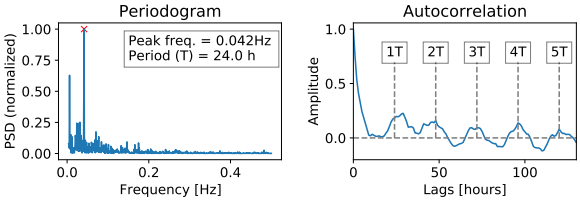
xxvi User 13, condition 2



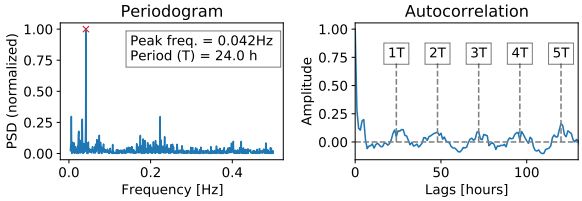
xxvii User 14, condition 1



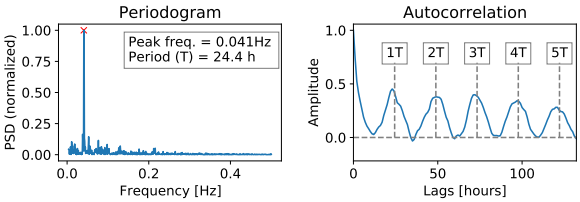
xxviii User 14, condition 2



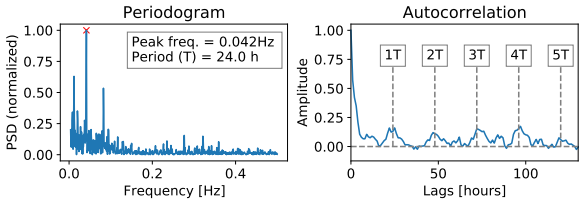
xxix User 15, condition 1



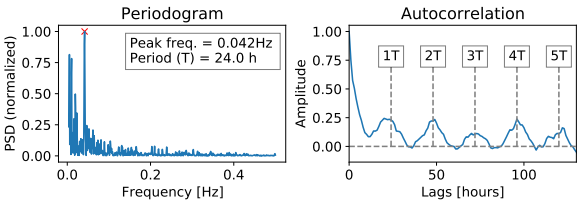
xxx User 15, condition 2



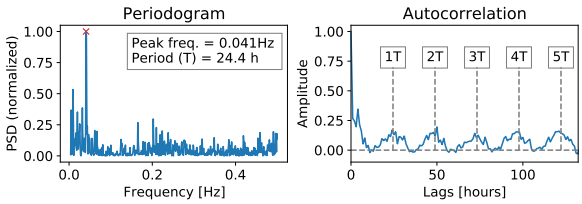
xxxi User 16, condition 1



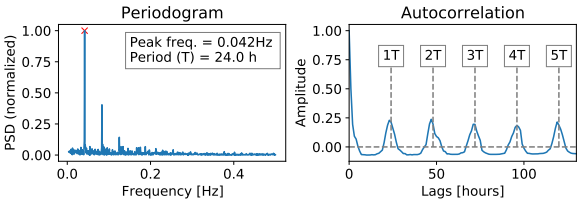
xxxii User 16, condition 2



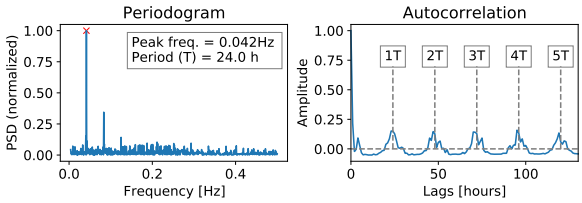
xxxiii User 17, condition 1



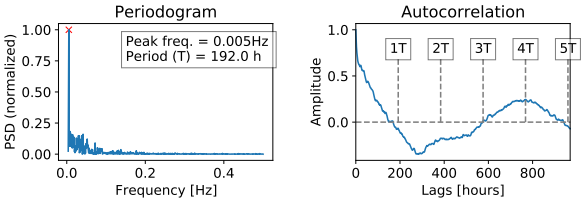
xxxiv User 17, condition 2



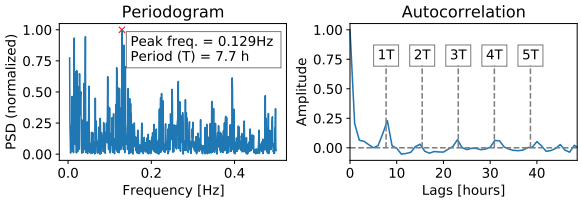
xxxv User 18, condition 1



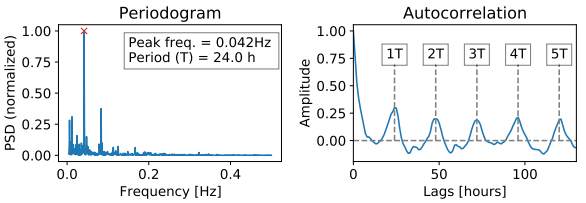
xxxvi User 18, condition 2



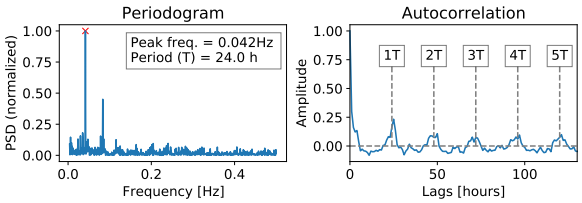
xxxvii User 19, condition 1



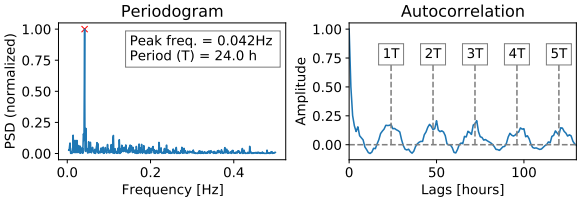
xxxviii User 19, condition 2



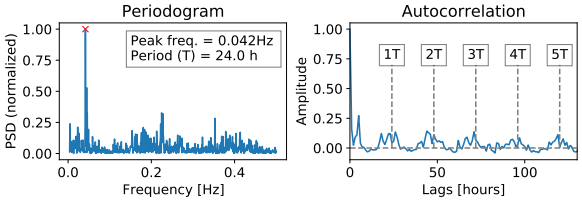
xxxix User 20, condition 1



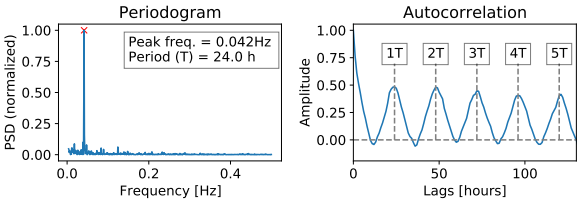
xl User 20, condition 2



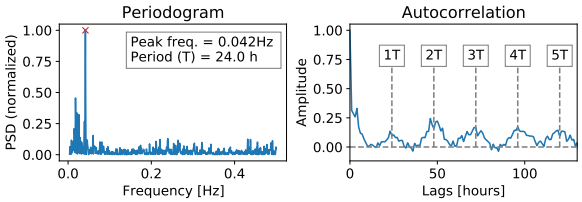
xli User 21, condition 1



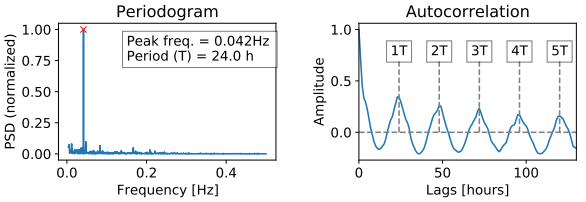
xlii User 21, condition 2



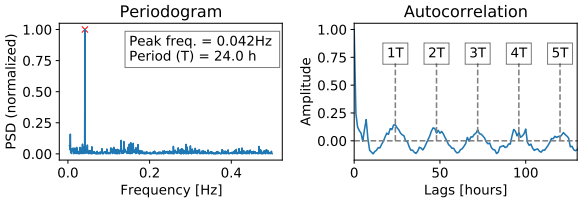
xliii User 22, condition 1



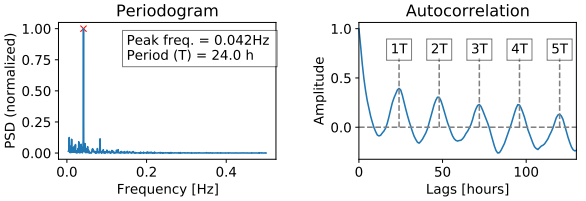
xliv User 22, condition 2



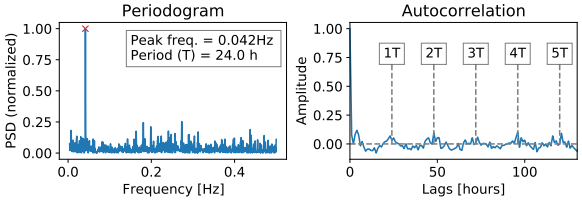
xlv User 23, condition 1



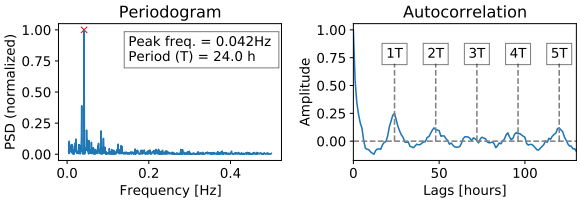
xlvi User 23, condition 2



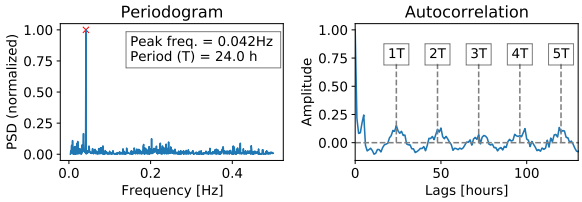
xlvii User 24, condition 1



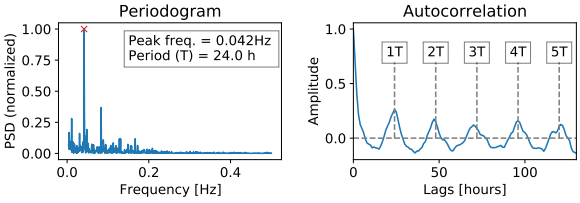
xlviii User 24, condition 2



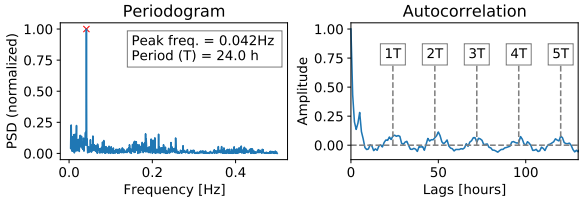
xlix User 25, condition 1



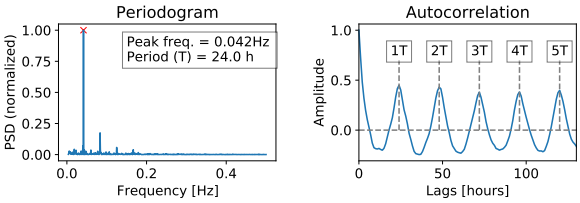
l User 25, condition 2



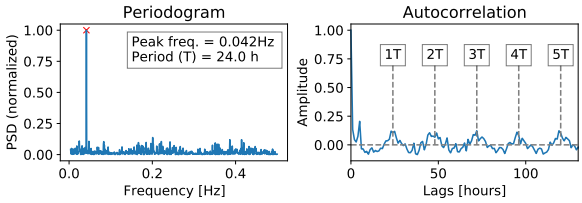
li User 26, condition 1



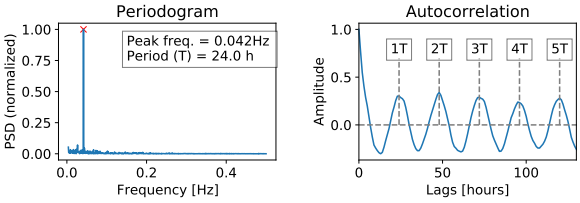
lii User 26, condition 2



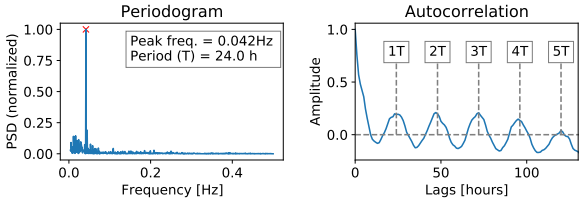
liii User 27, condition 1



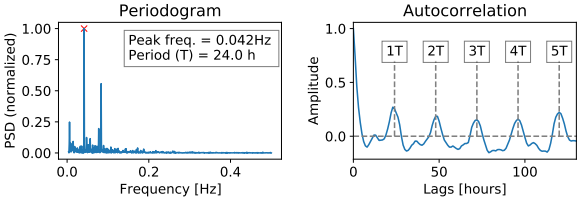
liv User 27, condition 2



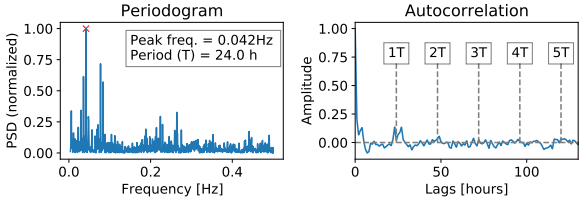
lv User 28, condition 1



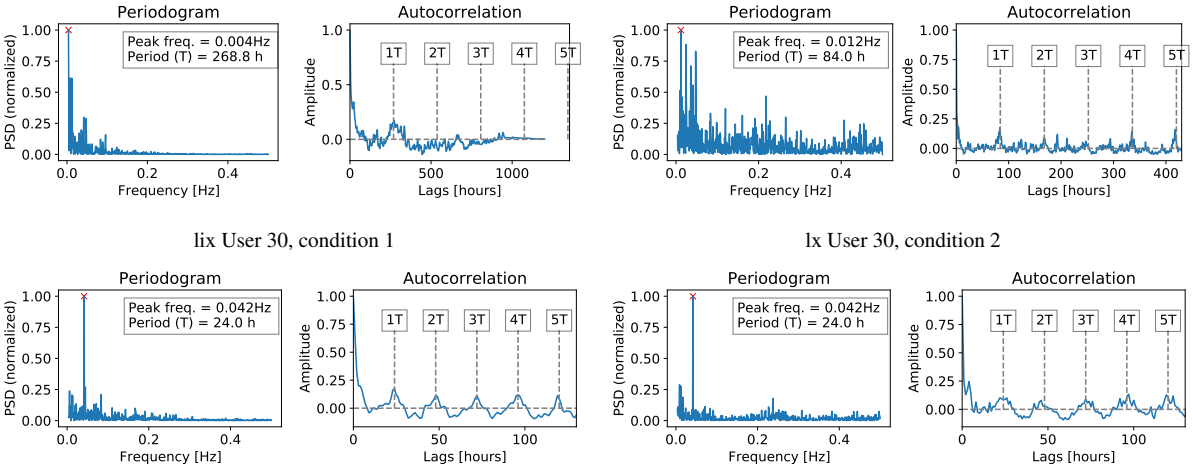
lvi User 28, condition 2



lvii User 29, condition 1



lviii User 29, condition 2



lix User 30, condition 1

lx User 30, condition 2

lxi User 31, condition 1

lxii User 31, condition 2

Figure B.2: **Periodogram and autocorrelation.** Results for all 31 participants in condition 1 (with background process activity) and condition 2 (without background process activity).

B.5 COMPLETE CLASSIFICATION RESULTS

Tables B.3, B.4, and B.5 exhibit the classification results for all window sizes obtained with the offline binary models, offine one-class models, and online binary models, respectively.

Table B.1: **Summary**. Study participants' demographics.

Category	Metric	Total (N = 31)
Gender	Female	20 (64.52%)
	Male	10 (32.26%)
	Gender Variant/ Non-Conforming	1 (3.23%)
Age	18–25 years	16 (46.43%)
	26–35 years	12 (39.28%)
	36–45 years	2 (7.14%)
	≥46 years	2 (7.14%)
Highest Formal Degree	Associate	3 (9.68%)
	Bachelor's	11 (35.48%)
	Master's	0 (0.00%)
	PhD/Doctorate	5 (16.13%)
	Other	12 (38.71%)
Marital Status	Single	7 (22.58%)
	Married	11 (35.48%)
	Divorced	3 (9.68%)
	In a relationship	10 (32.26%)
Living Condition	Alone	5 (16.13%)
	With spouse/S.O.*	13 (41.94%)
	With child(ren)	3 (9.68%)
	Assisted living	0 (0.00%)
	Other	10 (32.26%)
Employment Status	Employed	17 (54.84%)
	Unemployed	14 (45.16%)
	Retired	0 (0.00%)
Yearly Household Income	<$10,000	9 (29.03%)
	$10,000 to <$50,000	10 (32.26%)
	$50,000 to <$100,000	6 (19.35%)
	≥$100,000	6 (19.35%)
Latino Ethnicity	Not Hispanic/Latino	23 (74.19%)
	Hispanic/Latino	8 (25.81%)
Primary Language	English	24 (77.42%)
	Portuguese	4 (12.90%)
	Sinhala	2 (6.45%)
	Chinese	1 (3.23%)

∗ S.O. = significant other

Table B.2: **Statistical comparison.** Statistical comparison for the sample entropy and Hurst exponent, obtained by leveraging the time series of computer usage profiles against their surrogate counterparts. *Condition 1* denotes that background process activities were assessed, and *Condition 2* indicates that these background activities were *not* assessed. All users have been de-identified.

	Sample Entropy										**Hurst Exponent**									
	Condition 1					**Condition 2**					**Condition 1**					**Condition 2**				
User	**Time Series**		**Surrogates**		**p**	**Time Series**		**Surrogates**		**p**	**Time Series**		**Surrogates**		**p**	**Time Series**		**Surrogates**		**p**
	μ	σ	μ	σ		μ	σ	μ	σ		μ	σ	μ	σ		μ	σ	μ	σ	
1	0.215	0.000	0.568	0.026	***	0.259	0.000	0.467	0.022	***	0.583	0.022	0.500	0.040	***	0.538	0.020	0.496	0.035	***
2	0.146	0.000	0.295	0.011	***	0.428	0.000	0.470	0.019	***	0.639	0.008	0.488	0.039	***	0.546	0.000	0.489	0.031	***
3	0.263	0.000	0.951	0.014	***	0.469	0.000	0.743	0.029	***	0.765	0.013	0.499	0.035	***	0.644	0.002	0.498	0.034	***
4	0.214	0.000	0.901	0.033	***	0.309	0.000	0.488	0.021	***	0.778	0.000	0.504	0.038	***	0.642	0.006	0.487	0.035	***
5	0.539	0.000	1.237	0.044	***	0.559	0.000	0.975	0.040	***	0.496	0.016	0.496	0.037	0.989	0.517	0.004	0.493	0.034	***
6	0.126	0.000	0.370	0.015	***	0.152	0.000	0.247	0.010	***	0.737	0.000	0.495	0.039	***	0.630	0.002	0.487	0.030	***
7	0.228	0.000	0.934	0.032	***	0.285	0.000	0.410	0.016	***	0.679	0.011	0.493	0.041	***	0.509	0.008	0.486	0.036	***
8	0.092	0.000	0.159	0.007	***	0.098	0.000	0.154	0.005	***	0.635	0.018	0.469	0.023	***	0.649	0.012	0.477	0.030	***
9	0.258	0.000	0.899	0.009	***	0.403	0.000	1.050	0.043	***	0.326	0.045	0.495	0.040	***	0.348	0.030	0.502	0.037	***
10	0.234	0.000	0.648	0.027	***	0.294	0.000	0.444	0.020	***	0.591	0.033	0.498	0.042	***	0.383	0.026	0.496	0.037	***
11	0.181	0.000	0.646	0.027	***	0.205	0.000	0.415	0.017	***	0.578	0.019	0.493	0.034	***	0.424	0.007	0.485	0.033	***
12	0.087	0.000	0.432	0.016	***	0.100	0.000	0.279	0.011	***	0.552	0.014	0.495	0.036	***	0.628	0.013	0.482	0.032	***
13	0.378	0.000	0.991	0.044	***	0.494	0.000	0.846	0.038	***	0.617	0.049	0.503	0.037	***	0.546	0.019	0.488	0.034	***
14	0.226	0.000	0.668	0.027	***	0.329	0.000	0.552	0.025	***	0.548	0.039	0.500	0.036	***	0.543	0.051	0.489	0.032	***
15	0.173	0.000	0.552	0.023	***	0.226	0.000	0.496	0.019	***	0.646	0.023	0.502	0.037	***	0.585	0.028	0.492	0.036	***
16	0.147	0.000	0.784	0.040	***	0.218	0.000	0.464	0.016	***	0.462	0.003	0.495	0.037	***	0.628	0.001	0.483	0.035	***
17	0.065	0.000	0.281	0.012	***	0.085	0.000	0.198	0.007	***	0.682	0.002	0.494	0.036	***	0.675	0.001	0.484	0.036	***
18	0.143	0.000	0.216	0.009	***	0.177	0.000	0.212	0.009	***	0.565	0.052	0.484	0.029	***	0.437	0.017	0.476	0.030	***
19	0.162	0.000	0.951	0.008	***	0.321	0.000	0.443	0.019	***	0.791	0.000	0.491	0.040	***	0.626	0.007	0.484	0.033	***
20	0.230	0.000	0.807	0.029	***	0.265	0.000	0.579	0.026	***	0.686	0.012	0.493	0.038	***	0.535	0.033	0.488	0.033	***
21	0.123	0.000	0.283	0.012	***	0.097	0.000	0.153	0.006	***	0.564	0.003	0.488	0.034	***	0.566	0.006	0.477	0.029	***
22	0.113	0.000	0.610	0.026	***	0.156	0.000	0.388	0.017	***	0.485	0.004	0.496	0.034	0.0033	0.627	0.005	0.493	0.031	***
23	0.175	0.000	0.621	0.026	***	0.256	0.000	0.495	0.022	***	0.538	0.073	0.494	0.037	***	0.558	0.010	0.500	0.040	***
24	0.392	0.000	1.444	0.026	***	0.329	0.000	0.424	0.017	***	0.657	0.041	0.487	0.043	***	0.568	0.011	0.490	0.035	***
25	0.228	0.000	0.512	0.023	***	0.223	0.000	0.415	0.018	***	0.669	0.098	0.497	0.035	***	0.557	0.031	0.493	0.035	***
26	0.320	0.000	0.832	0.033	***	0.337	0.000	0.671	0.027	***	0.635	0.006	0.488	0.035	***	0.526	0.011	0.488	0.035	***
27	0.338	0.000	1.092	0.040	***	0.417	0.000	0.674	0.027	***	0.439	0.042	0.495	0.039	***	0.551	0.049	0.488	0.034	***
28	0.471	0.000	1.382	0.041	***	0.372	0.000	0.869	0.031	***	0.568	0.025	0.496	0.041	***	0.716	0.000	0.499	0.035	***
29	0.286	0.000	0.914	0.039	***	0.311	0.000	0.611	0.025	***	0.593	0.008	0.493	0.037	***	0.530	0.026	0.494	0.034	***
30	0.060	0.000	0.383	0.017	***	0.108	0.000	0.192	0.008	***	0.733	0.026	0.503	0.036	***	0.646	0.005	0.481	0.036	***
31	0.271	0.000	0.784	0.034	***	0.304	0.000	0.654	0.032	***	0.625	0.006	0.493	0.042	***	0.599	0.009	0.490	0.039	***
Average	**0.222**	**0.000**	**0.714**	**0.025**		**0.277**	**0.000**	**0.499**	**0.021**		**0.608**	**0.023**	**0.494**	**0.037**		**0.564**	**0.014**	**0.489**	**0.034**	

*** = p-value < .001

Table B.3: **Results.** Offline Binary Classification Results.

Classifier	Window Size	F-Score Mean	F-Score 95% CI	Recall Mean	Recall 95% CI	Precision Mean	Precision 95% CI
Random Forest	1	75.94%	[74.82%, 77.05%]	75.10%	[73.96%, 76.24%]	79.58%	[78.35%, 80.82%]
	2	83.08%	[81.98%, 84.18%]	83.68%	[82.6%, 84.75%]	84.94%	[83.73%, 86.16%]
	5	90.66%	[89.73%, 91.6%]	90.12%	[89.05%, 91.18%]	93.11%	[92.16%, 94.07%]
	10	92.34%	[91.24%, 93.45%]	92.00%	[90.81%, 93.2%]	94.50%	[93.44%, 95.57%]
	30	94.37%	[93.31%, 95.42%]	93.51%	[92.35%, 94.66%]	97.35%	[96.61%, 98.08%]
	60	**95.00%**	[94%, 96%]	**94.03%**	[92.9%, 95.16%]	**98.36%**	[97.87%, 98.85%]
SGD	1	76.03%	[74.95%, 77.11%]	73.45%	[72.32%, 74.59%]	82.71%	[81.5%, 83.92%]
	2	83.85%	[82.71%, 84.99%]	83.37%	[82.27%, 84.47%]	86.44%	[85.14%, 87.74%]
	5	89.88%	[88.7%, 91.05%]	90.54%	[89.43%, 91.64%]	**90.66%**	[89.37%, 91.95%]
	10	**90.38%**	[89.17%, 91.59%]	92.33%	[91.21%, 93.45%]	90.10%	[88.77%, 91.43%]
	30	89.58%	[88.33%, 90.84%]	93.32%	[92.23%, 94.41%]	88.04%	[86.66%, 89.41%]
	60	88.13%	[86.88%, 89.38%]	**93.52%**	[92.46%, 94.58%]	85.75%	[84.35%, 87.15%]
MLP	1	77.07%	[76.01%, 78.12%]	78.04%	[77.02%, 79.06%]	77.60%	[76.43%, 78.77%]
	2	83.00%	[81.92%, 84.07%]	86.06%	[85.08%, 87.03%]	81.81%	[80.61%, 83%]
	5	89.76%	[88.73%, 90.8%]	91.72%	[90.91%, 92.54%]	89.29%	[88.12%, 90.45%]
	10	**91.94%**	[90.89%, 92.99%]	**92.88%**	[92.02%, 93.73%]	92.37%	[91.2%, 93.54%]
	30	91.85%	[90.65%, 93.04%]	92.53%	[91.39%, 93.68%]	92.49%	[91.24%, 93.74%]
	60	91.76%	[90.62%, 92.91%]	91.70%	[90.46%, 92.93%]	**93.62%**	[92.59%, 94.65%]
LinearSVC	1	77.19%	[76.15%, 78.23%]	76.67%	[75.61%, 77.73%]	79.47%	[78.31%, 80.63%]
	2	84.52%	[83.4%, 85.63%]	85.56%	[84.52%, 86.6%]	84.93%	[83.69%, 86.17%]
	5	90.10%	[88.97%, 91.22%]	91.19%	[90.17%, 92.21%]	90.40%	[89.17%, 91.63%]
	10	**91.92%**	[90.86%, 92.98%]	**93.30%**	[92.43%, 94.17%]	**92.00%**	[90.82%, 93.19%]
	30	90.66%	[89.42%, 91.9%]	93.18%	[92.04%, 94.33%]	90.01%	[88.68%, 91.33%]
	60	90.79%	[89.56%, 92.02%]	93.24%	[92.07%, 94.42%]	90.09%	[88.8%, 91.39%]

Table B.4: **Results.** Offline One-Class Classification Results.

Classifier	Window Size	F-Score Mean	F-Score 95% CI	Recall Mean	Recall 95% CI	Precision Mean	Precision 95% CI
Isolation Forest	1	3.08%	[2.84%, 3.32%]	**49.99%**	[49.99%, 49.99%]	4.44%	[3.31%, 5.56%]
	2	3.08%	[2.84%, 3.32%]	49.98%	[49.97%, 49.98%]	11.15%	[9.29%, 13.01%]
	5	3.08%	[2.28%, 3.88%]	49.92%	[49.87%, 49.97%]	23.51%	[15.72%, 31.3%]
	10	3.13%	[2.34%, 3.92%]	49.83%	[49.72%, 49.95%]	23.79%	[16.43%, 31.15%]
	30	3.93%	[3.68%, 4.19%]	48.96%	[48.59%, 49.34%]	37.51%	[35.49%, 39.52%]
	60	**7.80%**	[6.8%, 8.8%]	48.62%	[48.02%, 49.22%]	**44.10%**	[42.3%, 45.9%]
One-Class SVM (Kernel = RBF)	1	44.97%	[43.62%, 46.31%]	55.09%	[54.03%, 56.15%]	52.70%	[52.25%, 53.15%]
	2	45.36%	[40.15%, 50.56%]	53.69%	[50.31%, 57.07%]	53.82%	[50.89%, 56.76%]
	5	46.20%	[45.04%, 47.36%]	56.88%	[55.81%, 57.95%]	53.44%	[52.77%, 54.1%]
	10	51.64%	[47.56%, 55.72%]	58.07%	[55.18%, 60.96%]	57.37%	[52.68%, 62.06%]
	30	60.20%	[56.14%, 64.25%]	**60.37%**	[57.32%, 63.42%]	74.30%	[66.69%, 81.92%]
	60	**62.17%**	[58.23%, 66.12%]	59.64%	[56.7%, 62.59%]	**84.10%**	[76.64%, 91.56%]
One-Class SVM (Kernel = Linear)	1	60.48%	[59.49%, 61.48%]	64.11%	[63.26%, 64.97%]	62.38%	[61.33%, 63.42%]
	2	60.92%	[59.79%, 62.04%]	63.55%	[62.63%, 64.48%]	65.07%	[63.7%, 66.44%]
	5	63.74%	[62.44%, 65.05%]	63.70%	[62.69%, 64.71%]	71.84%	[70.32%, 73.36%]
	10	67.48%	[63.86%, 71.1%]	**64.88%**	[62.05%, 67.71%]	80.29%	[75.04%, 85.54%]
	30	**68.00%**	[64.25%, 71.75%]	64.64%	[61.62%, 67.66%]	**83.31%**	[77.41%, 89.2%]
	60	65.95%	[62.1%, 69.8%]	63.73%	[60.63%, 66.83%]	82.09%	[75.32%, 88.87%]

Table B.5: **Results.** Online Binary Classification Results.

Classifier	Window Size	F-Score Mean	F-Score 95% CI	Recall Mean	Recall 95% CI	Precision Mean	Precision 95% CI
Adaptive Random Forest (No Drift Detection)	1	73.39%	[66.46%, 80.33%]	63.23%	[54.63%, 71.83%]	94.56%	[93.08%, 96.05%]
	2	81.66%	[75.24%, 88.07%]	73.11%	[64.97%, 81.26%]	97.96%	[97.26%, 98.65%]
	5	93.16%	[90.11%, 96.2%]	88.36%	[83.71%, 93.01%]	99.60%	[99.26%, 99.95%]
	10	97.07%	[95.4%, 98.73%]	94.70%	[91.82%, 97.58%]	99.92%	[99.88%, 99.96%]
	30	99.60%	[99.37%, 99.84%]	99.25%	[98.8%, 99.7%]	99.97%	[99.95%, 99.99%]
	60	**99.90%**	[99.81%, 99.98%]	**99.81%**	[99.65%, 99.97%]	**99.99%**	[99.97%, 100%]
SGD	1	85.90%	[84.26%, 87.54%]	80.76%	[78.86%, 82.65%]	93.64%	[92.92%, 94.35%]
	2	93.98%	[92.9%, 95.07%]	91.45%	[90.11%, 92.79%]	97.29%	[96.77%, 97.82%]
	5	98.45%	[98.07%, 98.83%]	97.80%	[97.36%, 98.25%]	99.13%	[98.81%, 99.44%]
	10	99.28%	[99.15%, 99.42%]	**98.95%**	[98.79%, 99.11%]	99.63%	[99.51%, 99.74%]
	30	99.22%	[99.02%, 99.43%]	98.69%	[98.35%, 99.04%]	99.81%	[99.77%, 99.84%]
	60	**99.29%**	[98.99%, 99.59%]	98.82%	[98.33%, 99.32%]	**99.87%**	[99.84%, 99.9%]
Perceptron	1	87.83%	[86.92%, 88.74%]	87.43%	[86.5%, 88.36%]	88.23%	[87.34%, 89.13%]
	2	94.50%	[93.92%, 95.08%]	94.53%	[93.95%, 95.11%]	94.48%	[93.9%, 95.06%]
	5	98.12%	[97.84%, 98.4%]	98.16%	[97.88%, 98.44%]	98.08%	[97.8%, 98.37%]
	10	99.17%	[99.07%, 99.27%]	99.22%	[99.12%, 99.31%]	99.12%	[99.01%, 99.23%]
	30	99.69%	[99.65%, 99.74%]	99.72%	[99.68%, 99.77%]	99.66%	[99.62%, 99.7%]
	60	**99.80%**	[99.77%, 99.84%]	**99.85%**	[99.81%, 99.88%]	**99.76%**	[99.73%, 99.8%]

APPENDIX C – APPENDIX FOR THE "THE SPICE MUST FLOW: CHALLENGES OF MODELLING SECURITY DATA AS STREAMS" PAPER

C.1 PREQUENTIAL AUC-PR

The Algorithm 4 presents the efficient implementation of SPICE's Prequential AUC-PR, as described in Subsection 5.1.6.

Algorithm 4 SPICE Prequential AUC-PR

Require: stream S of scored samples, window size d

$W \leftarrow \emptyset$ ▷ Sliding window
$p \leftarrow 0$ ▷ Number of positives
$n \leftarrow 0$ ▷ Number of negatives
while S has samples **do**
    $s \leftarrow S$ next sample
    Add s to W
    $tree.add(s)$ ▷ Add new score
    $updateTPsAndFPs()$ ▷ Update TPs and FPs of lower scores on tree
    if true(s) **then**
        $p \leftarrow p + 1$
    else
        $n \leftarrow n + 1$
    end if
    $AUC \leftarrow 0$
    if W length is equal to d **then**
        while W has samples **do**
            $w \leftarrow W$ next sample
            $tp \leftarrow w$ true positives
            $fp \leftarrow w$ false positives
            if $tp + fp = 0$ **then**
                continue
            end if
            Calculate recall and precision
            Append recall and precision to points list
        end while
        Reverse sort points list by recall
        Append point (0, 1) to points list
        Insert point (1, $\frac{p}{p+n}$) at the beginning of points list
        $AUC \leftarrow integrate(points)$
        $f \leftarrow W$ first sample
        Remove f from W
        $tree.remove(f)$
        $updateTPsAndFPs()$
        if true(f) **then**
            $p \leftarrow p - 1$
        else
            $n \leftarrow n - 1$
        end if
    end if
end while