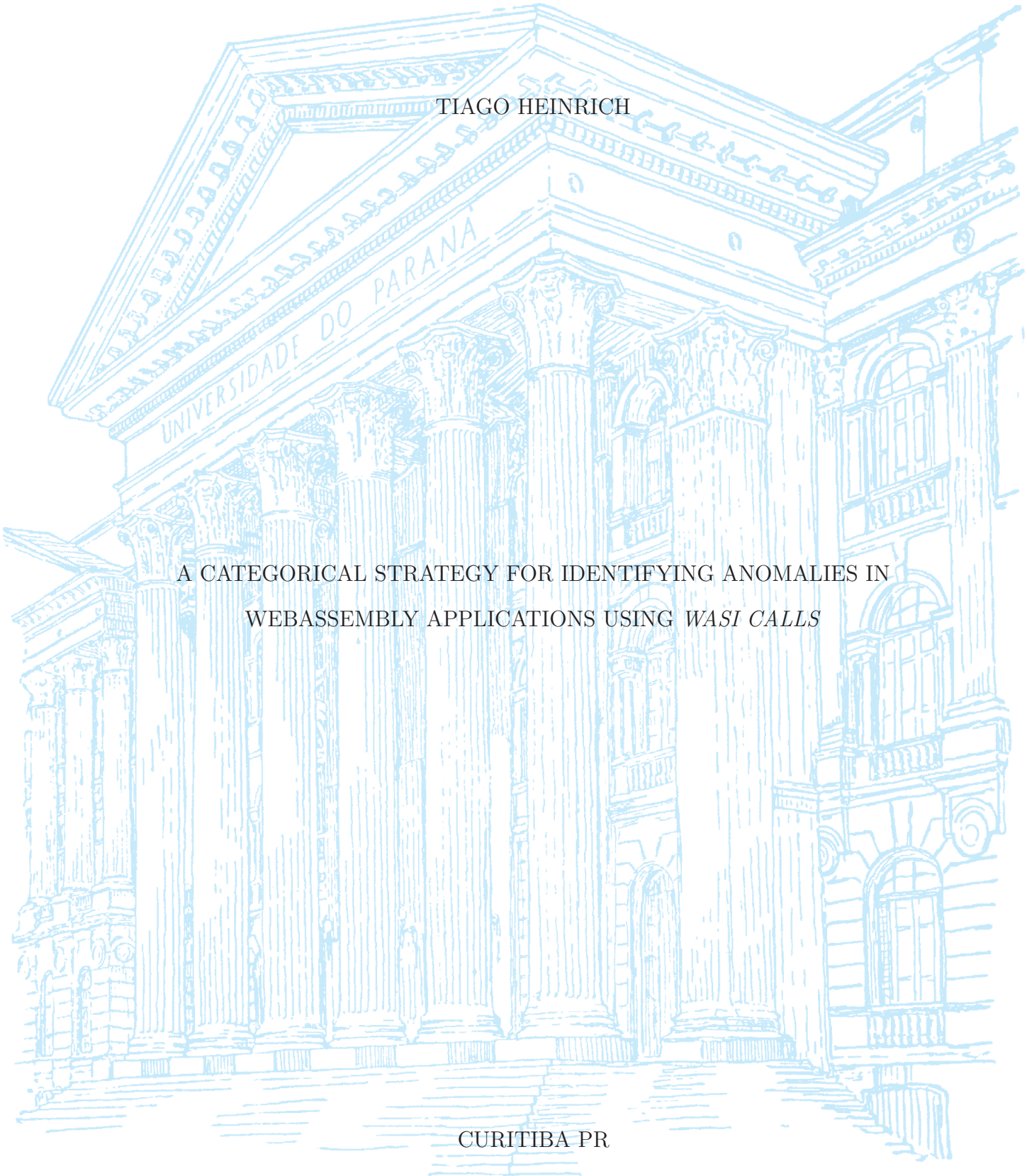
UNIVERSIDADE FEDERAL DO PARANÁ

TIAGO HEINRICH



A CATEGORICAL STRATEGY FOR IDENTIFYING ANOMALIES IN WEBASSEMBLY APPLICATIONS USING *WASI CALLS*

CURITIBA PR

2023

TIAGO HEINRICH

A CATEGORICAL STRATEGY FOR IDENTIFYING ANOMALIES IN WEBASSEMBLY APPLICATIONS USING *WASI CALLS*

Documento apresentado como requisito parcial para o exame de qualificação de Doutorado, no Programa de Pós-Graduação em Informática, setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Computação.*

Orientador: Carlos Alberto Maziero.

Coorientador: Rafael Rodrigues Obelheiro.

CURITIBA PR

2023

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

Heinrich, Tiago
A categorical strategy for identifying anomalies in webassembly applications using WASI calls / Tiago Heinrich. – Curitiba, 2023.
1 recurso on-line : PDF.

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

Orientador: Carlos Alberto Maziero
Coorientador: Rafael Rodrigues Obelheiro

1. Aprendizado do computador. 2. Interfaces (Computador). 3. WebAssembly. I. Universidade Federal do Paraná. II. Programa de Pós-Graduação em Informática. III. Maziero, Carlos Alberto. IV. Obelheiro, Rafael Rodrigues. 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 **TIAGO HEINRICH** intitulada: **A Categorical Strategy for Identifying Anomalies in WebAssembly Applications using WASI Calls**, sob orientação do Prof. Dr. CARLOS ALBERTO MAZIERO, 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, 06 de Dezembro de 2023.

Assinatura Eletrônica
07/12/2023 07:04:29.0
CARLOS ALBERTO MAZIERO
Presidente da Banca Examinadora

Assinatura Eletrônica
07/12/2023 09:10:54.0
EDUARDO LUZEIRO FEITOSA
Avaliador Externo (UNIVERSIDADE FEDERAL DO AMAZONAS)

Assinatura Eletrônica
20/12/2023 11:37:09.0
MAURO SERGIO PEREIRA FONSECA
Avaliador Externo (UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ)

Assinatura Eletrônica
12/12/2023 22:28:28.0
LUIZ CARLOS PESSOA ALBINI
Avaliador Interno (UNIVERSIDADE FEDERAL DO PARANÁ)

Assinatura Eletrônica
08/12/2023 17:28:18.0
RAFAEL RODRIGUES OBELHEIRO
Coorientador(a) (UNIVERSIDADE DO ESTADO DE SANTA CATARINA)

Aos meus pais.

ACKNOWLEDGEMENTS

The journey for a PhD is not easy and during these years many people were important. This is recognition for some people who were essential during this journey.

I will start by thanking my parents, Catia Arlene Hoeller Heinrich and Arlindo Heinrich, for all the support, teachings, guidance, and help, I would not be where I am without you. I would like to thank my brother Rodrigo Heinrich, who was always present during these years of study and shared this journey with me, both on good and bad days.

I could not have undertaken this journey without the guidance of my advisors, which not only advise me in my research but also help me with many important decisions. Prof. Dr. Carlos Alberto Maziero your guidance was essential during these four years of research, I am grateful for the teachings, corrections, and feedback that in some cases occurred during a Saturday and for the disposition in helping me in all the challenges that I embraced during this period. Prof. Dr. Rafael Rodrigues Obelheiro I would like to thank you for all the partnership over these seven years, for all the teachings, for all the time invested, in all the meetings that lasted hours, and for all the advice in essential moments. I learned a lot from you and I certainly wouldn't be where I am without your help.

I am also grateful to Profa. Dra. Rebeca Schroeder Freitas for teaching and work carried out during my graduation at Santa Catarina State University (UDESC). To professors Dr. Guilherme Piêgas Koslovski, Dr. Mauricio Aronne Pillon, and Dr. Charles Christian Miers for their support throughout this journey and teachings over the years.

I need to thank the Universidade Federal do Paraná (UFPR) and the Graduate Program in Informatics (PPGinf) for the opportunity I had during all these years. The Network, Distributed Systems, and Security Lab (LarSis) friend group who were always open for discussion, which was very helpful in the construction of this work. So, thanks Amanda Benites Viescinski, Marcus Botacin, Fabrício Ceschin, Vinícius Fülber Garcia, José Wilson Vieira Flauzino, Florian Hantke, and Rafael Gomes de Castro.

I need to thank the UDESC for having contributed to the laboratories which were used for long hours at dawn to finish work and experiments, especially the friends of the Networks and Distributed Applications Group (GRADIS) and Laboratory of Parallel and Distributed Processing (LabP2D) research group. This was a key factor during my bachelor's in computer science and master's in applied computing. A special thanks to Diogo Bortolini, for the attention to the honeypots we have deployed over the years and the BRUTE Competitive Programming group where I have made close friends over the years.

I am also grateful to my friends Gustavo Diel and Leonardo Rosa Rodrigues who were always present, collaborating with the work and helping with the experiments carried out, but mainly for always being by my side without hesitating to help.

I would like to thank Newton Carlos Will for his partnerships in publications and unexpected accommodations over the years. Eloiza Rossetto dos Santos for all the ideas, discussions about machine learning and for having welcomed me when I started this new stage at UFPR. I would also like to thank Beatriz Michelson Reichert for all the help reviewing papers, life advice, and TV show recommendations.

I am thankful for the friendships made at events, especially Victor Le Pochat, Lars Prehn, and Casey Deccio who always take the time to discuss my research. I would also like to thank Prof. Dr. Marinho Barcellos for all the opportunities.

Finally, I would like to thank CAPES for helping me with funding to make this research possible. I also would like to thank Brazilian Symposium on Information and Computational Systems Security (SBSeg), for fomenting research in Brazil. The Passive and Active Measurement (PAM) and Internet Measurement Conference (IMC) committees for the opportunity to attend the conference and discuss ideas over the years.

"In God we trust. All others must bring data" – W. Edwards Deming

RESUMO

Na última década os navegadores Web se tornaram um dos recursos indispensável para usuários acessarem a Internet. Novos recursos e funcionalidades foram introduzidos, visando trazer mais praticidade para os usuários. O *WebAssembly* é um formato que foi criado para a execução de conteúdo dinâmico em navegadores web. Sendo um formato binário e portátil que vem sendo bem aceito devido a oferecer ganhos de desempenho em comparação com as soluções existentes. Atualmente o *WebAssembly* não está mais limitado ao uso em aplicações Web, sendo adotado por um conjunto de áreas que conseguem explorar características específicas do formato para algum tipo de ganho. Novos recursos que visam trazer praticidade para os usuários também geram novas superfícies de ataque que podem ser exploradas para comprometer a segurança do ambiente. Considerando estratégias de detecção de intrusões em um navegador Web, as estratégias de detecção estão limitadas à identificação de ataques que exploram *JavaScript (JS)* ou extensões. A detecção de anomalias em aplicações *WebAssembly* é pouco explorada na literatura. Trabalhos existentes têm foco na segurança do formato, adição de recursos e otimização do compilador e não na detecção de aplicações maliciosas que poderiam estar sendo executadas no ambiente. Visando esta lacuna na literatura, este trabalho visa propor uma estratégia de detecção de intrusão baseada em anomalias utilizando dados categóricos para aplicações *WebAssembly*. Através do uso de modelos de aprendizado de máquina, será realizada a classificação dos dados e detecção de anomalias. O recurso observado para a realização deste processo são as chamadas *WebAssembly System Interface (WASI)*, que são chamadas para o suporte em tempo de execução *WebAssembly* análogas às chamadas de sistema em um sistema operacional. As chamadas serão modeladas usando dados categóricos, uma abordagem que vem sendo usada com sucesso na identificação de anomalias em outros contexto e que, apesar de promissora, ainda é pouco empregada em detecção de intrusões. A representação dos dados considerando a categorização permite a adição de características que não estão diretamente associadas com as chamadas WASI. Por fim, será possível discutir como estas representações podem ser melhor utilizadas no contexto de segurança computacional.

Palavras-chave: Detecção de Anomalias, WebAssembly, Aprendizado de Máquina e Dados Categóricos.

ABSTRACT

In the last decade, Web browsers have become one of the essential resources for users to access the Internet. New features and functionalities were introduced, aiming to bring more practicality to users. *WebAssembly* is a format that was created for executing dynamic content in web browsers. Being a binary and portable format that has been well accepted due to offering performance gains compared to existing solutions. Currently *WebAssembly* is no longer limited to use in Web applications, being adopted by a set of areas that are able to exploit specific characteristics of the format for some type of gain. New features that aim to bring convenience to users also generate new attack surfaces that can be exploited to compromise the security of the environment. Considering intrusion detection strategies in a Web browser, detection strategies are limited to identifying attacks that exploit *JS* or extensions. Anomaly detection in *WebAssembly* applications is little explored in the literature. Existing work focuses on format security, adding features and optimizing the compiler and not on detecting malicious applications that could be running in the environment. Aiming at this gap in the literature, this work aims to propose an intrusion detection strategy based on anomalies using categorical data for *WebAssembly* applications. Through the use of machine learning models, data classification and anomaly detection will be carried out. The resource observed for carrying out this process are *WASI* calls, which are calls to runtime support *WebAssembly* analogous to system calls in an operating system. Calls will be modeled using categorical data, an approach that has been used successfully in identifying anomalies in other contexts and which, despite being promising, is still little used in intrusion detection. Data representation considering categorization allows the addition of features that are not directly associated with WASI calls. Finally, it will be possible to discuss how these representations can be better used in the context of computational security.

Keywords: Anomaly Detection, WebAssembly, Machine Learning and Categorical Data.

LIST OF FIGURES

2.1 Generic architecture of an IDS.. 21
2.2 Architecture of Firefox web browser.. 24
2.3 Representation of DOM. 24
2.4 *System calls* flow on Linux. 26
2.5 Steps of the ML based attack detection.. 29
2.6 Types of Measurements and relation with Categorical data. 32

3.1 WebAssembly design. 35
3.2 WASI Software Architecture. 37

5.1 Overview of our proposal for intrusion detection using WASI calls. 57

LIST OF TABLES

2.1 Intrusion detection strategies. 22

3.1 Structure of the WebAssembly binary (A WebAssembly Module). 35

4.1 Overview of WebAssembly Flaws and Vulnerabilities used in Attacks 45
4.2 WebAssembly Datasets 52

5.1 Classification according to threat level. 58
5.2 Classification according to functionality. 59
5.3 WASI Calls classification. 59

6.1 Performance of the models for the offline strategy. 64
6.2 Performance of the models for the real-time strategy (non-overlapping window). 66
6.3 Performance of the models for the real-time strategy (overlapping window). 67

A.1 List of Attacks presented at the Dataset. 84
C.1 System Calls and WASI Calls classification. 89

LIST OF LISTINGS

2.1 Function parameter example. 27
2.2 Example of a system call trace. 27

3.1 WAT module (Example from Wasmtime (BytecodeAlliance, 2021a)). . . . 36

6.1 Trace of a Wasm application. 62
6.2 WASI calls extracted. 64
6.3 Classification results. 64

LIST OF ACRONYMS

ABE Application Boundaries Enforcer

API Application Programming Interface

APK Android Application Pack

ASLR Address Space Layout Randomization

AV Antivirus

AVF Attribute Value Frequency

BOBE Break-Once-Break-Everywhere

CFI Control Flow Integrity

CIDR Classes Inter Domain Routing

CLI Command-Line Interface

CPG Code Property Graph

CROW Code Randomization of WebAssembly

CSRF Cross-site request forgery

CSS Cascading Style Sheets

CT-Wasm Constant-Time WebAssembly

CWE Common Weakness Enumeration

DEP Data Execution Prevention

DOM Document Object Model

DoS Denial of Service

DTN Disruption-Tolerant Networking

ELF Executable and Linkable Format

EVM Ethereum Virtual Machine

GC Garbage Collection

GRADIS Networks and Distributed Applications Group

HIDS Host-based Intrusion Detection System

HMMs Hidden Markov Models

HTML HyperText Markup Language

IDE Integrated Development Environment

IDS Intrusion Detection System

IDSs Intrusion Detection Systems

IFC Information-Flow Control

IMC Internet Measurement Conference

IoT Internet of Things

IP Internet Protocol

IPC Inter-Process Communication

IR Intermediate Representation

ISA Instruction Set Architecture

JIT Just-In-Time

JS JavaScript

JVM Java Virtual Machine

KNN K-Nearest Neighbors

LabP2D Laboratory of Parallel and Distributed Processing

LarSis Network, Distributed Systems, and Security Lab

LIFO Last-In, First-Out

MEWE Multi-variant Execution for WebAssembly

ML Machine Learning

MLP Multilayer Perceptron

MSDD Multi-Stream Dependency Detection

MSWasm Memory-safe WebAssembly

MVP Minimum Viable Product

NIDS Network Intrusion Detection System

OOP Object-Oriented Programming

OS Operating System

PAM Passive and Active Measurement

PC Personal Computer

PE Portable Executable

PPGinf Graduate Program in Informatics

PSI Private Set Intersection

RDF Resource Description Framework

REMUS REference Monitor for UNIX Systems

RQ Research Question

RQs Research Questions

SBSeg Brazilian Symposium on Information and Computational Systems Security

SC Selection Criteria

SFI Software-fault Isolation

SGD Stochastic Gradient Descent

SGX Software Guard Extensions

SOP Same Origin Policy

TC Trusted Computing

TEE Trusted Execution Environment

UDESC Santa Catarina State University

UFPR Universidade Federal do Paraná

VM Virtual Machine

WASI WebAssembly System Interface

WASP WebAssembly Symbolic Processor

WAT WebAssembly Text

WSARE What's strange about recent events

WWW World Wide Web

XML Extensible Markup Language

XSRF Cross-site reference forgery

XSS Cross Site Scripting

ZKSNARK Zero-Knowledge Succinct Non-Interactive Argument of Knowledge

CONTENTS

1 INTRODUCTION 17
1.1 CONTEXTUALIZATION 17
1.2 MOTIVATION 19
1.3 RESEARCH GOALS 19
1.4 OUTLINE 20
2 THEORETICAL FOUNDATION 21
2.1 INTRUSION DETECTION 21
2.2 BROWSER SECURITY 23
2.2.1 Browser Architecture 23
2.2.2 Execution Environment 23
2.2.3 Browser Security 25
2.3 SYSTEM CALLS 25
2.3.1 Example of system call use 27
2.3.2 Intrusion Detection using System Calls 28
2.4 MACHINE LEARNING 29
2.4.1 Application for Cybersecurity 29
2.4.2 Evaluation Metrics 30
2.4.3 Categorical Data Representation 31
2.5 CONCLUDING REMARKS 33
3 WEBASSEMBLY 34
3.1 OVERVIEW 34
3.2 DEVELOPMENT AND USE 36
3.2.1 Security Model 37
3.2.2 WASI 37
3.3 CURRENT STATE OF WEBASSEMBLY 38
3.4 CONSIDERATIONS 38
4 WEBASSEMBLY SECURITY LITERATURE REVIEW 39
4.1 METHODS 39
4.2 WEBASSEMBLY SECURITY RESEARCH 40
4.2.1 WebAssembly as an Attack Vector 40
4.2.2 Improving WebAssembly Security 44
4.2.3 Analysis and Protection for WebAssembly 48
4.2.4 Proposals for WebAssembly 49
4.2.5 General Applications 50

4.2.6 Datasets available . . . 51
4.2.7 Wasm Current Limitations . . . 53
4.2.8 Open Research Topics . . . 53
4.2.9 Current developments in WebAssembly . . . 53
4.3 DISCUSSION OF FINDINGS . . . 54
5 AN ANOMALY DETECTION SOLUTION FOR WEBASSEMBLY . . . 56
5.1 PROBLEM DEFINITION . . . 56
5.2 ANOMALY DETECTION SOLUTION . . . 56
5.3 DATA CATEGORIZATION . . . 57
5.4 DEVELOPMENT AND TEST . . . 60
5.5 WASI CALLS VS SYSTEM CALLS . . . 60
5.6 CONSIDERATIONS . . . 61
6 EVALUATION OF WASI CALLS FOR ANOMALY DETECTION . . . 62
6.1 EVALUATION STRATEGY . . . 62
6.1.1 Dataset . . . 62
6.1.2 Machine Learning Approach . . . 63
6.1.3 Data Preprocessing . . . 64
6.2 OFFLINE DETECTION . . . 64
6.3 ONLINE DETECTION . . . 65
6.4 DISCUSSION OF RESULTS . . . 67
7 CONCLUSION . . . 69
REFERENCES . . . 70
APPENDIX A – LIST OF ATTACKS PRESENTED IN THE DATASET . . . 84
APPENDIX B – USING CATEGORICAL DATA FOR CYBERSECURITY SOLUTIONS . . . 85
B.1 RESEARCH USING CATEGORICAL DATA . . . 85
B.1.1 Categorical data for Anomaly Detection . . . 85
B.1.2 Cybersecurity and Categorical Data . . . 87
B.2 CONSIDERATIONS . . . 87
APPENDIX C – SYSTEM CALLS AND WASI CALLS CLASSIFICATION . . . 88

1 INTRODUCTION

This thesis describes and evaluates an approach for anomaly-based intrusion detection in WebAssembly programs, using machine learning strategies on categorical data extracted from WebAssembly executions. This chapter presents a brief contextualization of this research work, the situation of the environment to be explored, together with the problem to be solved. Section 1.1 presents the contextualization of the area. Section 1.2 describes the motivation of this thesis, and Section 1.3 presents the research hypothesis and goals, with the description of the objective for anomaly detection in WebAssembly through the use of machine learning strategies on categorical data.

1.1 CONTEXTUALIZATION

The need for a tool to enable browsing and retrieval of content on World Wide Web (WWW) was solved in 1990 with the first web browser (Grosskurth and Godfrey, 2006). This application is responsible for retrieving the content of a web page when making a request to a server and presenting this information in a user-friendly way. Currently these applications are not limited to *desktops* (48.8%) having a relevant presence when considering *mobile* (48.6%) and *tablet* (2.5%) devices, with 4.9 billion users in Q3 2020 (StatCounter, 2020; Internet World Stats, 2020).

The wide use of web browsers contributed to the development of numerous features to bring convenience to users and developers. Feature-rich browsers constitute sizable attack surfaces, prompting the need for security measures such as isolation between the different pages accessed, treating each page as a separate application (Grier et al., 2008; Reis et al., 2019). Even so, problems still exist, such as malicious extensions, application limitations, and different interpretations of the security policy of the web browser (Ter Louw et al., 2008; Karami et al., 2020).

While the early web was based on static content, active/dynamic content in which the client-side (browser) runs a script to update content in the page is now pervasive (Stock et al., 2017). An important aspect of the web environment is the programming languages used for the development of applications and services, as they define and limit the operations to be carried out, and consequently the security policies. Historically, three languages have been proposed with a focus on client-side web development, these being: Java applets, Flash, and JavaScript (JS) (Golsch, 2019). The Java applet and Flash have been discontinued for web applications due to performance and security issues, respectively. JS then earned exclusivity on web browsers, this factor was further aided by the growth that the Internet has had in the last years (Feldman, 2019).

JS is the major player in the Web environment and is a language easy enough for developers to build any type of small/medium application. However, when the front-end requires a more complex application that needs to be efficient and secure, some problems related to JS appear. Two main problems found in JS are the lack of types and runtime interpreter overhead (due to the lack of a binary format). These problems were identified in 2015, leading major developer organizations to propose WebAssembly as an alternative to overcome the limitations found in JS.

WebAssembly had its first version released in 2017 with the goals of safeness, speed, and portable semantics (Golsch, 2019). It is important to note that WebAssembly is not

intended to be a replacement for JS, but rather to improve specific issues. WebAssembly allows developers to use a wider range of programming languages in the web environment, being faster to download, compile, and run (Battagline, 2021).

WebAssembly is a portable binary code format designed for safe and efficient execution, with a compact representation (Rossberg, 2022). It is typically used as an executable code format generated by compilers for high-level languages such as C, C++, Go, and Rust. In this way, WebAssembly allows applications and libraries developed in these languages to be executed in Web browsers (Wasm's initial target) or in native environments (Hoffman, 2019). WebAssembly bytecode typically runs inside a sandbox called WebAssembly runtime. This design choice allows applications to run on a wider range of platforms, reduces the size of messages to be exchanged with the server to retrieve content, and provides greater security for the client environment running the content.

Attackers may try to exploit WebAssembly features and vulnerabilities to carry out attacks with the intent of compromising the web browser and/or the underlying host. In a web browsing session, it is not uncommon for a user to have several tabs or windows open, some of which may hold sensitive data (such as authentication tokens, credit card and other financial information, or personally identifiable information). A successful attack against the browser may put such information at risk, and even enable the attacker to access external resources (such as local files) visible to the browser (Guha et al., 2011; Šilić et al., 2010).

In the client-side critical information is stored and handled. In the cloud environment, the risk is associated with applications running in the same environment, where one vulnerable WebAssembly application could risk the security of the environment, putting the cloud environment at risk.

Intrusion Detection System (IDS) is a strategy used to monitor and identify malicious activities in an environment. The first approach exploited system logs to identify unwanted activity (Anderson, 1972, 1980). Over the years, two sets of intrusion detection strategies were defined, having wide use to detect malicious activities (Liu et al., 2018), namely: (1) *signature-based (or misuse-based) detection*, which compares events observed in the system to a database with information on known attacks; and (2) *anomaly-based detection*, which seeks to characterize a default behavior for the system or applications, treating inconsistencies (anomalies) as unexpected behavior or attacks to the system.

Anomaly-based detection is a common approach for intrusion detection in environments such as cloud platforms, virtualized environments, Android systems (Liao et al., 2013; Castanhel et al., 2021; Lemos et al., 2023). However, so far it has been little used for detecting malicious behavior in WebAssembly code, with most proposals focusing on the static evaluation of binaries. The flexiblity of the WebAssembly environment (where an application can be executed inside a web browser, on the cloud, or on a local host) and its security implications are also largely unexplored.

A strategy that has gained popularity in recent years is the use of Machine Learning (ML) templates to identify attacks. These techniques can be used as an anomaly detection strategy, as the method learns the behavior of the application through a dataset. Such methods offer some advantages in comparison with other strategies (1) after the process of learning, classifying an entry is quite efficient, (2) the evolution of the behavior can be considered, and (3) if only one type of behavior is known (supposing that only one good behavior of the applications is know) the strategy still can be applied.

The key factor behind any anomaly detection method consists of the data use and representation applied. A well-fitted representation of the data will directly impact

the capacity of the models to learn and understand the sample patterns. Categorization is one option of data representation that allows the detection of anomalies and outliers through the use of categories. It is a recurring approach in other fields of research, such as economics and statistics (Powers and Xie, 2010).

The use of categorical data representation for cybersecurity solutions is limited, and previous proposals presented good results using this type of data representation (Bernaschi et al., 2002). Therefore, the study of this type of representation can be interesting for cybersecurity strategies.

1.2 MOTIVATION

WebAssembly started as a resource aimed at solving the problem of limitations and performance of languages for web development, offering a way to develop web applications using languages like C++, Python, Rust, and Go. Thus developers are not limited to one language like JS to develop web applications. However, WebAssembly also expanded into a "sandboxed environment", which aims to improve the cybersecurity of the runtime environment, expanding the use of the format outside of the Web browser (Battagline, 2021).

Considering the adoption and expectations for WebAssembly, targeting cybersecurity in this new context is essential. Strategies for identifying anomalies using machine learning have become popular in recent years due to the high rate of identification and adaptation to new attacks. With this motivation, the focus of the research consists of the proposal of a new strategy for identifying anomalies in WebAssembly.

In the literature, approaches focused on cybersecurity in WebAssembly are limited to the development of new applications in a safe way, or dealing with problems present in WebAssembly compilers. The identification of attacks performed by WebAssembly applications is not yet an active research topic. An interesting perspective for anomaly detection with the WebAssembly consists of the WebAssembly sandbox. Interactions from the sandbox with the environment are required to use an Application Programming Interface (API) that generates WebAssembly System Interface (WASI) calls to access system resources. The monitoring of this API brings opportunities to better understand the behavior of applications and allows the identification of threats.

The categorization of WASI calls provides an opportunity to introduce extra information for the algorithms used in the anomaly detection process. Through categorization, extra operation and functionality information can be presented during the identification process. This information allows a better understanding of the process of executing WebAssembly applications. The use of categorical data also allows us to better understand how the use of categorical information can improve cybersecurity solutions.

Considering proposals for anomaly detection and outlier detection, the use of categorical data representations appears to be a good fit for the problem to be explored. As categorical data representation is quite popular in other fields of research and machine learning solutions can take advantage of this representation, we intend to assess the use of categorical data representation for intrusion detection in WebAssembly applications.

1.3 RESEARCH GOALS

WebAssembly has gained popularity due to its performance for web applications and the sandbox environment. Considering the cybersecurity side of WebAssembly applications,

strategies for anomaly detection are limited. The available strategies focus on identifying compiler threats, static evaluation of binaries, format flaws, or external flaws that do not directly depend on WebAssembly.

A WebAssembly application uses WASI calls to request access to Operating System (OS) resources, providing a means for monitoring running applications. This feature allows Host-based Intrusion Detection System (HIDS) strategies to use this information offered between the application's interactions with the environment to identify anomalies that could be attacks or unexpected behavior.

Thus, the main goal of the work is to propose and evaluate a strategy for anomaly detection in WebAssembly applications. The hypothesis is that it is possible to differentiate between benign and malicious WebAssembly applications based on the WASI calls they invoke.

To identify such behaviors, the categorical representation of collected data appears to be promising for identifying outliers and anomalies. Categorical approaches define numerous strategies to find such behaviors, with wide use due to the type of data found when dealing with this kind of problem. In the social sciences, these models are used due to the type of data, and problems such as fraud identification exploit these strategies. In computer security, few cases explore categorical strategies despite their wide use in other areas.

By approaching WASI calls in a categorical way, a new interpretation of the data can be performed, in addition to allowing the application of outlier identification models aimed at categorical data. The use of categorical strategies also allows a new layer of abstraction for the data. This level of abstraction allows calls like *fread* and *read* to be classified as similar, not creating an additional level of complexity on the dataset. The identification process with ML models also takes advantage of this representation format, since information not present directly in the dataset can be added.

To achieve the proposal, the following specific objectives were defined:

- The construction of a dataset with samples that represent WebAssembly applications;
- Evaluate the use of ML models for anomaly detection;
- Verify the effectiveness of using WASI calls to identify attacks in WebAssembly; and
- Evaluate the effectiveness of the use of a categorical data representation in cybersecurity.

1.4 OUTLINE

This document is organized as follows. Chapter 2 presents the theoretical foundation. Chapter 3 presents a contextualization of WebAssembly concept. Chapter 4 discusses how WebAssembly is being used, its cybersecurity problems, and its limitations. Chapter 5 details the proposal for an anomaly detection solution for WebAssembly. Chapter 6 presents the evaluation the proposal. Finally, we conclude our work in Chapter 7.

2 THEORETICAL FOUNDATION

This chapter presents the theoretical foundation for understanding this work. Section 2.1 reviews Intrusion Detection System (IDS). Section 2.2 presents an overview of web browser security. Section 2.3 describes system calls and exemplifies their functionality. Section 2.4 highlights machine learning concepts, focusing on the strategies used in this work.

2.1 INTRUSION DETECTION

An intrusion is a violation, misuse, or exploitation of a policy for the purpose of obtaining some kind of benefit. This can be performed by an intruder either remotely or locally. In order to protect environments and users, intrusion identification strategies were developed (Stallings et al., 2012). An IDS is a security mechanism with the purpose of monitoring networks, hosts, and/or applications looking for signs of attacks or intrusions. When an indication of an attack is observed, an alert is issued for further analysis, and in some cases the system can be programmed to take active countermeasures, like adjusting the firewall to stop the attack.

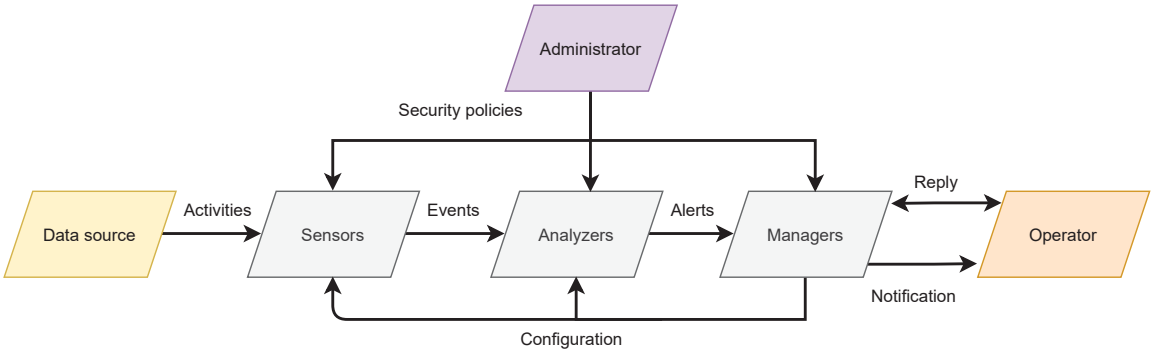


Figure 2.1: Generic architecture of an IDS.
Source: (Erlinger and Wood, 2007).

Figure 2.1 presents a generic architecture of an IDS, which comprises *sensors*, *analyzers*, and *managers*. The *sensors* are responsible for observing system activity from *data sources* (such as the network or running applications) and extracting events of interest which are sent to the *analyzers*, which will issue alerts for characteristics violating the security policies. The generated alerts are received by the *managers*, which will decide what action to take and notify the *operator* if the IDS components follow security policies defined by an administrator.

Ideally, the analysis of events of interest results in a true positive, when an alert is generated for a malicious event, or in a true negative, when an alert is not generated for a benign event. However, an IDS is subject to error during the detection process. This factor is commonly due to the complexity of the systems and operations being performed. Thus, the analysis of events can also result in a false positive, when an alert is generated for a benign event, or in a false negative, when an alert is not generated for a malicious event (or for other malicious behavior that the sensors fail to capture correctly). In this

way, the constant updating and study of new strategies contributes to the improvement of the identification accuracy.

An IDS tries to distinguish different behaviors and patterns in order to identify unwanted actions. Two strategies are popularly used to identify threats (Lam, 2005). *Signature-based* intrusion detection uses a database of known attack samples to identify unwanted behavior. These samples are known as signatures and are intended to store information that represent threats or known attack behavior. On the other hand, an *anomaly-based* IDS uses statistical models to define what would be a normal/usual activity in the system/environment. Thus, an anomaly portrays some activity that deviates from the previously defined normal model (Debar et al., 1999; Yassin et al., 2013).

It is important to note that the models are not limited to just these two strategies. Stateful protocol analysis that can trace a protocol state to represent the operations to be performed, and hybrid approaches that use mixed strategies, are also explored in the literature. Table 2.1 briefly summarizes the four strategies presented.

Signature-based	*Anomaly-based*	*Hybrid*	*Stateful protocol analysis*
Pros			
• Simple and efficient method • Detailed analysis	• Effectiveness to identify unknown attacks • Less dependency on the system type	• Explores more than one method for better efficiency • More complete representations in relation to other strategies	• Better understanding of states • Distinguish unexpected sequences of commands
Cons			
• Ineffective for identifying new or unknown attacks • Little understanding of states and protocols • Maintaining and updating signatures is difficult	• Assessment limited to training dataset • Difficult to work with large and real-time datasets	• Greater limitations due to model specialization	• Resource consumption to evaluate data • Unable to inspect similar attacks

Table 2.1: Intrusion detection strategies.
Source: Based on (Liao et al., 2013).

Intrusion detection systems can be classified according to their data sources into: Host-based Intrusion Detection System (HIDS) and Network Intrusion Detection System (NIDS). HIDS is aimed at monitoring one or more hosts. It observes the file system, system calls, processing, memory operations, applications, among others, looking for changes or irregularities that may be related to a breach of security policies (Brown et al., 2002; Liu et al., 2018).

NIDS is an approach to monitor the network, usually by observing network traffic. Different strategies are possible to observe this environment, such as using malware signatures to perform monitoring (Kumar, 2007). This network-oriented approach must place sensors in strategic positions, allowing inbound, outbound, and local network operations to be observed.

Studies about IDS solutions are motivated by (I) speed in identifying attacks, identifying and blocking an attack before any damage is done; (II) an efficient detection system; and (III) in addition to performing detection, the system must be able to collect information and store it (Stallings et al., 2012).

2.2 BROWSER SECURITY

A browser or "web browser" is an application intended to allow access to the World Wide Web (WWW). Its ultimate goal is to retrieve and present remote content in a user-friendly way. Due to their importance, complexity, and widespread use, these applications receive a great deal of attention from attackers, who aim to exploit them to gain access, extract information, or even carry out phishing attacks on users (Reis et al., 2009).

Web Browsers are used in a wide range of devices, including desktops, laptops, TVs, and smartphones. This also reflects the complexity behind these applications that will run a multitude of functionalities for users, without the requirement of external applications and dependencies. Therefore, today these applications are not only used to browse the Internet, but also for querying databases, processing data, and running operations that in the past would require an entire Operating System (OS) (Taivalsaari et al., 2008).

2.2.1 Browser Architecture

A browser is composed of an interface, which is responsible for the interaction between the user and the application, and an engine, which will interconnect the interface with other components such as network, storage, and interpreters. The engine is responsible for the heavy work of sending and receiving messages, storing desired content, and executing/interpreting content found on web pages to present to the user.

Figure 2.2 presents the architecture of the Firefox web browser. The *top-down* view, describes how a user interacts with the browser from the Interface to the engine. This architecture can vary a little in other browsers, it is also possible to highlight that the position of each resource is a representation of its "desired position". Numerous components are needed in several regions of the application and, consequently, isolating them in a single position turns out to be impossible (Grosskurth and Godfrey, 2006; Barth et al., 2008).

Figure 2.3 shows a five-step process in how a browser processes web content by interacting with a tree data structure known as Document Object Model (DOM) (Mozilla, 2022a). The DOM is an interface that handles Extensible Markup Language (XML) and HTML after the content has been loaded and parsed. Cascading Style Sheets (CSS), XML, HyperText Markup Language (HTML), and JavaScript (JS) manipulate web page contents by reading and modifying DOM attributes.

JS uses the DOM Application Programming Interface (API) to obtain knowledge about the components of a web page (other components as WebAssembly also do the same process). This allows scripts running on a web page to access, manipulate, and perform the operations expected from the web page (Mozilla, 2022b). As shown in Figure 2.3, DOM is created after the parsing step, and ends up being accessed both by the compiler and by other components of the page to fulfill its duties.

2.2.2 Execution Environment

Browsers use the same engine to run WebAssembly and JS applications, despite the different execution models (Yan et al., 2021). JS employs dynamic typing, with variable types being inferred by the runtime during execution. As a consequence, optimization is a complex task since each time a function or variable is being used, the typing must be checked, to guarantee that JS can understand what kind of information it is dealing with.

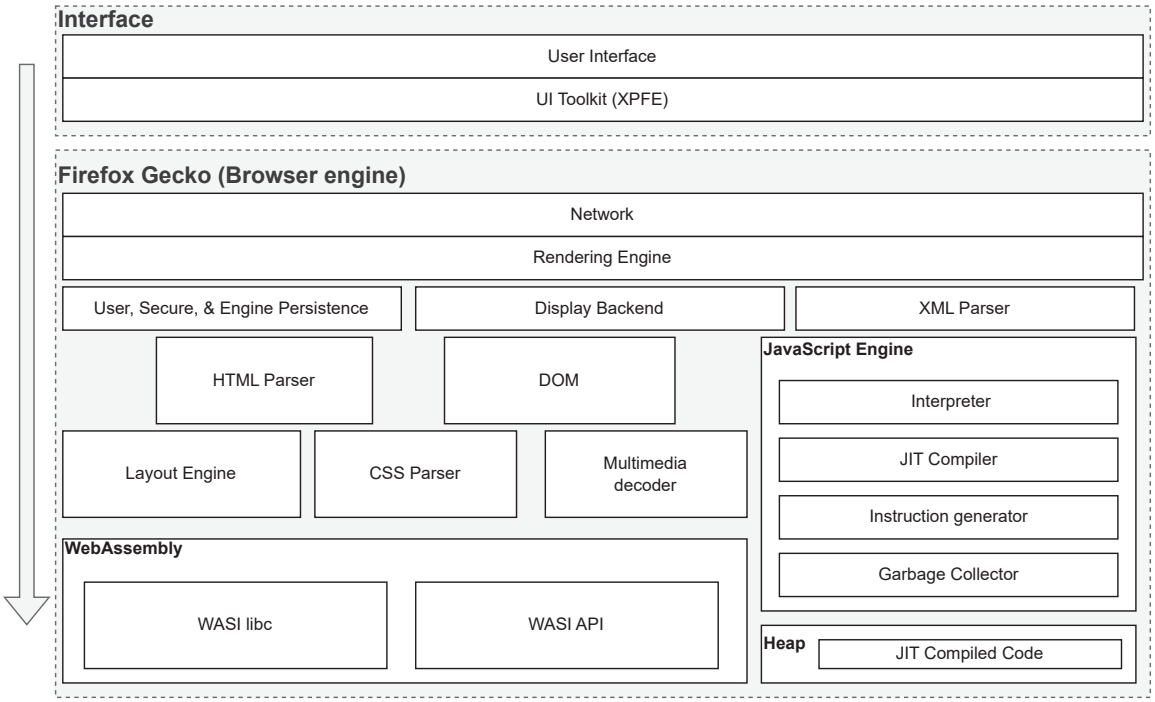


Figure 2.2: Architecture of Firefox web browser.
Source: Based on (Grosskurth and Godfrey, 2006; Zhang et al., 2015).

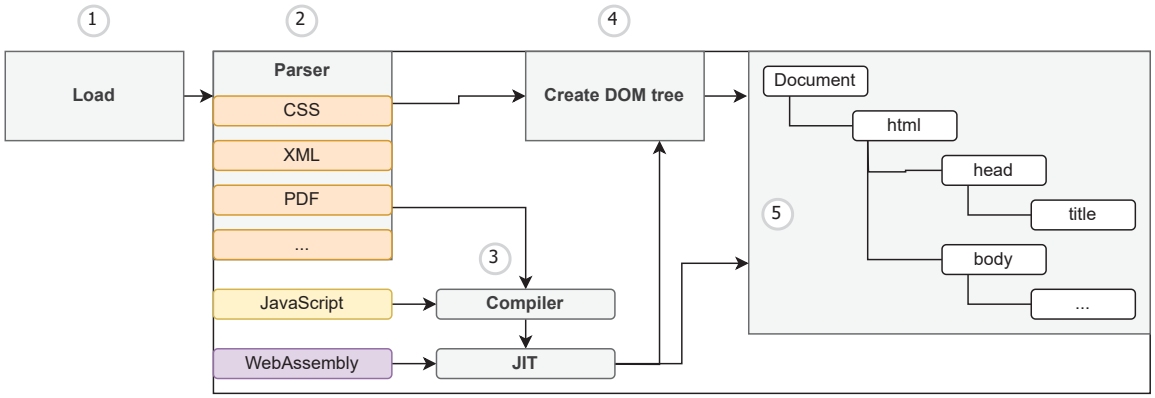


Figure 2.3: Representation of DOM.
Source: Built using information from (Mozilla, 2022a).

This problem does not occur in WebAssembly, in which four variable types exist, and the compiler does not have to worry about type changes during the execution.

Just-In-Time (JIT) compilation was proposed for JS before the emergence of WebAssembly, with the idea of using binary code to improve performance. Thus, functions that are repeated could be compiled to get a performance gain. This solution was implemented and is being used, but it is limited due to type changing, since there is the possibility of changing the type of a variable in a function at any time. Another relevant factor contributing to performance overhead in JS is the existence of a garbage collector, responsible for managing memory usage.

Unlike JS, WebAssembly does not need to be parsed at runtime as the bytecode is already compiled to run in the WebAssembly virtual machine. Of course, if an application in WebAssembly needs to import some functionality from JS, the limitations of the JS runtime will appear. As a linear memory model is used in WebAssembly (detailed in

Section 3.1), a garbage collector does not exist. Operations in memory by consequence turn out to be similar to operations performed on a buffer in dynamically allocated memory (Yan et al., 2021).

2.2.3 Browser Security

A web browser interacts with external resources, being responsible for receiving and executing external commands (without user control). This application also communicates through encrypted channels (which limits the observation by a defender), performs routines in the background, and executes instructions that may require sending information (Alcorn et al., 2014). The main security feature of a web browser is the Same Origin Policy (SOP). This control mechanism restricts the interaction of resources with different origins.

The two main attack vectors against web browsers are (I) software vulnerabilities, i.e., bugs with cybersecurity implications (this is not the focus of this work); and (II) browser features that can be exploited for a malicious purpose, such as executing malicious JS programs (Bandhakavi et al., 2010).

Well-known attacks exploiting browser features include *Injection attacks*, where codes and commands are embedded in requests (similar to SQL commands, and stack-smashing), Cross-site reference forgery (XSRF) that steal cookies from other tabs, Cross-site request forgery (CSRF) that replaces content from one page to redirect the user to another page. However, such attacks require the user to access or run content from a specific page (Wadlow and Gorelik, 2022).

An assessment of vulnerabilities found in web browsers is presented by (Šilić et al., 2010). The web browsers are monolithically structured and suffer from problems found in the sandbox, cross-site attacks, and session hijacking. However, it is possible to point out that recurring vulnerabilities in the literature have already been fixed in the most popular web browsers.

Due to the popularity of JS, attackers also aim to exploit it to carry out attacks. (Fraiwan et al., 2012) presents a proposal aimed at identifying attacks that use JS. The dataset used in the study is small, with only 6.7 k application samples. The proposed approach was able to reduce the number of false positives and false negatives through the use of 32 features, some of them already used in a previous work. An algorithm extracts such information from the code and stores it for later evaluation.

With a focus on studying malicious JS, (Patil and Patil, 2017) evaluates the semantics and syntax of malicious JS to identify relevant features when performing attack identification. To identify attacks, the proposal uses a strategy that considers the most frequent words, expected behavior of JS, content, and expected output of the application. The results highlight an adequate strategy for identifying attacks with results superior to other proposals found in the literature.

System calls can also be used for intrusion identification in web browsers (Canfora et al., 2014; Pendleton and Xu, 2017). However, using system calls for this purpose requires dealing with noisy data: it is necessary to collect and process all the system calls issued by the browser, as it is unfeasible to isolate only those issued by target browser components.

2.3 SYSTEM CALLS

System calls provide an interface between applications and the operating system kernel (Linux Programmer's Manual, 2021b). Applications use system calls to access input/output

devices and interact with OS abstractions (e.g., files, communication channels, processes) and services (e.g., memory management, process scheduling). In a nutshell, when a running application needs to do anything beyond computing with data already mapped to its address space, it issues a system call to request the OS to intervene and perform the operation on its behalf.

Figure 2.4 illustrates how user space applications interact with the Linux kernel. *User App1* and *User App2* represent two user applications running on the OS. The difference between these applications is the use of a wrapper functions (in this case the GNU C library, or glibc) to execute or request a certain resource. Using wrapper functions (as done by *User App1*) is the norm, but the Linux kernel allows system calls to be invoked directly (this is what *User App2* does).

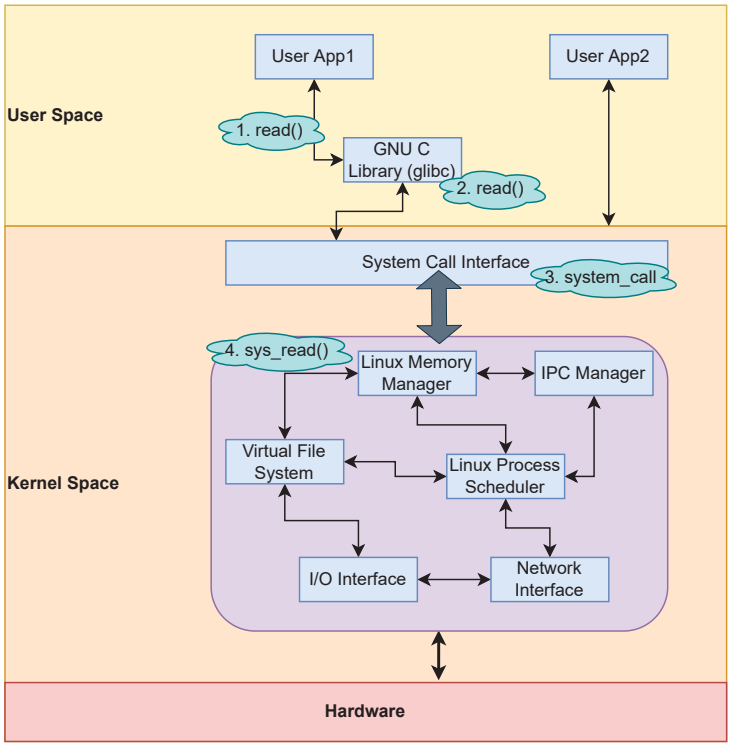


Figure 2.4: *System calls* flow on Linux.
Source: Author

An example of interaction is presented by *User App1*. The application invokes the *read()* function from the C library (1), which in turn invokes the *read()* system call in the kernel (2). This is done through the system call interface (3), which maps the call to the internal implementation provided by *sys_read* (4).

A library wrapper handles the requirements before invoking a system call. The wrapper is responsible for (Linux Programmer's Manual, 2021a):

- Copying the call parameters to the registers where the kernel expects them;
- Trapping to the kernel, who performs the system call; and
- Checking for error conditions and propagating them back to the application.

A system call is usually associated with a set of information needed to perform a certain task. This information is defined by function parameters, which are required to

perform operations. The parameters will vary according to the system call, and may include memory addresses (pointers), OS resource descriptors, resource-specific flags, resource demands, mode, among others. Listing 2.1 shows the *mmap()* system call, which maps files or devices into memory; its parameters are *addr* (a memory address), *len* (length of the mapping), *prot* (protection flags), *flags* (flags that control aspects of the mapping), *fd* (a file descriptor), and *offset* (the starting offset within the file).

```
1 #include <sys/mman.h>
2 void * mmap(void *addr, size_t len, int prot, int flags, int fd, off_t offset);
```

Listing 2.1: Function parameter example.

The set of system calls provided by an OS usually evolves over time, either to support new functionalities or to provide a better interface to an existing functionality. To ensure backward compatibility and allow gradual adoption, it is common to introduce new system calls alongside the existing ones, keeping the old system calls in the kernel even when newer versions are available. OSs that target multiple architectures may also have some architecture-specific system calls.

2.3.1 Example of system call use

Listing 2.2 presents an example of an application trace obtained using the strace tool on a Linux system. These forty-nine lines show the execution of the command `cat` on a file `README.md`. This sequence demonstrates calling the application *cat*, reading the file `README.md`, storing the information in memory, presenting the information to the user and closing the application. Lines 36–45 show the system calls actually issued by the application; lines 1–36 and 46–48 correspond to calls issued by the shell and the runtime.

```
1  execve("/usr/bin/cat", ["cat", "idpt-2021/README.md"], 0x7ffd366101b8 /* 26 vars */) = 0
2  brk(NULL)                               = 0x5589e7532000
3  arch_prctl(0x3001 /* ARCH_??? */, 0x7fff7a778b40) = -1 EINVAL (Invalid argument)
4  access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
5  openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
6  fstat(3, {st_mode=S_IFREG|0644, st_size=31934, ...}) = 0
7  mmap(NULL, 31934, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f8b21174000
8  close(3)                                = 0
9  openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
10 read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\360\215\2\0\0\0\0"..., 832) = 832
11 pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) = 784
12 pread64(3, "\4\0\0\0\20\0\0\0\5\0\0\0GNU\0\2\0\0\300\4\0\0\0\3\0\0\0\0\0\0\0", 32, 848) = 32
13 pread64(3, "\4\0\0\0\24\0\0\0\3\0\0\0GNU\0~\303\347M\250B\312<j\233\242\v!0<\341"..., 68, 880) = 68
14 fstat(3, {st_mode=S_IFREG|0755, st_size=1995896, ...}) = 0
15 mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f8b21172000
16 pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) = 784
17 mmap(NULL, 2004064, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f8b20f88000
18 mprotect(0x7f8b20fae000, 1810432, PROT_NONE) = 0
19 mmap(0x7f8b20fae000, 1495040, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x26000) = 0
       x7f8b20fae000
20 mmap(0x7f8b2111b000, 311296, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x193000) = 0x7f8b2111b000
21 mmap(0x7f8b21168000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1df000) = 0
       x7f8b21168000
22 mmap(0x7f8b2116e000, 13408, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0
       x7f8b2116e000
23 close(3)                                = 0
24 mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f8b20f86000
25 arch_prctl(ARCH_SET_FS, 0x7f8b211735c0) = 0
26 mprotect(0x7f8b21168000, 12288, PROT_READ) = 0
27 mprotect(0x5589e56d4000, 4096, PROT_READ) = 0
28 mprotect(0x7f8b211aa000, 4096, PROT_READ) = 0
29 munmap(0x7f8b21174000, 31934)          = 0
30 brk(NULL)                               = 0x5589e7532000
31 brk(0x5589e7553000)                     = 0x5589e7553000
32 openat(AT_FDCWD, "/usr/lib/locale/locale-archive", O_RDONLY|O_CLOEXEC) = 3
33 fstat(3, {st_mode=S_IFREG|0644, st_size=3041456, ...}) = 0
34 mmap(NULL, 3041456, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f8b20c9f000
35 close(3)                                = 0
36 fstat(1, {st_mode=S_IFCHR|0600, st_rdev=makedev(0x4, 0x1), ...}) = 0
37 openat(AT_FDCWD, "idpt-2021/README.md", O_RDONLY) = 3
38 fstat(3, {st_mode=S_IFREG|0644, st_size=63, ...}) = 0
39 fadvise64(3, 0, 0, POSIX_FADV_SEQUENTIAL) = 0
40 mmap(NULL, 139264, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f8b20c7d000
41 read(3, "https://sites.google.com/inf.ufp"..., 131072) = 63
42 write(1, "https://sites.google.com/inf.ufp"..., 63) = 63
43 read(3, "", 131072)                     = 0
44 munmap(0x7f8b20c7d000, 139264)          = 0
```

```
45 close(3)                              = 0
46 close(1)                              = 0
47 close(2)                              = 0
48 exit_group(0)                         = ?
49 +++ exited with 0 +++
```

Listing 2.2: Example of a system call trace.

As discussed in Section 2.3, processes are confined and require assistance from the OS (via system calls) to do anything beyond manipulating data already mapped to their address spaces. This includes allocating memory, communicating with other processes or over the network, manipulating files, accessing I/O devices, and adjusting privilege levels. This means that a malicious application has to resort to system calls to perform security-critical operations, such as read/write other processes' data, read/write data from/to the network, or escalate its privileges. A system call is required either for the operation itself or for gaining access to an OS abstraction (such as memory-mapping a file) that enables the process to perform the unwanted operation. Therefore, monitoring the system calls issued by a process allows us to observe its security-relevant behavior.

2.3.2 Intrusion Detection using System Calls

Thus, a behavior classification process can use *system calls* to identify malicious applications (Jain and Sekar, 2000). In the literature, monitoring applications through the use of *system calls* is already a common technique, and system call attributes (including the attributes of the calling process, system call parameters, and preceding system calls) may be used to identify unwanted processes/behavior (Rajagopalan et al., 2006).

The concept of using system calls to identify attacks in a OS was proposed in 1996 by (Forrest et al., 1996). The study concludes that the use of system call sequences can be used to identify anomalies, and that this strategy appears to be feasible for real-time evaluations. It also highlights that although applications generate varied sets of calls, smaller sequences tend to be similar and present patterns similar to previous evaluations. These conclusions are essential to the use of system calls and have been explored over the years.

There are different approaches for using system calls as a data source for intrusion detection. System calls may be analyzed individually, in batches (usually employing a sliding window of size N), or as complete execution traces. System call traces may be augmented with external data (such as library calls). Another issue is whether or not to use system call parameters in the detection process (which requires collecting these parameters in the first place).

Different techniques can be used for the identification process. In the literature, some of the techniques used are: clustering, Hidden Markov Models (HMMs), and neural networks (Liu et al., 2018).

The popularization of this strategy contributed to the definition of threat levels for system calls. (Bernaschi et al., 2002) presents one of the first classifications of system calls, which takes into account the threat level of the operations that each call performs. The work presents the REference Monitor for UNIX Systems (REMUS) which aims to monitor system calls according to a set of classifications. Altogether four threat levels are proposed, which aim to group from harmless calls up to calls that offer total control of the system. By monitoring these operations, it is possible to identify and block attacks such as buffer overflow. The work also presents details of how system calls interact and are used in the OS.

Over the years this concept has matured and has been used with a varied set of strategies to identify intrusions. Some examples would be: (a) monitoring of specific applications or even specific groups of system calls (Bridges et al., 2019); (b) Monitoring of virtualized environments (Castanhel et al., 2021) and cloud computing (Liu et al., 2021a); (c) Use of pattern identification strategies such as machine learning (Creech and Hu, 2013); and (d) Improving the identification of attacks that use obfuscation techniques to try to hide operations by using system calls (Văduva et al., 2019; Zhang et al., 2021).

Finally, an extensive review of the literature is presented by (Liu et al., 2018), which identifies varied strategies, models, and datasets used to identify intrusion by means of system calls. The study highlights strengths when using system calls to identify intrusion and strategies that combine more than one technique for identifying attacks, where it is possible to highlight a clear increase in the identification rate.

WebAssembly applications running in browsers issue WASI calls to the underlying runtime. Such calls are somewhat similar to system calls in functionality and behavior. Studies exploring system calls in WebAssembly, or comparing system calls and WASI calls, were not found in the literature. However, some authores have explored intrusion detection in WebAssembly; we review this body of work in Chapter 3.

2.4 MACHINE LEARNING

Machine Learning (ML) solutions are widely applied for cybersecurity, given that signature-based intrusion detection solutions are of limited use against new threats and may be circumvented by attackers (Handa et al., 2019). ML for cybersecurity focus on detection, analysis, attribution, triage, and forensics (Ceschin et al., 2020a).

2.4.1 Application for Cybersecurity

Figure 2.5 shows the steps involved in using ML for intrusion detection, based on (Pedregosa et al., 2011; Ceschin et al., 2020b; Sudusinghe et al., 2021). There are steps related to *data* operations, and a *pipeline* that consists of data investigation, feature selection, preprocessing, training, and testing.

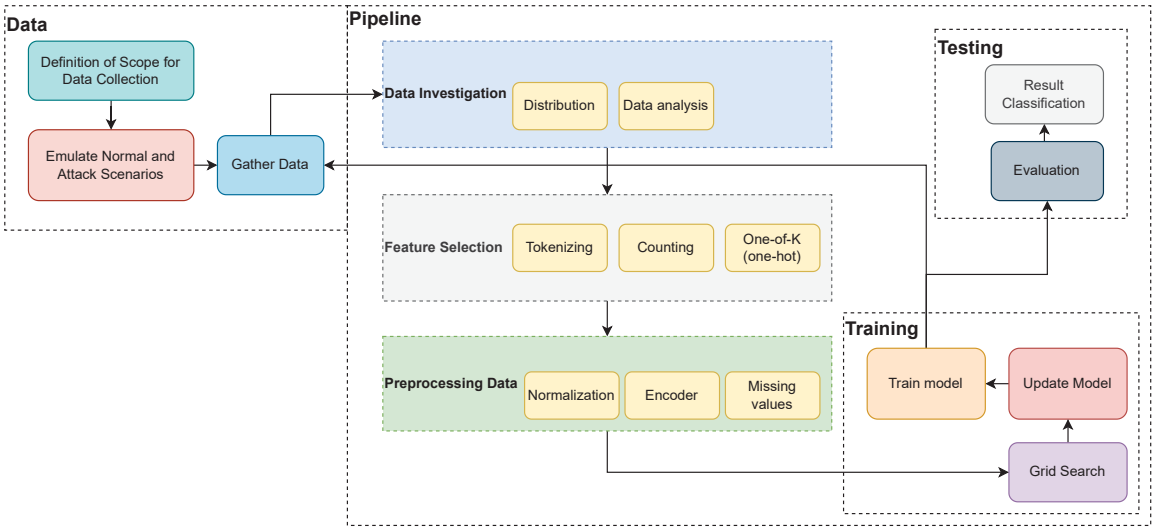


Figure 2.5: Steps of the ML based attack detection.
Source: Author

Before reaching the stage of training and processing the ML models, an evaluation dataset is needed. The three steps required for data collection are: (1) scope definition, which describes the limitations for the data collection and how this information will be collected; (2) emulation, which consists of running the applications that will generate the behavior; and (3) gather data, that consist of the data collection itself.

Three types of data can be obtained according to the type of data collection performed. This being (1) raw, which consists of raw data, without any processing, such as PCAP files, Executable and Linkable Format (ELF), and Android Application Pack (APK); (2) attributes, information extracted from raw data with a focus on filtering relevant information; and (3) features, raw information or attributes that have already been processed by a feature extraction algorithm (Ceschin et al., 2020b).

The next step consists of the data investigation, where the manual analysis of the data is performed to identify key characteristics and the best way to represent each feature. After that, strategies for feature selections are applied to the data, and the required preprocessing is done adapting the models for the next step. At last, the model is trained and tested with the data. In case the model does not present an adequate result, the process could restart in the data collection.

2.4.2 Evaluation Metrics

Metrics of evaluation are key calculations to understand how a ML model is learning and the quality of the data used in the learning phase. Confusion matrix are specifically useful since allows a better understanding of the sensitivity and specificity. A classification will achieve one of the following four states:

True Positive (TP) a correct accepting, demonstrating a presence of characteristics;

True Negative (TN) a correct rejection, indicating an absence of a characteristic;

False Positive (FP) an overestimation, wrongly indicating that a characteristic is present;

False Negative (FN) an underestimation, wrongly indicating that a characteristic is absent;

Precision or Positive Predictive Value (PPV) consists of the True Positive values divided by the True Positive and False Positive. The precision will represent the impact of False Positives in the models. The values describe the relevance of the retrieved instances.

$$PPV = \frac{TP}{TP + FP} \tag{2.1}$$

Recall or True Positive Rate (TPR) consists of the True Positive values divided by the True Positive and False Negative. The recall represents the impact of False Negative in the models. The values describe how many relevant instances were retrieved.

$$TPR = \frac{TP}{TP + FN} \tag{2.2}$$

The F1Score is the harmonic mean of the Precision and Recall, in which, both metrics are combined to evaluate the accuracy of the model.

$$F_1 = \frac{2TP}{2TP + FP + FN} \tag{2.3}$$

The Accuracy represent how close the values found are to the true value.

$$ACC = \frac{TP + TN}{TP + TN + FP + TN} \tag{2.4}$$

The Balanced Accuracy (BAC) considers the True Positive Rate and True Negative Rate (TNR). The True Negative Rate considers the False Positive and False Negative in comparison to the TPR.

$$BA = \frac{TPR + TNR}{2} \tag{2.5}$$

$$TNR = \frac{TN}{TN + FP} \tag{2.6}$$

The Brier Score consist of a mean squared difference between probability and an outcome.

$$B = \frac{1}{N}\sum_{t=1}^{N}(f_t - o_t)^2 \tag{2.7}$$

2.4.3 Categorical Data Representation

Before applying the data to a ML model, a type of representation or encoding needs to be used. Different data types and encoding can be used, through the use of a range of techniques the data is retrieved from the dataset and encoded (Kunft et al., 2019; Hancock and Khoshgoftaar, 2020). Considering a categorical data representation, a data structure is built that can be used to encode the dataset into a finite number of classes that represent a specific characteristic in the dataset.

Techniques for encoding such as labeling, counting, hashing, and leave-one-out, are a small sample of strategies that can be used to represent categorical data (Hancock and Khoshgoftaar, 2020). Quantitative and qualitative data are represented by the technique that better fits the data. The representation occurs after the data analysis. From a machine learning perspective, the representation process occurs during the preprocessing phase. The result is a transformation of the dataset, that is adequate for the training and testing of the ML models.

Categorical data or categorical variables are defined as variables that are measured by categorizing a limited set of "values" (Powers and Xie, 2008). Categorical data are common in the social sciences field; some of the information represented by categories are gender, age, language, and location. Categorical variables are a type of representation that provides a form of classification of items according to a specific constraint, characteristic, or standard (Agresti, 2003; Charu, 2017).

There are a variety of measurement types (Powers and Xie, 2008), as shown in Figure 2.6. Before discussing categorical data in deep, it is important to understand the relation of categorical variables with these other types of measurements.

A quantitative measurement uses a numerical value to describe variables (Powers and Xie, 2008). Quantitative data is countable and measurable, being divided into continuous and discrete variables. Continuous variables can assume any real value, unlike discrete variables that can only take integer values. Discrete variables are also considered as categorical variables (Powers and Xie, 2008).

In a qualitative measurement, the data is non-numerical. Variables to describe the quality and relationship between variables. A qualitative variable is also categorical

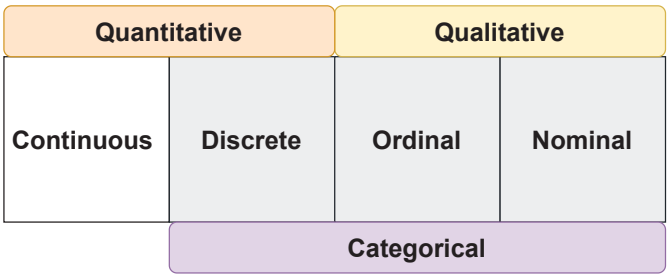


Figure 2.6: Types of Measurements and relation with Categorical data.
Source: Based on (Powers and Xie, 2008).

and is distinguished in ordinal and nominal variables (Powers and Xie, 2008). Ordinal variables are concerned with ordering the data, as an example, we can consider revision notes *reject*, *weak reject*, *weak accept*, and *accept*. It is not possible to describe a measure of distance between the values, but it is possible to create an ordering between them. Nominal variables have no order, and the distance between observations cannot be measured. An example would be dog breeds, which can assume values such as *German Shepherd*, *Labrador*, *Rottweiler*, and *Mastiff*. The values cannot be ordered, and it is not possible to say, for instance, that *German Shepherd* is closer to *Rottweiler* than *Mastiff*.

It is important to notice that numerical representation can be used for qualitative and categorical variables. However, the distance and the numerical value does not have the same meaning (Li et al., 2016).

Categorical data encompass qualitative (nominal and ordinal) and discrete quantitative variables. There may be different possibilities for categorizing a variable, depending on the application. For example, an Internet Protocol (IP) address can be categorized as *private* or *public*, where a private address is an address that should only be used in internal networks and a public address is globally routable. This is an example of a qualitative nominal categorization. IP addresses may also be categorized according to their network address: 192.0.2.4 and 192.0.2.51 belong to the 192.0.2.0/24 category, while 128.9.2.51 belongs to the 128.9.0.0/16 category. This is an example of a qualitative ordinal categorization (network addresses can be ordered, and it is possible to define a distance measure between networks). Using categories to describe IP addresses can provide more compact representation of data (1 bit is enough to differentiate between *public* and *private*), and may lead to simpler decision rules (for instance, in a network access control policy, one may write rules that apply to all *private* or *public* IP addresses).

As another example, system calls can also be represented as categorical data. An OS has a finite set of system calls, each with a well-defined behavior. It is possible to classify the system calls according to their functionality. Thus, *open* and *read* calls belong to a *file manipulation* category, *mmap* and *sbrk* belong to *memory management*, *fork* and *exit* belongs to *process control*. These groups can be used later to identify patterns or behaviors. So, system calls are qualitative data since there is no numerical relationship between the calls, and therefore nominal since the concept of order between *system calls* does not exist. But this dataset can also be represented through categorical approaches since there are well-defined groups of *system calls*.

A categorical representation of a population can consist of a one-to-one correspondence, where there is a direct representation between the population samples and the categories. Otherwise, it can follow a threshold to classify the population samples according to their respective categories (Powers and Xie, 2008). For example, a many-to-one

classification can consider the car's respective manufacturer for the categorization or the car's respective power for a threshold categorization.

Machine Learning (ML) models use categorical encoders to transform, represent, or convert features. Encoding features to categorical schemes also allows numeric representation to be used for specific models that would be limited to numerical inputs (Pedregosa et al., 2011). Categorical data representation allows both the simplification of data representation (such as replacing IP addresses with public/private classes) and the addition of qualitative characteristics to quantitative attributes (as presented by (Bernaschi et al., 2002), where a threat factor was associated with each system calls).

Taking into account possible learning techniques, both cases can take advantage of categorical representations. Unsupervised strategies do not have previous knowledge about the data label and are required to learn what would be normal, abnormal, or data noise. In supervised scenarios, the label for the data is presented, where this previous knowledge is used to distinguish the difference between the data points (Charu, 2017).

2.5 CONCLUDING REMARKS

This chapter presented in detail the principles of intrusion detection and its respective strategies, along with the execution model present in web browsers. The concepts of system calls were presented, and how system call data can be used in the process of identifying threats. Finally, ML concepts were discussed with a focus on solutions for anomaly detection.

3 WEBASSEMBLY

This chapter presents the WebAssembly format concepts, contextualize the use of the format, and its current state. Section 3.1 gives an overview of WaebAssembly. Section 3.2 present its development and use. Section 3.3 present the current state of WebAssembly development.

3.1 OVERVIEW

WebAssembly, also known as Wasm, is a format designed to be a binary target for programming languages. The portability allows developers to choose the most suitable language for implementing an application. Therefore, a range of languages not previously supported in the Web environment now can be used. With high-level languages, developers have more flexibility, access to frameworks, and a compact binary target.

WebAssembly bytecode is executed inside a stack-based virtual machine. Its Instruction Set Architecture (ISA) is a binary format designed to be run inside a virtual machine, allowing the execution of WebAssembly in a range of architectures. Applications can be run in a host with a runtime Command-Line Interface (CLI), from a server-side, inside the Web browser, or even on Internet of Things (IoT) devices. From a client-side perspective, compact binaries are retrieved from a server to be run inside a sandboxed environment, offering a performance gain relative to interpreted languages such as JavaScript (JS).

Figure 3.1 presents the expected interaction for a WebAssembly application. A WebAssembly binary is executed inside a sandbox environment, where instructions push and pop data from the stack, information is stored in the linear memory, and external resources can interact with the sandbox environment by an exposed function (Rourke, 2018). An example of external interaction through the exposed function is presented in Figure 3.1.

Instructions in a stack-based machine will assume that most operations are on the stack instead of registers (Hoffman, 2019). The WebAssembly stack is a Last-In, First-Out (LIFO) structure, any interaction and commands with the stack inside the virtual environment will be limited to the element at the top of the stack (Battagline, 2021). These design choices allow easy portability from other languages and binaries with a small size (Hoffman, 2019).

The WebAssembly format support four data types: `i32` (32-bit Integer), `i64` (64-bit Integer), `f32` (32-bit floating-point numbers), and `f64` (64-bit floating-point numbers). Intrinsic signed-ness to numeric representation is only carried out when an operation is made, with specific operations for signed or unsigned data (Hoffman, 2019). Strings can only be used through linear memory. The control of elements found in the memory for external resources as JS is made by passing the length and position of the `String` in the linear memory (Battagline, 2021). Since WebAssembly does not have a heap, Object-Oriented Programming (OOP) does not exist and objects are not stored in the same way as other programming languages. The linear memory is a block of bytes, if more space is required the blocks can be incremented in pages up to a fixed size limit for linear memory. Access to the host memory through the linear memory is not possible.

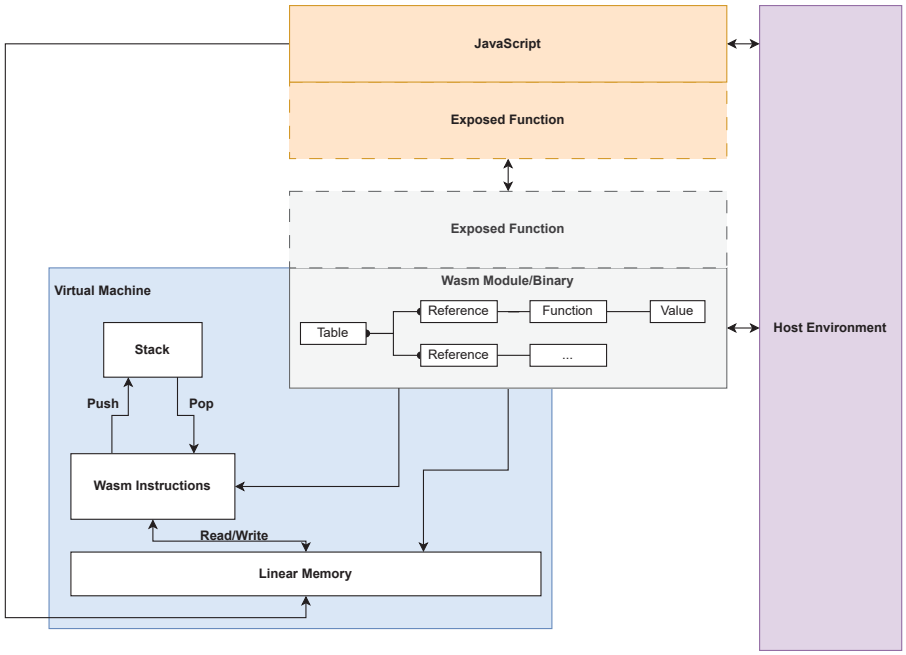


Figure 3.1: WebAssembly design.

Table 3.1 presents the structure of a WebAssembly binary (or Wasm module), the binary being the result of a compilation of a `.wat` file or through porting from other languages such as Rust or C to WebAssembly. A Wasm binary is composed of a WebAssembly module, which is structured in three sections: *Preamble*, *Standard*, and *Custom*. The *Preamble* indicates that the file is a WebAssembly module and the version format being used. The *Standard* section presents known sections that have specific functionalities. Table 3.1 shows eleven standard sections. No sections are required by a Wasm module, appearing only when needed. The *Custom* section can be in any position of a module, allowing the inclusion of data that can be used for debugging or functions (Hoffman, 2019).

Preamble
Magic
Version
Standard Section
Type
Import
Function
Table
Memory
Global
Export
Start
Code
Element
Data
Custom Section
Any kind of data

Table 3.1: Structure of the WebAssembly binary (A WebAssembly Module).

3.2 DEVELOPMENT AND USE

WebAssembly applications are not limited to only implementation using third-party languages. A human-readable representation called WebAssembly Text (WAT) allows developers to write WASM code using S-Expressions or linear instruction list styles (Battagline, 2021; W3C Community Group, 2022b). A WebAssembly application (or a WebAssembly Text (WAT) file) contains a module, which contains functions and variables (following the structure presented in Table 3.1). Export and import operations provide access to WebAssembly functions. The export operation makes a Wasm function available to other operations in the environment (e.g., a JS shared value during the execution process). Functions in the environment can also be imported to the WebAssembly using the import operation.

Listing 3.1 presents the implementation of a simple `Hello World` in WebAssembly using the WAT format.

```
(module
    ;; Import the required fd_write WASI function which will write the given io vectors to stdout
    ;; The function signature for fd_write is:
    ;; (File Descriptor, *iovs, iovs_len, nwritten) -> Returns number of bytes written
    (import "wasi_unstable" "fd_write" (func $fd_write (param i32 i32 i32 i32) (result i32)))

    (memory 1)
    (export "memory" (memory 0))

    ;; Write 'hello world\n' to memory at an offset of 8 bytes
    ;; Note the trailing newline which is required for the text to appear
    (data (i32.const 8) "hello world\n")

    (func $main (export "_start")
        ;; Creating a new io vector within linear memory
        (i32.store (i32.const 0) (i32.const 8))  ;; iov.iov_base - This is a pointer to the start of the 'hello world\
            n' string
        (i32.store (i32.const 4) (i32.const 12))  ;; iov.iov_len - The length of the 'hello world\n' string

        (call $fd_write
            (i32.const 1) ;; file_descriptor - 1 for stdout
            (i32.const 0) ;; *iovs - The pointer to the iov array, which is stored at memory location 0
            (i32.const 1) ;; iovs_len - We're printing 1 string stored in an iov - so one.
            (i32.const 20) ;; nwritten - A place in memory to store the number of bytes written
        )
        drop ;; Discard the number of bytes written from the top of the stack
    )
)
```

Listing 3.1: WAT module (Example from Wasmtime (BytecodeAlliance, 2021a)).

We can understand the WebAssembly workflow through three main abstraction layers. The first layer describes the compilation target and high-level language supports. This is one of the main resources of WebAssembly and brings advantages to the development of applications for the web environment, where a range of languages can be ported to it. The second layer is the Intermediate Representation (IR), which contains translation and transformations made by compilers until producing the code for a specific machine. The third and last layer consists of the result of the previous process. The resulting bytecode is executed on the WebAssembly runtime, allowing execution in different architectures.

In practice, WebAssembly is used for cryptography, video, audio, graphics, and other activities that may enjoy the advantages of the format design. However, the WebAssembly format is not limited to the web browser. Server-side applications, distributed services, and mobile applications can also take advantage of WebAssembly (Rossberg, 2022).

3.2.1 Security Model

Aiming for better security, WebAssembly applications are executed inside a sandbox environment. An application is only able to access information outside of the sandbox if the correct permission is granted. The WebAssembly module follows the Same Origin Policy (SOP) and access to specific resources also require the use of an Application Programming Interface (API) (Kim et al., 2022).

Through the linear memory, data is passed to a WebAssembly module, and information is shared between JS and WebAssembly, acting as an array of data. The memory is allocated through pages and after being allocated they cannot be deallocated. The developer must track what is being stored in memory and where it is (Battagline, 2021).

3.2.2 WASI

The standard API defined for WebAssembly development is known as WebAssembly System Interface (WASI). It defines the rules for the runtime and interactions that WebAssembly performs with the environment. Through WebAssembly System Interface (WASI), WebAssembly can perform native operations found in other programming languages, in addition to guaranteeing the security levels defined for the WebAssembly (Battagline, 2021).

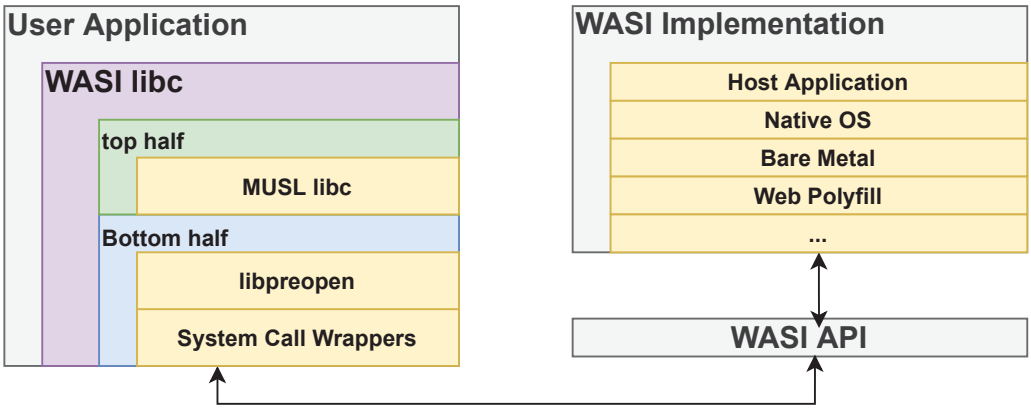


Figure 3.2: WASI Software Architecture.
Source: Based on (BytecodeAlliance, 2021a).

The initial focus of WASI API is to provide access to resources like files and networking. The WASI API uses WASI libc (standard C library), which aims to be a libc interface. This implementation can be divided into `bottom half`, which is composed of `libpreopen` and system call wrappers; and `top half`, which is an implementation of MUSL libc[1] (BytecodeAlliance, 2021a). Figure 3.2 shows this architecture, with the expected communication flow. The system call wrapper is responsible for making calls to the WASI API. Consequently, the WASI API makes the corresponding calls to the WASI implementation present in the system. These calls can be processed by the Operating System (OS) or runtime resources of some language.

Every interaction between the application and the operating system is carried out through WASI calls. These calls have functionality similar to system calls, as without these calls an application would not be able to access system resources.

[1]MUSL libc implements the C standard library on top of the Linux system calls API (Felker, 2023).

3.3 CURRENT STATE OF WEBASSEMBLY

Emscripten and Wasmtime are the two most popular compilers for WebAssembly. Both follow the WASI standard but have different goals. Emscripten is based on LLVM Compiler Infrastructure (LLVM, 2023), and is focused on compiling C/C++ code into WebAssembly. The primary execution target are web browsers, and required JS and HyperText Markup Language (HTML) files are packaged alongside the generated *.wasm* code (Emscripten, 2021).

Wasmtime generates *.wasm* applications that may be executed either in the browser or as a host application without requiring a browser engine. Due to these characteristics, Wasmtime provides a range of debugging tools to evaluate code interaction with the host and also provides a way to retrieve WASI calls from the runtime of an application (BytecodeAlliance, 2021b).

A total of 45 calls are already implemented in Wasmtime; these are the calls that follow the WASI standard (we are going to refer to these calls by *WASI calls*). Wasmtime defines snapshots to control the feature support of the compiler, and consequently also controls the type of WASI calls supported in a specific version. The current version 6.0.0 of Wasmtime defines the WASI calls in the *snapshot0*, and future snapshot versions allow developers to use functions from a different snapshot, which consequently will provide more calls. The snapshot1 will follow the current *wasi_unstable*, and the *snapshot1* will implement the functions presented in *snapshot0*.

In general, compilers for WebAssembly are in early stages of development, and changes are frequent. The advantage of the Wasmtime compiler is in the way that WASI calls are handled, with the possibility of executing and observing these applications without the need to use a browser.

WebAssembly has an active community that is constantly adding new features to improve the current Minimum Viable Product (MVP). The current WebAssembly version 1.0 does not support a Garbage Collection (GC) for memory management and does not provide direct access to the Document Object Model (DOM) (Rourke, 2018).

3.4 CONSIDERATIONS

The design offered by the WebAssembly format ends up bringing practicality to the development of Web applications, with the constant development of new features and mechanisms to improve this new technology. The interactions of the WebAssembly sandbox that generates WASI calls with the host environment represent an interesting resource for study due to the possibility of observing the operations carried out in the application inside the sandbox environment.

4 WEBASSEMBLY SECURITY LITERATURE REVIEW

This chapter presents a systematic literature review about WebAssembly security. Section 4.1 presents the research questions and how the review was performed. Section 4.2 reviews the references deemed relevant. Section 4.3 discusses the main findings of our review.

4.1 METHODS

A systematic literature review is a method to analyze the state of the art of a specific topic of interest, providing a background for new studies (Kitchenham and Charters, 2007). Although the security discussion is not new when discussing WebAssembly, we did not find a current study that performs a literature review focused on defining gaps and structuring security issues/solutions for this environment.

Aiming at enlarging our knowledge in the context of WebAssembly security, we conducted a systematic literature review on this subject. To achieve our research objective, we define four Research Questions (RQs):

- **RQ1:** What security issues are encountered when using WebAssembly?
- **RQ2:** What has already been developed to improve security in WebAssembly?
- **RQ3:** How WebAssembly is being used to carry out attacks? and
- **RQ4:** What is the current status of the WebAssembly compilers?

The phrasing model proposed by (Wazlawick, 2009) suggests investigating: the technique + the area of application. Thus, the search string comprises *WebAssembly*, as the main subject of the search, and *Security* since we are focusing on only topics related to security. So the search string was: *WebAssembly AND security*.

The next step was to define the search engines. The focus was to cover search engines with more features like ACM Digital Library, IEEE Xplore, Scopus, and Springer Link (Ortega, 2014). To get the most up-to-date results as possible, we also used the arXiv preprint server. Google Scholar was used to include articles that were not present in the engines selected and because this engine indexes several databases and has white papers, which are relevant because they report problems facing the industry and not just the academia.

We use the same search string in all selected search engines. We select the article's title, abstract, and keywords search fields in Scopus. In the other engines, we choose all available fields, i.e., we use the default search mode.

All resulting publications were taken into account, and all papers that match the following Selection Criteria (SC) were selected for this review:

- **SC1:** Published after 2017 (release of WebAssembly);
- **SC2:** Written in English;
- **SC3:** Primary studies (i.e., not surveys, meta-analysis, systematic mappings, or reviews);

- **SC4:** Not duplicated;
- **SC5:** The paper was available (to us) for download; and
- **SC6:** Central theme is associated with WebAssembly Security.

The search was performed in August 2022 and updated in 2023. After applying the selection criteria to the search results, we obtained 83 studies. The papers that met the criteria were also analyzed and categorized.

4.2 WEBASSEMBLY SECURITY RESEARCH

Research about the WebAssembly format in the first year of the development proposal is quite limited since the focus was on defining the format. The first overview of the format is presented by (Haas et al., 2017). This is the first in-depth discussion about functionalities and design choices. The paper makes an excellent approach to present this new technology, discussing the concepts behind the validation process, execution, and respective restrictions. As one of the first texts on WebAssembly, the work helps the community with rich documentation and guides numerous works that follow.

Overall, WebAssembly research is being developed with numerous purposes. Current discussions focus on exploring the security within this technology, identifying design flaws, the use of this format with malicious intentions, and applying the benefits of this technology to other resources. WebAssembly is a new technology highly deployed and used in the wild to solve various problems, with a range of languages being used by developers and compilers in constant development in recent years (Stephen, 2022).

4.2.1 WebAssembly as an Attack Vector

From an attacker perspective, WebAssembly can be explored to carry a wide range of attack strategies. In this section, we focus on studies that consider the use of WebAssembly as an attack vector.

The first large-scale study to assess the presence and popularity of WebAssembly on the Internet was carried out by (Musch et al., 2019). It pointed out that 50% of the pages identified as using WebAssembly ended up exploiting this feature for some malicious activity (in the Alexa top 1 million[1], 56% of the pages are using WebAssembly for malicious purposes such as obfuscation or cryptojacking) and only 10% of the pages observed had unique codes. Therefore, before diving into WebAssembly flaws and vulnerabilities, we will discuss the types of attack vectors explored by attackers.

With a focus on the WebAssembly binaries (*.wasm*) that can be found on the Internet, the paper (Hilbig et al., 2021) collected and evaluated the security properties of 8.4 k binaries. They identified that binaries generated by languages that do not have security guarantees in memory can export these vulnerabilities to WebAssembly, the import of potentially dangerous API exists in 21% of the observed binaries, and 65% of the observed binaries are using a portion of the linear memory that is not managed.

Cryptojacking is an attack that aims to exploit their victim's computational resources to mine cryptocurrencies. WebAssembly is attractive for this type of operation

[1]The Alexa rank consists of a dataset where websites are ranked according to their traffic. (Le Pochat et al., 2018).

due to the performance gain in comparison to other web languages, since performance has a direct impact on paid earnings for this type of attack.

WebEth is a distributed cryptominer presented by (Tiwari et al., 2018). Its implementation uses JS and WebAssembly for a distributed web browser Ethereum miner architecture. Without the requirement of external dependencies, the implementation handles the memory and network restriction imposed on web browsers. Despite the performance difference of 30% slower than a C++ implementation, the lazy technique adopted is the main contribution of the work.

The authors (Bian et al., 2019) and (Bian et al., 2020) propose *MineThrottle*, a tool that allows identifying and interrupting the execution of this type of attack, through the use of WebAssembly features and known cryptojacking algorithms. One contribution of the work is the discussion of possible strategies for identifying this type of attack, presenting existing strategies and their strengths and weaknesses. The strategies for the detection of cryptojacking in the wild use a profile to penalize the mining process that could be running. The evaluation uses Alexa top 100 thousand and 1 Million. The fingerprint strategy reached a false negative/positive lower than 2%.

Seismic is a detection system that warns users when cryptojacking activity is detected (Wang et al., 2018). Signature-based antivirus detection is easily avoided by an attacker with the use of obfuscation strategies. Seismic uses semantic code features for the detection process, reaching an accuracy of 98% for four WebAssembly cryptojacking algorithm families.

MinerGate is a machine learning-based defense against cryptojacking (Yu et al., 2020), that uses a proxy extension to protect the web browser. The extension was trained with a set of WebAssembly mining algorithms and is capable of identifying elements interaction of a cryptojacking activity in the network.

The paper (Petrov et al., 2020) presents *CoinPolice*, a detection method based on a deep neural network classifier. The features used for the model consist of performance characteristics and execution patterns. The proposal detected 97.8% of miners and had a false positive of 0.7%. In the wild, 6.7 k pages with hidden miners were identified.

Obfuscation aims at the transformation of source or binary code with the objective of changing its appearance. This can be used for the protection of code intellectual property or to hide code with malicious purposes (Balakrishnan and Schulze, 2005). Since WebAssembly is a new player in the web environment, attackers may try to use it to circumvent security measures through the use of code obfuscation.

The paper (Romano et al., 2022) presents a new strategy to escape classical identification processes already proposed for JS through the use of code obfuscation. *Wobfuscator* is an obfuscation tool that aims to use WebAssembly to move part of the processing that would be performed in JS. In this way, the application will still reach its goal, but with key points of the implementation being moved to WebAssembly, it is possible to escape from static analysis solutions already proposed for the identification of attacks in JS.

A study of the potential of obfuscation through the use of WebAssembly is the focus of (Bhansali et al., 2022). Through a range of benign and malicious samples, different obfuscation strategies were applied and tested against a cryptojacking detector. The results have shown a highly effective opportunity in the use of obfuscation to prevent reverse engineering and decompilation of the Wasm binary. The success of obfuscation is influenced by the type of application and the code complexity, where, where code with specific characteristics will be harder to be obfuscated.

The work (Wang et al., 2019) presents a solution for obfuscating intensive numeric operations found in JS. This system, named *JSPro*, consists of a code virtualization for JS built in WebAssembly. However, the translation process is limited, and in some cases, applications could not be translated correctly into WebAssembly.

Code diversification for WebAssembly binaries is presented in (Arteaga et al., 2021). This strategy aims at the generation of different binaries for the same program. The study presents *CROW*, a framework for code diversification in WebAssembly. The solution enables the diversification of binaries and traces for a program. The framework is responsible for automating workflows for LLVM[2]. The results show that the proposal was able to achieve diversity when generating code for 79% of the total samples in the dataset used.

Write Primitive explores vulnerabilities from other languages, like C, that may allow an attacker to obtain a write primitive in WebAssembly. In some cases, this vulnerability is not present in the source language but appears in WebAssembly when ported (Lehmann et al., 2020). In the literature, some vulnerabilities used to reach this ability are:

Integer overflows/underflows: To store variables, the WebAssembly format supports static numeric types, similar to languages like C/C++ and Rust. An expected type needs to be defined with the variable that will store the value. Static typing provides better performance since there is no need to track the types of variables during execution. Languages like Python and JS have dynamic typing, where the type of a variable is assumed during runtime (this way the developer does not need to worry about the variable type, and this type may change during execution). However, when WebAssembly and JS interact with each other, an unexpected value could be sent from JS to WebAssembly. An example would be the creation of a value for a variable that is outside of the representation range, in WebAssembly, possibly resulting in an integer overflow. This could be further explored to perform a buffer overflow (McFadden et al., 2018).

Stack Overflow: An example of this attack in WebAssembly can be achieved in case of an excessive recursion or a violation in the internal assumptions of the stack. Instead of a crash, attackers will be able to overwrite data, since protection in the unmanaged stack does not exist in WebAssembly (Lehmann et al., 2020).

Heap Metadata Corruption: the attacker explores the memory allocator that was used in the Wasm binary, since this information is stored with the binary (Lehmann et al., 2020).

Overwriting Data is an attack primitive that allows attackers to overwrite data in a way that additional control is granted to the malicious actor during the execution of the application (Lehmann et al., 2020). Some of the vulnerabilities used to reach this ability of overwriting data in WebAssembly are:

Format String: if an attacker controls the format of a *print* function, he may read or write the linear memory. This happens because of the lack of support for some formats, which has a direct impact on the function operation (McFadden et al., 2018).

[2]The LLVM project offers a set of compilers and toolchains that can be used for the development of any programming language (LLVM-Team, 2023).

Stack-Based Buffer Overflows: WebAssembly has protections to block applications that try to write outside the bounds allocated for the linear memory (McFadden et al., 2018). However, it has no protection against the overwriting of variables found within those bounds. Variables could be overwritten by the use of an unsafe function (Lehmann et al., 2020).

Cross Site Scripting (XSS): Since variables stored in the linear memory may be overwritten, static variables are not safely stored. Information could be written over positions already occupied in the linear memory, allowing a XSS attack. The memory is not the only vector for XSS, attackers could also control function pointers (McFadden et al., 2018).

Overwriting Stack Data: Since no security mechanism is present in an unmanaged stack, the attacker could explore the linear memory by overwriting function data. However, the reach of this attack is limited, since return addresses are not present in the stack and only active calls will be explored (Lehmann et al., 2020).

Overwriting Heap Data: The memory design in WebAssembly puts the heap after the stack, with no defense mechanism to prevent attacks. The corruption of the heap is just a matter of overflowing the stack (Lehmann et al., 2020).

Overwriting Constant Data: The design of the linear memory makes it impossible to define an immutable variable. A constant data could be overwritten or changed by a stack overflow or a stack-based buffer overflow (Lehmann et al., 2020).

Redirect Indirect Calls: Through the overwrite of the linear memory, an attacker can change an index that could be used to issue an indirect function call (Lehmann et al., 2020).

Code Injection: Allows attackers to make unwanted changes in the environment. An example would be in the host, where code can be added to overwrite arguments found in the WebAssembly application.

Side-channel Attacks explore implementation mistakes that allow attackers to access and extract information from an application. Different strategies are explored according to the application and the environment (Spreitzer et al., 2017; Genkin et al., 2018). The invocation of functions present in Emscripten could enable a server-side code execution (McFadden et al., 2018).

A demonstration of side-channel attacks in WebAssembly is presented by (Genkin et al., 2018). The attack does not rely on vulnerability in the browser, exploring memory access from outside the sandbox. The attack was made against cryptographic libraries through a portable code.

Timing side-channel attacks is the focus of (Mazaheri et al., 2022). Despite the study focusing on JS, countermeasures are presented for these types of attacks in JS and WebAssembly. Cache attack and Spectre attack are the two strategies used to carry the side-channel attack. Countermeasures are presented with a detection approach named *Lurking Eyes*, which applies fixes at the hardware and software levels.

Porting Programs & Compilers: Compiling a program from a given language to a Wasm binary is known as porting a program. Considering a security perspective, some questions appear in the propagation of vulnerabilities of the source language and the

raise of an unexpected behavior. This occurs due to different design choices between them. The paper (Stiévenart et al., 2022b) presents findings about programs that would crash in the source language but would run in WebAssembly and different size types for variables.

A study of bugs in WebAssembly compilers is presented by (Romano et al., 2021). As WebAssembly applications tend to depend on a compiler that converts code from another language, dealing with the bugs becomes an even more complex problem. A total of nine unique challenges for WebAssembly compilers were identified when considering the development for WebAssembly. They are *Asyncify Synchronous Code*, *Incompatible Data Types*, *Memory Model Differences*, *Bugs in Other Infrastructures*, *Emulating Native Environment*, *Supporting Web APIs*, *Cross-Language Optimizations*, *Runtime Implementation Discrepancy*, and *Unsupported Primitives*. These challenges were found after investigating the Emscripten compiler, and most of the bugs are related to handling strings and filesystems.

Overall, when considering the compiler and the portability of the application, three attack vectors in WebAssembly are possible. The first would be to explore vulnerabilities in languages that will be compiled into Wasm. This strategy allows attackers to trigger **unexpected behavior**[3] and allows access to resources inaccessible in WebAssembly. The second vector is vulnerabilities found in the compiler, these vulnerabilities are due to particular features found in each implementation. The third vector is vulnerabilities found in WebAssembly, which exists because of its design. These vectors could be combined to effectuate an attack and, as discussed, a range of vulnerabilities and flaws are already mapped.

Table 4.1 presents an overview of the attack vectors previously discussed, with the respective attack category, the reference where this threat was presented, and references that propose countermeasures. Countermeasures solutions are few and have limitations when to be applied. For attackers, WebAssembly offers, in addition to a vast set of vulnerabilities, flexibility in using the WebAssembly format design to carry out attacks.

4.2.2 Improving WebAssembly Security

Improvements to WebAssembly security encompass studies that introduce fixes to previous design flaws and/or that add security features to the format and its environment. The use of WebAssembly sandbox to improve the security of an environment was also observed.

Sandbox is a restricted environment that allows the execution of applications that could contain malicious content or unreliable precedents (Prevelakis and Spinellis, 2001). WebAssembly is a portable sandbox that executes the application module inside a sandbox environment. The sandbox executes the application independently, only granting external access through the use of an API. This design allows the fast execution of applications in the web environment, since the application is ready to run after the download of the byte code. The paper (Bosamiya et al., 2022) explores two sandboxing techniques for WebAssembly, evaluating the security and performance of the proposals. The first technique uses machine-checked proofs to ensure that all inputs are safe to be executed in a sandbox. The second technique uses provable guarantees for safety, levering the safeguards present in the Rust design to bring security and performance to the compiler. The runtime does not have a performance impact and security was improved. *RLBox* is a framework for sandboxing third-party libraries that may offer some danger to browser security (Narayan

[3]consist of behavior that can be triggered because of the porting of the application, or interactions with other resources in the environment (Lehmann et al., 2020).

Category	Threats	Countermeasures
Buffer Overflows		(Michael et al., 2023)
XSS	(McFadden et al., 2018; Lehmann et al., 2020)	
Format String	(McFadden et al., 2018)	
Heap Metadata Corruption	(Lehmann et al., 2020)	
Integer Overflow/Underflow	(McFadden et al., 2018)	(Michael et al., 2023)
Stack Based Buffer Overflows	(McFadden et al., 2018; Lehmann et al., 2020)	
Stack Overflow	(Lehmann et al., 2020)	
Overwriting Constant Data	(Lehmann et al., 2020)	
Overwriting Heap Data	(Lehmann et al., 2020)	
Overwriting Stack Data	(Lehmann et al., 2020)	
Server-side Remote Code Execution	(McFadden et al., 2018)	
Side-Channel	(Genkin et al., 2018)	(Narayan et al., 2021; Protzenko et al., 2019)
Redirect Indirect Calls	(Lehmann et al., 2020)	
Code Injection	(Lehmann et al., 2020)	
Porting Programs & Compilers	(Stiévenart et al., 2022b; Romano et al., 2021)	
Constant-Time		(Mazaheri et al., 2022; Tsoupidi et al., 2021; Watt et al., 2019a; Renner et al., 2018)

Table 4.1: Overview of WebAssembly Flaws and Vulnerabilities used in Attacks

et al., 2020). Two mechanisms of isolation were evaluated: Software-fault Isolation (SFI) and multi-core process-based. The performance impact of the solution is modest, with the highest value being 13% in latency and 25% in memory use.

Spectre is an attack that can bypass WebAssembly security measurements (Kocher et al., 2020). *Swivel* is a framework developed to protect against this type of attack, through hardening the the security of the WebAssembly sandbox against breakout and poisoning attacks (Narayan et al., 2021). The proposals have an overhead, however, they offer a more secure sandbox environment by using stronger memory isolation.

Some authors have explored the use of trusted execution environments, such as Intel SGX (Will et al., 2021), to improve the security of the WebAssembly execution environment:

- *AccTEE* is a two-way sandbox framework that offers security in remote computation cases (Goltzsche et al., 2019). The protection exists with the use of Software Guard Extensions (SGX) and WebAssembly sandbox for securing the execution and data in the environment. Besides the insurance with integrity and confidentiality, the solution model offers protection against malicious and untrusted code.

- TWINE is a runtime design that explores SGX and the WASI interface to create a trusted environment to execute unmodified applications that were already compiled to WebAssembly (Ménétrey et al., 2021). The proposal is a Trusted

Execution Environment (TEE) that uses the WebAssembly sandbox to abstract the environment from the application. This proposal brings a more secure runtime environment for cloud computing, in addition to presenting a detailed performance comparison.

- *Se-Lambda* is a serverless computing framework that focuses on security in the cloud computing environment (Qiang et al., 2018). The two-way sandbox solution protects the API gateway, user data, and cloud platform. The framework presents a low performance impact and, since the prototype is based on an open-source project, new features could be added following new improvements in WebAssembly.
- An enclave design based on WebAssembly is presented by (Pop et al., 2022). The proposal enforces integrity and confidentiality, offering a secure channel for Wasm services to migrate between different architectures.
- *WaTZ* is a remote attestation and runtime for WebAssembly (Ménétrey et al., 2022). The WebAssembly sandbox isolation prevents privilege escalation attacks, with some performance impact.
- *Edgedancer* is a platform for mobile edge computing that uses WebAssembly to ensure security and allows self-migration inside the infrastructure (Nieke et al., 2021). The platform also takes advantage of TEE to improve security. Small services present better performance during migration with a higher security standard.

Fuzzing is a technique for software testing that aim to find weaknesses, flaws, and vulnerabilities of design through the generation of input tests (Liang et al., 2018). This strategy allows the monitoring and study of the behavior of an application:

- *WAFL* is a fuzzing tool for WebAssembly binary that uses AFL++, a well-known fuzzer (Haßler and Maier, 2021). The solution connects a WebAssembly application that is being fuzzed with AFL++. Therefore, the Wasm binary is able to use AFL++. The solution does not harm the performance of compilers.
- *Fuzzm* is a fuzzer for WebAssembly format, mainly focused on the detection of vulnerabilities in memory (Lehmann et al., 2021). Through the implementation of canaries, the fuzzer uses inputs from AFL to detect memory overflows/underflows that could be explored in the stack and heap. The fuzzer only requires the binary.
- The work (Chen et al., 2022) presents *WASAI*, a fuzzer for the identification of vulnerabilities in smart contracts implemented in WebAssembly. The study found that, in a set of 991 samples, over 70% had some type of vulnerability.

Taint analysis traces the the flow of untrusted data input through an application during execution (Ming et al., 2016). This is strategy is important to check for policy violations and unsafe operations.

- Taint tracking of WebAssembly applications is presented in (Szanto et al., 2018). Through a WebAssembly Virtual Machine (VM) implemented in JS, the framework allows a taint tracking strategy for the study of the data sensitivity of the bytecode. The evaluation describes an acceptable memory overhead, with the framework being limited by only some missing functionalities presented in WebAssembly.

- *TaintAssembly* is a framework for taint tracking of WebAssembly, allowing the study of interactions between WebAssembly and JS (Fu et al., 2018). The framework has a performance overhead, suffers from design limitations that require random number generation, and does not implement comparison operations.

SELWasm is a framework to protect WebAssembly code intellectual property (Sun et al., 2019). The framework makes protections against the reuse of source code without authorization, using encryption to ensure that attackers are unable to obtain the source code, and a lazy loading strategy for optimization.

BLADE is a framework to protect against speculation leaks from cryptographic algorithms (Vassena et al., 2021). The framework is implemented in the WebAssembly compiler and allows the evaluation of its implementation through vulnerable WebAssembly implementations. The solution does not require supervision and can be effective in other languages. The paper (Tsoupidi et al., 2021) presents *Vivienne*, a tool for the analysis of cryptographic libraries implemented in WebAssembly. The proposal implements constant time verification in WebAssembly. The study evaluates the implementation with 57 real-world libraries, proving the effectiveness of the proposal.

In the literature, a set of works focuses on the application of WebAssembly to improve the IoT environment.

- *Aerogel* is a framework to better secure access control between bare-metal and IoT applications using the WebAssembly environment, addressing security gaps previously not approached (Liu et al., 2021b). The framework uses the WebAssembly sandbox to enforce access control measures, with a performance overhead of 1% and the growth of energy consumption reaching almost 46%.

- *Wasmachine* is a OS mainly focused on the security of WebAssembly for the execution of applications in IoT and fog computing devices (Wen and Weber, 2020). Through the execution of WebAssembly binary in kernel mode, the cost of execution is reduced. The paper also implements the OS kernel in Rust to ensure memory security.

- *ThingSpire* is a cloud-edge OS based on WebAssembly (Li et al., 2021). The proposal approach three challenges related to the development of the infrastructure; (I) how to design the environment effectively; (II) how activate intercommunication between modules; and (III) how security and fault tolerance are supported.

- *MEWE* is a framework that presents a technique for edge-cloud computing (Arteaga et al., 2022). The framework adds randomization for WebAssembly runtime execution and diversifies Wasm binaries. The combination of these two strategies hardens attacks as Break-Once-Break-Everywhere (BOBE). With real-world data, it was possible to identify flexibility to generate binary variants.

WebCloud is an encryption solution for communication between cloud services and browsers (Sun et al., 2022). The client-side application solves limitations related to the security and access control of previous solutions.

Aiming in understanding the vulnerabilities behind WebAssembly technology for cross-platform applications, the work (André et al., 2022) focuses on developer's questions found in StackOverflow[4] to better understand security issues. More than half of the

[4]StackOverflow is a web site that focus in questions and answer (stackoverflow, 2023).

questions are directed to information about features or bugs. Authentication is the group of most prevalent questions.

Considering the security impact of compiling applications from C to WebAssembly, and exploring what type of behavior could be expected, the authors in (Stiévenart et al., 2021) present a study focusing on evaluating the vulnerabilities/divergences that are ported from C after the compilation process is done. In a dataset with 4.4 k samples, 24% of applications tested had a different outcome, because of a lack of security measurements as stack canaries when ported to WebAssembly.

4.2.3 Analysis and Protection for WebAssembly

The analysis of WebAssembly applications allows developers to have a better understanding of the environment in which the applications will be executed, offering information about interactions with other languages, performance impacts, and security. Protection measurements could be taken after this type of study.

- (Romano and Wang, 2020a) presents *WASim*, a tool for the classification of WebAssembly modules. Eleven categories were used to train four machine learning models. The dataset was extracted from the Alexa 1 million, with the models reaching an accuracy of 91.6%.
- WebAssembly Symbolic Processor (WASP) is the focus in (Marques et al., 2022). These strategies allow developers to find bugs and security flaws, through the analysis of multiple program paths.
- *Oron* is an instrumentation platform that was implemented using WebAssembly to tackle *Linvail* performance issues (Munsters et al., 2021). The platform was evaluated through the instrumentation of AssemblyScript code. The evaluations show less performance overhead in comparison with other solutions.
- *Wasmati* is a static analysis tool presented by (Brito et al., 2022), focusing on finding vulnerabilities in Wasm binaries. *Wasmati* was developed to be used in the development stage, being limited to binaries based on Emscripten. The technique used for the evaluation consists of Code Property Graph (CPG) that includes information on the code being analyzed as property-value pairs.
- (Lehmann and Pradel, 2019) introduces *Wasabi*, which is a framework for the dynamic evaluation of WebAssembly. The first contribution consists of a survey of approaches with dynamic analysis found in the literature. *Wasabi* takes advantage of the WebAssembly implementation to extract information at runtime, besides not being limited to only applications implemented in WebAssembly.
- (Stiévenart and De Roover, 2020) uses analyze Wasm binaries through isolating segments of the code. The evaluation focuses on developing a flow analysis of the WebAssembly applications. Despite a low precision of 64%, the contribution of the paper consists in a static analysis that allows the evaluation of functionalities of WebAssembly code.
- *EOSafe* is a framework that analyzes Wasm binary to find EOSIO[5] smart contracts vulnerabilities (He et al., 2021). The detection strategy relies on heuristics and

[5]EOSIO is a blockchain protocol (EOSIO, 2023).

the limitation of symbolic execution. Four of the most popular vulnerabilities for EOSIO are explored, with the models achieving an f1-score of 98%.

- *WasmView* is a framework to evaluate/visualize the interaction between JS and WebAssembly (Romano and Wang, 2020b). The framework allows the understanding of function calls between the languages and stack traces of applications, through the presentation of a visual call graph and trace logs of the information captured.

4.2.4 Proposals for WebAssembly

As discussed in section 4.2.1, despite the security focus behind the WebAssembly design, flaws and vulnerabilities exist and are being explored in the wild. However, there is a set of works that aim to improve the security of WebAssembly, presenting countermeasures with improvements and proposals to increase security in the WebAssembly environment. This section aims to discuss these works.

Constant-Time WebAssembly (CT-Wasm) is a expansion of WebAssembly compiler, mainly focusing on cryptographic security (Watt et al., 2019a). CT-Wasm permits the verification of security properties and is secure against side-channel attacks. Data protection enables developers to improve security considering the information flow. (Renner et al., 2018) also focuses on the security limitation of constant-time in WebAssembly for cryptographic primitives. To achieve this goal, the WebAssembly compiler was modified, with the adding the support of the sensitivity of a variable type being added. The expansion of the validation process with the addition of type checker functionalities, and the runtime environment was modified to ensure security. The verification of cryptographic primitives in WebAssembly is the focus of (Protzenko et al., 2019). A toolchain to compile F*[6] to WebAssembly is implemented, this delivers the option of using platform libraries via WebAssembly. The validation process consists of compiling HACL*[7] to WebAssembly and verfying protection against side-channel attacks.

Gobi is a SFI for WebAssembly, aimed at addressing limitations in WebAssembly sandboxing (Narayan et al., 2019). The system groups compilers and changes in the runtime that allow libraries implemented in languages such as C/C++ to be compiled to WebAssembly. This prototype of WebAssembly SFI was incorporated by the Wasm community by the creation of WebAssembly System Interface (WASI) and improvements to the Lucet toolchain (however, in 2020 focus was changed to the Wasmtime engine (BytecodeAlliance, 2023)).

The contribution of (Watt et al., 2019b) is directly linked to the development of the Wasm standard, presenting memory and operation extensions for WebAssembly and JS. The WebAssembly extensions allow the use of threads, atomics, and mutability. The memory model ensures sequential data use and shared memory concurrency is guaranteed for future implementations.

WebAssembly had issues related to memory that still happen inside the sandbox (Section 4.2.1). Memory-safe WebAssembly (MSWasm) is an extension for memory security presented by (Michael et al., 2023). The formal specification defines the use of segmented memory for memory security. Two implementations are discussed, the MSWasm for memory safety in WebAssembly compilers and a C to MSWasm compiler that guarantees

[6] F* is a general-purpose, proof-oriented programming language (fstar-lang, 2023).
[7] HACL* is a cryptographic library written in F* (HACL*, 2023).

memory safety from unsafe sources. The extension when evaluated presented an overhead of 197.5%.

(Vassena and Patrignani, 2020) discusses memory issues found in WebAssembly, and points out known solutions that could be explored in each problem. Through the use of the MS-Wasm extension, the proposal can guarantee memory safety. An extension of WebAssembly is presented in (Disselkoen et al., 2019), which presents a design to ensure spatial and temporal safety and pointer integrity. With explicit memory safety at the language level, the proposed WebAssembly implementation ensures memory safety.

(Kolosick et al., 2022) modify the WebAssembly sandbox to guarantee better performance and security. The Wasm sandbox uses heavyweight transitions that weigh the runtime, changing this to zero-cost conditions also achieves the same level of security guarantee with better performance. Through a checker named *VeriZero*, it is possible to identify when a function is semantically capable to explore this feature.

WebAssembly suffers from memory consumption and start-up time, a design for an effective solution is the focus in (Titzer, 2022). The proposal uses a compact side table generated during the validation process to provide better performance performance for Wasm.

SecWasm is a hybrid Information-Flow Control (IFC) system design for the confidentiality of information in WebAssembly application (Bastys et al., 2022). The solution overcomes structured control flow and linear memory problems with only a performance overhead.

(Arteaga, 2022) presents a solution to mitigate software monoculture in WebAssembly. Randomization and multi-variable execution were proposed for the Code Randomization of WebAssembly (CROW) and Multi-variant Execution for WebAssembly (MEWE) frameworks.

4.2.5 General Applications

The use of WebAssembly to solve problems found in a variety of scenarios is considered in this section. Since WebAssembly offers a range of features, different applications can explore this technology for performance and security gain.

In the context of IoT security, (Radovici et al., 2018) propose a framework that focuses on running bytecodes in an isolated WebAssembly sandbox. The proposal mainly depends on the WebAssembly environment, with a focus on a design to securely expose the required resources to run an application. It is expected that the proposal contributes to the security and performance in the IoT scenario.

Considering edge computing and the demand for performance in these resources/environments, (Koren, 2021) presents a proof-of-concept of the use of WebAssembly for microcontrollers. A web server of an Integrated Development Environment (IDE) was developed to deliver new modules to edge computing devices and the cloud.

(Baumgärtner et al., 2019) presents an alternative for Disruption-Tolerant Networking (DTN). The network is established through a combination of Bundle Protocol 7 and WebAssembly. A web application that is enabled by the browser will carry out the exchange of messages.

Sledge is a serverless framework using WebAssembly for edge computing (Gadepalli et al., 2020). It offers lightweight function isolation in the runtime implemented in WebAssembly and scheduling policies for the infrastructure. Results have shown a low latency and efficient management of concurrency.

WasmAndroid is a cross-platform runtime for native languages in Android that only requires the compilation of the code to WebAssembly (Wen et al., 2021). The model is 1.3x slower than the benchmark suite SPEC CPU 2006.

WasmTree is a Resource Description Framework (RDF) implementation that explores WebAssembly and Rust to optimize and gain performance in the Web environment (Bruyat et al., 2021). The use of WebAssembly did not show a gain in performance. However, the evaluation with the SPARQL query showed performance improvement.

(Stiévenart et al., 2022a) propose a static strategy for slicing WebAssembly programs. The study aims at a static intra-procedural backward slicing approach for WebAssembly. The proposal is evaluated with real-world applications.

(Ménétrey et al., 2022) presents a position paper for the development of applications that could run in a set of hardware devices without loss of performance and security. The study focuses on describing the impact that the use of WebAssembly can bring to this medium.

An evaluation of the use of Ethereum Virtual Machine (EVM) and WebAssembly for smart contracts is the focus in (Zheng et al., 2020). The use of WebAssembly in the Ethereum blockchain clients presents issues related to support, development limitations, and instability. The performance of WebAssembly varies significantly.

Hector is a web application framework that enables distributed applications in the web browser (Goltzsche et al., 2020). WebAssembly is one of the components that guarantee integrity, confidentiality, and isolation.

EVulHunter is a tool developed for the detection of EOSIO WebAssembly vulnerabilities, mainly focused on fake-transfer attacks (Quan et al., 2019). Two strategies are used for the static analysis, (I) predefined functions; and (II) comparison operations. *EOSDFA* is a framework for the analysis of the EOSIO smart contracts (Li and Zhang, 2022). The framework expands the automatic deployment system known as Octopus framework, allowing the evaluation of pointer access for control flow and dataflow analysis.

ZAWA is a virtual machine that emulates WebAssembly bytecodes (Gao et al., 2022). Aiming at a more secure runtime, ZAWA takes advantage of Zero-Knowledge Succinct Non-Interactive Argument of Knowledge (ZKSNARK), to evaluate if any leakages are happening.

4.2.6 Datasets available

The availability of data for experimentation helps researchers to compare results, improve previously published research, and develop new approaches. Table 4.2 presents the currently available WebAssembly datasets, describing the type of data, number of samples, extraction methodology, and year of creation.

Overall, despite the limited amount of datasets that may be used in security studies, the most recent samples are aiming to represent the current Wasm ecosystem. It is important to notice that these datasets are not being updated with newer samples. This can be considered risky, considering that WebAssembly is still experiencing major changes and newer, more representative data will be required in the future.

The Alexa rank (Amazon, 2023) is a recurrent source of data for experiments not only in WebAssembly. In the data collection presented by (Kim et al., 2022), only one approach does not explore the Alexa rank. However, the use of the Alexa rank for security research is not adequate, since the method used for data gathering varies widely and the lack of details may affect research results (Le Pochat et al., 2018; Ruth et al., 2022). Some

Dataset	Purpose	Samples	Extraction	Year
WasmBench (Hilbig et al., 2021)	A real-world dataset representation	8.4 K	From a variety of sources, such as: GitHub, package manager, websites, and manually	2021
SnowWhite (Lehmann and Pradel, 2022)	Recovering of WebAssembly types	6.3 M	4 k samples from from C and C++ Ubuntu source code packages	2022
(Chen et al., 2022)	Identification of Wasm smart contract vulnerabilities	3.3 k	Vulnerabilities and smart contract in the wild	2022
QRS (Stiévenart et al., 2021)	C applications with different behavior when compiled to WebAseembly	1,088	Vulnerable C code from Juliet Test Suite 1.3 (2017)	2021
SAC (Stiévenart et al., 2022b)	Present Wasm binary with security issues from porting WebAssembly from C	4.9 K	Vulnerable C code from Juliet Test Suite 1.3 (2017)	2022
Alexa (Amazon, 2023)	A ranking of 1 million of the highest traffic domains globally, updated daily	Variable	Traffic data is ranked by domain. Data is collected though a browser extension used by users	2008-2022
Base on Alexa (Musch et al., 2019)	Define the use of WebAssembly on the web	150	WebAssembly module from webpages was collected, with 1950 samples (150 unique) from 1,639 websites	2019
Question dataset (André et al., 2022)	WebAssembly security related questions made by developers	359	Security questions from Stack OverFlow	2022
Bug dataset (Romano et al., 2021)	Emscripten bug collection	1,054	Bugs were extracted from the Emscripten project	2021

Table 4.2: WebAssembly Datasets

alternatives that are more adequate to represent the Web environment for security research are discussed in (Xie et al., 2022; Le Pochat et al., 2018; Durumeric, 2023).

It is also important to consider the type of information each dataset provides. For the extraction of information from binaries, most of the samples available in the datasets presented can be used. However, for an evaluation requiring the execution of the binaries, restrictions related to missing dependencies from the period of data extraction or even support of the data, limit the use of that dataset. An alternative, in this case, would be to use benchmark tools and test suites that are known to work in compilers.

4.2.7 Wasm Current Limitations

Despite its security focus, some design choices in WebAssembly end up limiting security measures found in other languages that can be compiled to WebAssembly. In some cases, these mechanisms will not be present when the code is compiled from another language to Wasm (McFadden et al., 2018).

Address Space Layout Randomization (ASLR) is a memory protection that, despite being present in other languages, would be difficult to be implemented in WebAssembly, considering the design of the linear memory (McFadden et al., 2018). In the future, this protection could be added (W3C Community Group, 2022a). However, the linear memory also removes the requirement of some protection as stack canaries.

The WebAssembly design also removes the requirements of resources as Data Execution Prevention (DEP), since low-level instructions will not have the same access to resources thus this protection is not required.

Heap Hardening is presented in some compilers and solutions were proposed in the literature. Also, compilers that support Control Flow Integrity (CFI) can export the protection to the compiled Wasm (McFadden et al., 2018).

4.2.8 Open Research Topics

Discussions about WebAssembly vulnerabilities are evolving and it is possible to notice changes made by the community based on these findings. However, in practice the malicious use of WebAssembly has been limited so far. We saw studies on how interactions between WebAssembly and resources presented in the environment could be helpful for attackers and the impact of the binary format for obfuscation. Future studies should be more aligned in evaluating the current use of WebAssembly by attackers, the effectiveness of the WebAssembly sandbox, and the real advantages in specific areas such as IoT, Trusted Computing (TC), and blockchains, in which we observe wide applicability of WebAssembly.

Studies that aim at describing the adoption and use of WebAssembly applications in the real-world are still limited, with many of them not representative of the current state of the Internet. Discussion of WebAssembly adoption, even considering the security point of view, also presents the same problems. These problems are directly associated with (I) the use of outdated databases, which in some cases were already shown to not be representative; (II) lack of samples, validation strategies, and worries about the reproducibility of the data used in future studies. The availability is considered, however, the usefulness of the data for future studies is not considered.

Attestations of WebAssembly format security measures are not studied. The security discussion in the literature could be expanded for the WebAssembly design/development, aiming at a better understanding of the security of the format. A similar discussion would be helpful to clarify the impact of porting from some languages to WebAssembly, where several security discussions are focusing.

4.2.9 Current developments in WebAssembly

The advances made by the community in the latest years were massive and future proposals could give us a glimpse for future discussions. Future proposals that are interesting considering the current security state are: (I) the addition of support for tail calls, (II) multiple memory, (III) garbage collection, and (IV) threads (WebAssembly, 2023b).

4.3 DISCUSSION OF FINDINGS

Research on WebAssembly security has grown between 2021 and 2023, more than doubling in comparison with works published between 2017 and 2019. The articles are mostly published in conferences, symposiums, and workshops. The topic also became more open, not being limited to problems related to format design. Between 2021 and 2023, some of the research focus concerned security solutions, behavior detection, and the use of WebAssembly features to bring a more secure environment to a variety of systems. Next we discuss the RQs defined for the review.

RQ1: What security issues are encountered when using WebAssembly?
Overall, security issues will exist in any format, as bugs/flaws can occur due to the way features and implementations are made by developers. In WebAssembly, attackers could explore vulnerabilities or design flaws that facilitate malicious operations, despite the security involved in WebAssembly design. Most of the security issues discussed in this chapter are related to some design limitation or choice.

The linear memory is one of the design highlights in WebAssembly, isolating the application, since memory accesses are limited to a specific region (Kim et al., 2022). However, despite this design, memory vulnerabilities persist due to the lack of security safeguards in languages that can be ported to Wasm. Not being limited to memory, one of the most attractive features of WebAssembly could be one of the most dangerous ones. Portability from a range of languages impacts directly the security of the Wasm binary, since a range of languages implement different safeguards that are not present in WebAssembly and, in many cases, may be carried over when generating a Wasm binary.

The security flaws/vulnerabilities discussed in the literature (section 4.2.1) explore a vast range of elements. However, most of these problems are not limited to WebAssembly, so emphasizing this problem as a constraint on the use of WebAssembly is not adequate. Considering the current state, advances were made for a more secure environment, and solutions have been proposed to patch security flaws/vulnerabilities.

RQ2: What has already been developed to improve security in WebAssembly?
Security measures for WebAssembly are not limited to a specific problem. The improvement of security is being achieved by new proposals for the WebAssembly design, frameworks that improve security, or strategies of evaluation for Wasm binaries.

Improving the environment security is being achieved by changes in the sandbox, exploring resources such as SGX and API to harden the WebAssembly runtime safeguards. For the development strategies with fuzzing, tracing, and taint were developed to demonstrate how resources interact in an application. It is also possible to point to the development of analysis and protection strategies for WebAssembly applications (Section 4.2.3).

New proposals are approaching major problems of WebAssembly with improvements in format design. These improvements are being used by the community to make changes in the format to guarantee a more secure environment.

RQ3: How WebAssembly is being used to carry out attacks? As discussed in Section 4.2.1, a range of attack options are possible. However, WebAssembly mainly helps attackers by (I) providing a binary format that may help in code obfuscation (unlike source-only languages such as JS); (II) the performance gain of WebAssembly in comparison with other languages in the same environment makes WebAssembly attractive for cryptojacking; (III) as in any language, vulnerabilities/flaws are attractive; (IV) being a resource supported for most web browsers and the proximity with JS, malicious WebAssembly applications become interesting.

RQ4: What is the current status of the WebAssembly compilers? Support for the WebAssembly format is reaching most of the major languages, and a range of compilers are available (Stephen, 2022). The adoption of the format is also high, with all of the major browsers having support for most of the features. Proposals also present the same rate, with constant releases of new features. Wasmtime and Emscripten are projects where new features are added frequently and have an active community.

WebAssembly security research focuses on niches around the format design, the detection of development errors in binaries, and the sandbox environment. With a range of proposals for the improvement of the security found in the WebAssembly format. However, the detection of anomalies inside the WebAssembly sandbox is not being explored. Since the WebAssembly it is also being explored to improve security solutions with other technologies such as SGX and IoT environment. The detection of anomalies has become even more relevant, since the WebAssembly sandbox adoption for the improvement of security.

5 AN ANOMALY DETECTION SOLUTION FOR WEBASSEMBLY

This chapter defines the research problem and provides an overview of the proposed solution and how the research was conducted. Section 5.1 states the problem. Section 5.2 outlines the solution. Section 5.3 describes how data categorization is used in our solution. Section 5.4 presents the methodology used for developing and evaluating the solution. Section 5.5 discusses system calls and WASI calls. Section 5.6 concludes the chapter.

5.1 PROBLEM DEFINITION

Computer security requires dealing with an ever-changing landscape of technologies and threats. New technologies provide new avenues for attacks, especially in early stages of adoption, when the technology is evolving rapidly and knowledge about how to use and deploy it securely has not been cemented or disseminated enough. Another relevant aspect of new technologies is that they may be affected both by already-known threats adapted from other environments and novel threats that leverage specific characteristics of the technology.

WebAssembly is an example of a new technology that has been gaining adoption. Its availability in popular web browsers means that WebAssembly-based attacks have the potential of affecting a significant number of users. This implies that WebAssembly security is a topic of great importance, especially given that, as discussed in Chapter 4, such attacks have already been documented and the technology still has some security gaps, noticeably in the detection of vulnerable and malicious applications (Section 4.3).

Thus, the research problem explored in this work is how to detect malicious behavior in WebAssembly applications.

5.2 ANOMALY DETECTION SOLUTION

Our proposal for tackling the research problem involves applying anomaly-based intrusion detection to identify malicious behavior in WebAssembly applications. More specifically, the idea is to monitor the WASI calls (Section 4.2.2) issued by an application, as these calls correspond to the interactions between the application and the environment.

WASI calls provide rich information about the security-relevant behavior of an application. Operations with security implications usually involve access to resources outside the application (e.g., files, network, user interface), which requires WASI calls. Prior work in performing intrusion detection by observing the interactions between applications and the operating systems using system calls (Forrest et al., 1996; Bernaschi et al., 2002) and Binder calls (Lemos et al., 2023) has shown this approach to be effective.

Figure 5.1 provides an overview of the proposal. We use machine learning models for anomaly detection. To apply such models, we record the WASI calls issued by WebAssembly applications and transform these data into a format suitable for processing. Our proposal introduces a categorical data representation for WASI calls. Based on previous work, we developed classification schemes for WASI calls according to functionality and threat level, as detailed in Section 5.3. The exact WASI calls issued by an application are substituted by their categorical representation, which combines the functionality and threat classifications. These categorical data are then fed into a previously trained ML

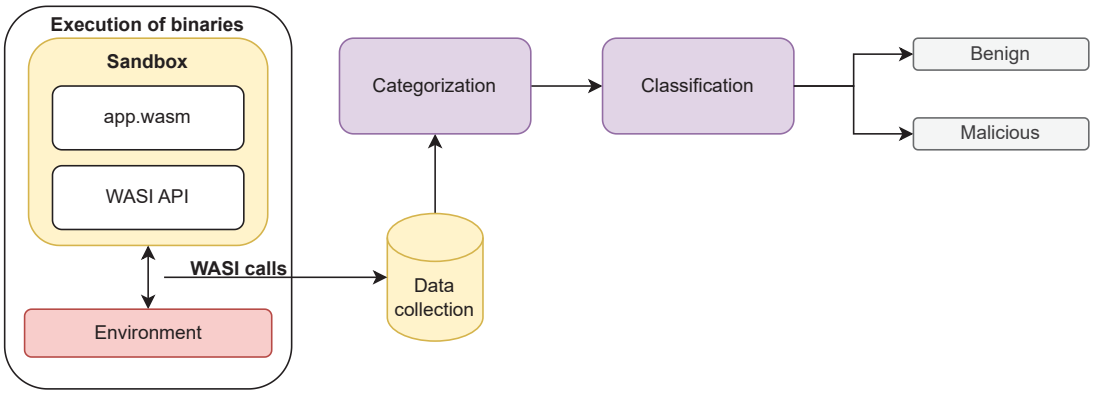


Figure 5.1: Overview of our proposal for intrusion detection using WASI calls.

classifier, which indicates whether the application is benign or malicious. The figure omits the classifier's training phase; Chapter 6 describes how the classifiers were trained in our experimental evaluation, and discusses the results obtained. It should be noticed that our approach is suitable for both on-line and off-line intrusion detection.

The collection of WebAssembly System Interface (WASI) calls made by WebAssembly applications is dependent on the compiler used. Although the available compilers adhere to the WASI standard, they provide different features. In our experiments we used Wasmtime, since it already has many resources for tracing running applications (BytecodeAlliance, 2021a). Wasmtime also offers a feature dedicated to watch the WASI calls made by an application. This option eliminates the need to change the compiler's source code to obtain the data.

5.3 DATA CATEGORIZATION

Data categorization allows characteristics of operations and functionalities to be associated with groups. These categorizations can later be used in the learning process, to highlight characteristics that would not be present in the raw data. This strategy contributes to classification models where categorical features can be used in the learning process.

The proposed classification aims to be applied to the classification of WASI calls. However, given the semantic proximity between WASI calls and system calls, a classification that works for both types is interesting, as it facilitates an eventual comparison between them. Two classifications are proposed here, one considering the threat associated with a call and the other considering the functionalities implemented by the calls.

The classification scheme proposed by (Bernaschi et al., 2002) is a pioneer work considering system call categorization. In their scheme, system calls are classified independently according to two criteria: functionality and threat level. The threat levels are 1 (enable full system compromise), 2 (allow a denial-of-service attack against the system), 3 (enable subversion of the calling process), and 4 (harmless calls). This classification is hierarchical, as it assumes that a system call classified at level i can also perpetrate an attack at threat level j for $i < j$. A problem with this classification is treating threat levels as an hierarchy, with threats at lower levels subsuming threats at higher levels, which may not always be true. A counter example would be the `fork` system call, which can cause a Denial of Service (DoS) on the system (and is thus classified at threat level 2) but cannot subvert a process (threat level 3).

In our proposal, we remove the hierarchical constraint from the classification defined by (Bernaschi et al., 2002) and propose a categorical definition that is flexible for using with other types of data.

Each WASI call has specific information related to its operation, functionality, and security. Each interaction (or operation) that an application performs with the system can be categorized. Also, each interaction also has a respective threat associated to it.

Table 5.1 presents the classification according to the threat offered by WASI calls. The five levels were defined based on the observation of WASI calls in the set of Wasm code samples used in the experiment (Chapter 6). The level represents correlation, not causation. Therefore, it is not because a code has a call classified in one of the levels considered high threat (A, B or C) that it is malicious/vulnerable. On the other hand, an application that uses several calls in class A (for example) tends to pose a greater threat than an application that only uses calls in class D.

Threat Level	**Threat Group**	**Type of operation**
A	High	calls that alone appear more frequently in malicious/vulnerable code than in benign code
B		calls that were used together more often in malicious/vulnerable code than in benign code
C		calls that, in malicious/vulnerable code, are repeated many more times than in benign code
D	Low	calls that are used in the same way in malicious/vulnerable code and in benign code, and whose semantics allow us to assume that they cannot cause a security breach
E		unused/deprecated calls

Table 5.1: Classification according to threat level.

The five threat levels presented aim to group interactions (that in our proposal are WASI calls) that alone pose some kind of threat to the system if used by a malicious application. When considering the defined threat group (high or low), we tend to directly relate it to the type of operation that the interaction performs in the system. The first three threat levels (A, B, and C), in the *high threat* group, are interactions that can be used by an attacker for malicious purposes. Such threat levels define different behavior of operations that malicious applications perform in the environment.

We also propose a classification based on the functions performed by each interaction with the runtime (WASI call), shown in Table 5.2. This classification is an expansion of categorizations already proposed, such as (Bernaschi et al., 2002) and (Galvin et al., 2003). We added the removed/debug (9) and the device manipulation (10) groups, since several changes were observed in modern operating system APIs. The separation is justified because current OSs have a richer set of calls for manipulating devices, and the distinction between files and devices is more pronounced in WASI calls. Another change was in group 9, which contained only unimplemented calls and now also includes calls removed or used for debugging kernel.

We assume that an application that frequently makes calls classified in class A tends to pose a higher threat than an application that mostly uses calls presented in class D. This classification is not used alone in the intrusion detection process, but together with

Group	Functionalities
1	File manipulation
2	Process control
3	Module management
4	Memory management
5	Time operation
6	Communication
7	System information
8	Reserved
9	Not implemented/removed/debug
10	Device manipulation

Table 5.2: Classification according to functionality.

the classification of functionalities. In this way, an application that makes calls classified as A-2 (a process control call at the highest threat level) poses a greater threat than if it makes calls classified as C-5 (time-related calls at a medium threat level). This distinction contributes to the training stage of the Machine Learning (ML) models, as it is possible to emphasize calls that pose more threat.

Table 5.3 presents the proposed data categorization for the WASI calls[1]. WebAssembly is a format in active development, thus we are using the calls supported for version 1.0. In future versions currently under development (`snapshots`) new calls will be introduced, but this proposal already has the required categories for the correct classification of such calls.

Level	Gr	WASI calls
A	1	`path_link`, `path_rename`, `path_symlink`, `path_unlink_file`, `path_remove_directory`, `path_filestat_set_times`, `args_get`, `environ_get`
B	1	`fd_fdstat_set_flags`, `fd_tell`, `fd_seek`, `path_create_directory`, `fd_pread`, `fd_pwrite`, `sock_recv`, `fd_read`, `sock_send`, `fd_write`, `fd_filestat_get`, `path_create_directory`
	2	`fd_advise`, `sched_yield`
C	1	`fd_renumber`, `fd_allocate`, `path_open`, `random_get`
	6	`sock_recv`, `sock_send`, `proc_raise`
D	1	`fd_close`, `path_readlink`, `path_filestat_get`, `args_sizes_get`, `environ_sizes_get`, `fd_prestat_get`, `fd_prestat_dir_name` `fd_readdir`
	5	`clock_res_get`, `clock_time_get`
	7	`sock_shutdown`
	9	`fd_filestat_set_times`
	10	`fd_datasync`, `fd_sync`, `fd_fdstat_get`

Table 5.3: WASI Calls classification.

[1]A classification for system calls and WASI calls is presented in Table C.1.

We note that threat level E (Table 5.1) and groups 3, 4, and 8 (Table 5.2) are not used in Table 5.3. We chose to keep these classes for three reasons: (i) to facilitate correspondence with the original classification of (Bernaschi et al., 2002); (ii) to allow us to apply the same categorization to system calls (as shown in Appendix C); and (iii) to be ready for future WASI calls (which are likely to provide functionalities that are similar to those of existing system calls).

The advantages of the categorization proposed here are (i) it is flexible enough to be applied to other types of data; (ii) it reduces the volume of data used for training and identification; (iii) it allows a better understanding of the behaviors behind application interactions; and (iv) it allows to relate different types of interactions.

5.4 DEVELOPMENT AND TEST

To develop an anomaly detection strategy for WebAssembly using categorical data a methodology was defined. The following steps are needed to achieve these goals:

- **Dataset definition:** The data selection must include different types of applications, to cover the widest possible range of application behaviors. The use of different data sources helps to improve data heterogeneity;
- **Data collection:** Since we focus on the interaction between the WebAssembly application and the host environment, the WebAssembly binaries must be executed in their sandbox environments, with the interaction being collected and saved for posterior evaluation. Each application will generate one trace, that stores the executed interaction;
- **Data categorization:** The data collected must be categorized considering the definition presented in Section 5.3. A set of machine learning models is selected, considering previous experiment and use in security solutions (Galante et al., 2019; Castanhel et al., 2020; Lemos et al., 2023);
- **Experimentation:** The data is then used for the training and testing of the machine learning models; and
- **Data analysis:** The evaluation focuses on assessing the impact of using categorical data for intrusion detection solutions.

5.5 WASI CALLS VS SYSTEM CALLS

WASI calls have similar functionality to system calls, but the mapping of calls is not one-to-one. Since WASI calls are closer to the application, they are planned to work with specific resources. A system call could be used by more than one WASI call and a WASI call may trigger more than one system call. Consequently, the WASI calls are more specific to better use the sandbox environment. However, considering the information about the operations of both types of calls, similar characteristics are expected.

As WASI calls are at a level closer to the application, they also require operations that would not exist when working with system calls. Some of these operations are associated with input/output and parameter handling, which could later be translated into a set of system calls.

The vast majority of Antivirus (AV) solutions are not capable of protecting the environment against malicious content that runs inside virtual machines such as Java Virtual Machine (JVM) (Botacin et al., 2021) and the WebAssembly sandbox. Our proposal offers an alternative for protecting such environments. Observing WASI calls is interesting for detecting behaviors for two main reasons:

- The proximity that a WASI call has to the application, which tends to help in the identification of behavior since a translation turns out to be clearer; and
- The similarity of these WASI calls to system calls, which have already proven to be useful for observing behavior.

Anomaly detection solutions will have similar results, between system calls and WASI calls, since the same information will be presented in different points of view. However, an overhead could exist considering the level at that system calls are being collected, in some cases making it difficult to identify threats. Since we are using a categorical data approach, both strategies will present similar models for the machine learning training.

5.6 CONSIDERATIONS

The anomaly detection proposal presented in this chapter focuses on the use of categorical data to represent the behavior of WebAssembly applications. As presented in Table 5.3, the corresponding classification can be used for a variety of types of data. After an in-depth evaluation of the representation a better understanding of the use of categorical data for security and WASI calls for anomaly detection in WebAssembly is expected.

6 EVALUATION OF WASI CALLS FOR ANOMALY DETECTION

The chapter presents the experimental evaluation of our proposal. Section 6.1 describes the evaluation strategy. Sections 6.2 and 6.3 present the results for offline and online detection, respectively. Section 6.4 provides a general discussion of results.

6.1 EVALUATION STRATEGY

To evaluate the proposal, we defined a classification strategy, collected data to conduct the experiments, and trained/tested the machine learning models. This allowed us to evaluate the effectiveness of our anomaly detection solution, based on the categorization of WASI calls.

Intrusion detection systems can perform online and/or offline evaluation. In online evaluation, events of interest are collected and analyzed occurs in real-time, making it possible to identify attacks in progress. However, online evaluation restricts the amount of data to be analyzed and limits the choice of algorithms, in order to be fast. On the other hand, in offline evaluation an application is executed and monitored, and the events of interest are analyzed later. This is useful in a controlled environment, enabling the study of the entire application and its interactions, while allowing us to analyze more data with more sophisticated algorithms.

To assess the effectiveness of using WASI calls for anomaly detection, we defined two test scenarios. The first scenario considers an offline approach and uses all the interactions from the environment. The second scenario used a fixed size sliding window, defining a partial view of the environment, and simulating an online approach for intrusion detection solutions.

In both scenarios, the interactions from the environment were treated equally, using the same classification process for the WASI calls. This way, we are able to evaluate the effectiveness of using this type of interaction for anomaly detection in WebAssembly and discuss how this type of solution based on the use of categorical data may be used in other solutions.

Overall, we selected six Machine Learning (ML) models for the experiment, to study the impact of the proposal in different model approaches. These models are from a range of supervised learning algorithms, encompassing decision tree classifiers, ensemble methods, and neural network models. The models were used with their default parameters.

6.1.1 Dataset

It is necessary to select a set of applications that will be executed and observed, to collect information about the calls performed by each application during its execution. In the case of this work, different strategies can be explored for the extraction of the WASI calls in each binary sample, we used the *Wasmtime* runtime CLI (BytecodeAlliance, 2021b), which executes standalone (i.e. outside a browser). It provides functionality to generate detailed traces of the WebAssembly application being run, without instrumenting nor modifying the application. Listing 6.1 presents an example of part of a trace obtained from the execution of a Wasm application.

```
1 TRACE wasi_common::snapshots::preview_1::wasi_snapshot_preview1 > wiggle abi; module="
      wasi_snapshot_preview1" function="environ_sizes_get"
```

```
2 TRACE wasi_common::snapshots::preview_1::wasi_snapshot_preview1 > result=Ok((0, 0))
3 TRACE wasi_common::snapshots::preview_1::wasi_snapshot_preview1 > wiggle abi; module="
      wasi_snapshot_preview1" function="args_sizes_get"
4 TRACE wasi_common::snapshots::preview_1::wasi_snapshot_preview1 > result=Ok((1, 36))
5 TRACE wasi_common::snapshots::preview_1::wasi_snapshot_preview1 > wiggle abi; module="
      wasi_snapshot_preview1" function="args_get"
6 TRACE wasi_common::snapshots::preview_1::wasi_snapshot_preview1 > argv=*guest 0x104a70 argv_buf=*guest 0
      x104a80
7 TRACE wasi_common::snapshots::preview_1::wasi_snapshot_preview1 > result=Ok(())
```

Listing 6.1: Trace of a Wasm application.

Data gathering poses a problem, which is the availability of malicious content already classified. Since WebAssembly is a format that is still under development, the standardization of some models has not yet been carried out and there are still pending topics for discussion in the community. Consequently, few samples of malicious content are available, and the few samples available often focus on exploiting vulnerabilities in the compiler rather than in the format or in the runtime.

The dissemination of vulnerabilities and discussions about malicious content found in this area is limited, as presented in Section 4.2. Thus, a large part of the set of attacks described in the literature is not publicly available, consequently the available set is not considered sufficient for the development of a representative dataset (Section 4.2.6). In this way, it becomes necessary to define a set of vulnerabilities present in WebAssembly applications, aiming later at evaluating and identifying attacks using WebAssembly System Interface (WASI) calls.

Our goal is to have a varied set of samples that represents the environment. The variety of the data samples allows us to better assess how the data categorization proposed is able to highlight the functionality and operations behind each interaction of the application with the underlying environment. The samples are also helpful to demonstrate that the proposal categorization is generic and may be applied to other data sources.

The dataset used for the training and testing of our proposal was created by us from different sources, allowing us to build a dataset that represents a variety of behavior found in the WebAssembly environment. Overall, we have 26 types of attack samples present[1], with a total of 377 samples to form the anomaly class, and we have 263 samples for the normality class. The code samples came from a range of data sources like test suits, benchmarks, single WebAssembly modules, and applications obtained from (Stiévenart, 2023; Denis, 2023; WebAssembly, 2023a; Beyer, 2023).

The normality part of the dataset consists of WebAssembly applications that perform data processing and mathematical calculations that would be expected from an application. For the anomaly portion similar operations are presented, however, we explore vulnerabilities such as stack, heap, and buffer overflow. We do not focus on selecting any specific application niche of WebAssembly behavior.

6.1.2 Machine Learning Approach

Six algorithms were chosen to make the classification of the samples. The classifiers were chosen considering their popularity in security proposals for anomaly detection and also for variety, to avoid being limited to a single model strategy.

The algorithms chosen are *XGBoost*, *Decision Tree*, *Nu-Support Vector*, *Multilayer Perceptron (MLP)*, *AdaBoost*, and *Stochastic Gradient Descent (SGD)*. With a variety of supervised models, we can better understand the impact of categorical data representation

[1] A description of the attacks is presented in Appendix A.

and present results from classifiers commonly adopted in the literature for anomaly detection.

For the evaluation of the models, we use a 50/50 approach, where the dataset was split into 50% of the code samples for the training and 50% for testing. This approach was used for both online and offline evaluation.

6.1.3 Data Preprocessing

Listing 6.1 presents the raw data collected. Considering the example trace, the resulting WASI calls after the data cleaning are presented in Listing 6.2. The result is a trace that contains only the sequence of WASI calls performed by a WebAssembly application.

```
1 environ_sizes_get
2 args_sizes_get
3 args_get
```

Listing 6.2: WASI calls extracted.

Before training and testing the models using the dataset, the data need to be categorized. The preprocessing of the dataset allows the application of the categorical strategy defined in Section 5.3. This process is responsible for converting calls into their respective categories. A WASI call will be represented by an operational and functional category (as presented by Listing 6.3).

```
1 D1
2 D1
3 A1
```

Listing 6.3: Classification results.

After this conversion, the dataset is ready to be used for training and testing. All information collected during the execution of each WebAssembly application is used by the models. The information is presented in a vector of the frequency of the respective characteristics, which are now represented as function and operation categories.

6.2 OFFLINE DETECTION

In the offline evaluation, all the information collected during the execution of an application is used by the models during the classification phase. Table 6.1 presents the results obtained for each model, using the performance metrics presented in Section 2.4.2.

Classifier	**Precision**	**Recall**	**F1Score**	**Accuracy**	**BAC**	**Brier Score**
AdaBoost	98.38%	100.00%	99.18%	99.06%	98.91%	0.94%
Decision Tree	98.38%	100.00%	99.18%	99.06%	98.91%	0.94%
MLP	96.81%	100.00%	98.38%	98.12%	97.81%	1.88%
Nu-Support Vector	92.86%	100.00%	96.30%	95.61%	94.89%	4.39%
SGD	73.09%	89.56%	80.49%	75.24%	72.88%	24.76%
XGBoost	98.36%	100.00%	99.18%	99.06%	98.91%	0.94%

Table 6.1: Performance of the models for the offline strategy.

The **precision** shows the impact of false positives in the models. Although we obtained high precision values, our models are still impacted by a small fraction of the samples being misclassified and generating false positives. The **recall** does not depend on

precision and is influenced by false negative samples, which do not impact our models. The **f1score** is based on precision and recall. This metric enables the description of the positive classes, describing the adequacy of the models to classify the normal behavior and the detection of the intrusion classes. In these three metrics, we are not considering the impact of true negative samples.

The **accuracy** and **balanced accuracy** (BAC) metrics show a better understanding of the true positive and true negative classifications. Despite not considering the false negative/positive classes, we notice that our results are similar to the previous results found for the f1score, showing that our models are being able to classify correctly most of the samples. The similarity between accuracy and BAC results also shows that the dataset used for the training and test is balanced.

The lower the **Brier score** is, the more calibrated the models are to make the classification. This evaluation strategy enables the measurement between the predicted probability of a sample and the achieved result. Only two models present a brier score higher than 2%, and only one of them presents a poor overall result.

Stochastic Gradient Descent (SGD) is a linear classifier, which in our case is being impacted by a variety of samples found in the dataset. With a high number of classes that come from a variety of samples, the linear model is not being able to correctly distinguish between the two groups, resulting in a model that is not adequate for anomaly detection with this type of data.

The similarity between *Decision Tree* and *AdaBoost* results (over even *XGBoost*), is due to the proximity of the strategies used in such classifiers (Hastie et al., 2009; Molnar, 2020). They are also popular for intrusion detection and malware detection, enabling users to study the decision made by the classifier.

Overall, our results are promising for the proposal for anomaly detection, with all models achieving an *f1score* above 80%. Most of the evaluated models were able to classify correctly both classes, with few samples being missclassified. The strategy for offline detection is quite adequate using the classification of WASI calls issued by the WebAssembly application to its runtime.

6.3 ONLINE DETECTION

For the online proposal, we have as the objective to identify threats as soon as possible, during execution time. A new model was trained and tested using a strategy that allows an online point of view. A sliding window consists of a strategy where a fixed-size window is used to limit the view of the data set. This window slides through the data, offering a view of a set of features that are contained in the window. Considering for instance a sequence of data */abcdefghi/*, we simulated a limited vision for the models using a fixed-size sliding window, using two strategies: *non-overlapping* sliding windows (*abc*, *def*, *ghi*, ...) and *overlapping* sliding windows (*abc*, *bcd*, *cde*, ...).

With a sliding window, we can understand the impact of our proposal in a real-time intrusion detection strategy where only small portions of the execution interactions are available since the application is still running. In comparison with the previous solution, the same categorical classification is used, with the only difference being the use of the sliding window.

The sliding window size was defined as three (3), based on previous research using this value, the size of the traced applications, and characteristics observed from WebAssembly applications (Liu et al., 2018; Castanhel et al., 2020). WebAssembly

applications are limited to a module, a limited number of types exist in the format, and its design limits the number of operations available. In consequence, the size of an application is quite smaller than what would be found in other languages, and it generates a smaller amount of calls to the runtime. Such reasons led to choose a small window size.

Table 6.2 presents the results of the online experiments with a non-overlapping sliding window. Overall, the restriction on the information amount provided to the models impacts the detection efficacy. However, the precision was reduced in four of the classifiers and the recall reduced for all of them, directly affecting the f1score. The f1score and accuracy for these models indicate a reduction in the detection rate, leading to an increase of false positives and false negatives. This behavior was expected because we are trying to detect threats with only partial information being provided to the classifiers.

Classifier	Precision	Recall	F1Score	Accuracy	BAC	Brier Score
AdaBoost	88.20%	96.01%	91.94%	93.03%	93.46%	6.97%
Decision Tree	99.86%	92.88%	93.86%	94.97%	94.66%	5.03%
MLP	93.70%	92.88%	93.29%	94.46%	94.23%	5.54%
Nu-Support Vector	93.70%	92.88%	93.29%	94.46%	94.23%	5.54%
SGD	32.81%	68.58%	44.38%	28.83%	34.66%	71.17%
XGBoost	94.69%	92.88%	93.78%	94.90%	94.60%	5.10%

Table 6.2: Performance of the models for the real-time strategy (non-overlapping window).

Two of the classifiers in Table 6.2 achieved a better result for precision here than in the offline strategy, showing a reduction in the number of false positives. However, the same models had an increase in the number of false negatives (as their recall shows a reduction in comparison with Table 6.1), and the accuracy describes an impact in both classes (negative/positive detection rates are lower than in the previous solution).

Considering BAC and Brier score, it is clear that the models were impacted and had an increase in the number of misclassified samples. However, Table 6.2 shows a small reduction in comparison with the offline approach; we consider such results acceptable. The classifiers are still able to detect threats despite missclassifying some samples.

Considering an online (i.e. real-time) anomaly detection, the f1score metric above 93% obtained in four of the classifiers is considered an adequate result. The low score obtained by the SGD classifier was expected; as discussed for Table 6.1, such a linear model seems not well-suited for this kind of data.

Table 6.3 presents the results for the overlapping sliding window with size 3 and overstep 1 (*abc*, *cde*, *efg*, ...). We included these results as it is quite popular when using sliding windows to consider overlapping to simulate a real-world case and to increase the amount of data, since more windows are created. We noticed an increase in the number of false positives, despite the reduction in the number of false negatives. The models had an inferior performance, as demonstrated by the f1score result. The similar values of accuracy and BAC between Tables 6.2 and 6.3 are not enough to suggest that the use of overlapping windows is a good strategy when using categorical data.

Giving a deeper look at the data, we can better understand why the overlapping approach is not ideal for this type of data. Since our interactions that are WASI calls are being categorized, when the overlapping strategy is adopted we duplicated some of the information. In this case, we are adding more categories to the window generated from the trace of the application, because we have an overlap of one (that adds a category in each window). In all the categories proposed in Table C.1 (Appendix C), most groups present

Classifier	Precision	Recall	F1Score	Accuracy	BAC	Brier Score
AdaBoost	82.54%	100.00%	90.44%	92.30%	93.95%	7.70%
Decision Tree	82.44%	100.00%	90.37%	92.25%	93.91%	7.75%
MLP	81.47%	99.84%	89.73%	91.68%	93.43%	8.32%
Nu-Support Vector	72.17%	100.00%	83.83%	85.96%	88.97%	14.04%
SGD	71.97%	99.84%	83.65%	85.80%	88.80%	14.20%
XGBoost	82.44%	100.00%	90.37%	92.25%	93.91%	7.75%

Table 6.3: Performance of the models for the real-time strategy (overlapping window).

a similar number of calls, except for three classes. Two of them are from the high-threat level and present higher number of calls for classes C1 (87.4%) and B1 (23.1%). Class D5 also presented a growth of 37.2%. The increase in the false positive rate of our models is directly associated with the increase in the number of calls in the categories with high threat. As we are categorizing the information used, in the case of overlapping we are also adding information to the dataset that is incorrect. Since we are duplicating small portions of the data, we are also adding information related to operations, functionality, and security of the application that in the reality do not exist. For instance, an 87.4% increase in the number of calls in class C1 tells the models that... the threat is higher in comparison with the previous experiment (Table 6.2), which is not exact, because the threat is the same.

The impact in machine learning models also exists because of the categorization, we are specifying distinct features that with an overlapping strategy are misleading the models. For this reason, we cannot recommend the use of categorization and overlapping in the same data.

6.4 DISCUSSION OF RESULTS

The use of system calls for intrusion detection is not new (Forrest et al., 1996; Bernaschi et al., 2002). However, an approach that considers this type of interaction in the WebAssembly sandbox is a novelty. Representing calls as categorical data is also a new strategy for anomaly detection in this context. Our results highlight how the use of this type of strategy is able to properly represent the characteristics found in the applications on the dataset.

Using categorical data allows ML models to focus on essential aspects of applications' runtime behavior, abstracting away details that to not contribute to intrusion detection. This choice of representation has led to good results in both offline and online evaluation.

The experimental evaluation allows us to say that our proposal is a viable solution for intrusion detection for WebAssembly applications. While we have used Wasmtime in our experiments, this proposal can be adapted to other runtime environments. The categorical classification presented is flexible and easily applicable to other types of communication as system calls and Inter-Process Communication (IPC) mechanisms.

We applied a range of machine learning classifiers intending to evaluate the efficacy obtained by exclusively using categorical data for intrusion detection. As previously discussed, the models do not appear to be harmed by this design choice.

A downside to the use of categorical data is limited flexibility, since a finite and known set of classes is required for data classification. This requirement also limits the

expansion of the models to cover new features, since the addition of new categories has a direct influence on the entire previously classified set.

7 CONCLUSION

WebAssembly (Wasm) has been designed to improve the security, portability, and performance of Web applications in comparison to JavaScript (JS). The WebAssembly sandbox offers a secure environment to execute Wasm applications. Even if security is one of its chief concerns, proposals for intrusion detection in WebAssembly do not exist, to the best of our knowledge. Thus, the research goal of this thesis was to propose and evaluate a strategy for anomaly detection in WebAssembly applications.

To detect anomalies in WebAssembly applications, we explored their interactions with the environment through WASI calls, which are akin to the system calls provided by operating systems. We adopted a dynamic approach, observing the WASI calls issued by an application during execution. Leveraging prior work on system call categorization, we introduced two classifications for WASI calls, one according to the threat level and the other according to the functionality provided by each call. The WASI calls issued by an application are therefore substituted by their corresponding groups and fed into an ML model that classifies the application as either benign or malicious.

To evaluate the proposal, we compiled a dataset with 640 WebAssembly applications. This dataset has 263 benign and 377 malicious samples, representing 26 different types of attacks. Our evaluation used six different supervised learning models, and was performed in three scenarios: (i) offline; (ii) online with non-overlapping window; and (iii) online with overlapping window. The overall results highlight the potential of using WASI calls for anomaly detection, and show that categorical data are a viable representation when exploring Machine Learning (ML) models. One limitation of the proposal is that the use of categorical representations for data requires knowledge of all possible sets of categories.

The work presented in this thesis resulted in two publications that are directly associated with its research goals (Heinrich et al., 2023; Helpa et al., 2023). Future research considering the security of the WebAssembly environment could expand the type of data used by the detection strategies. The interactions from the environment are not limited to only WASI calls and the use of more parameters may be helpful for the improvement of the detection of anomalies. Also, the WebAssembly format has a range of tools for debugging. Accessing the static information found in a Wasm binary may prove helpful for the detection of attacks without relying on the WebAssembly sandbox.

REFERENCES

Agresti, A. (2003). *Categorical Data Analysis*. Wiley Series in Probability and Statistics. Wiley.

Akoglu, L., Tong, H., Vreeken, J., and Faloutsos, C. (2012). Fast and reliable anomaly detection in categorical data. In *Proceedings of the 21st ACM international conference on Information and knowledge management*, pages 415–424.

Alcorn, W., Frichot, C., and Orru, M. (2014). *The Browser Hacker's Handbook*. John Wiley & Sons.

Amazon (2023). Alexa Rank. `https://www.alexa.com/company`.

Anderson, J. P. (1972). Computer security technology planning study. volume 2. Technical report, Anderson (James P) and Co Fort Washington Pa.

Anderson, J. P. (1980). Computer security threat monitoring and surveillance. *Technical Report, James P. Anderson Company*.

André, P. M., Stiévenart, Q., and Ghafari, M. (2022). Developers struggle with authentication in Blazor WebAssembly. In *Proceedings of the 38th International Conference on Software Maintenance and Evolution*, pages 389–393, Limassol, Cyprus. IEEE.

Arteaga, J. C. (2022). Artificial software diversification for WebAssembly. Stockholm, Sweden.

Arteaga, J. C., Laperdrix, P., Monperrus, M., and Baudry, B. (2022). Multi-variant execution at the edge. In *Proceedings of the 9th Workshop on Moving Target Defense*, pages 11–22, Los Angeles, CA, USA. ACM.

Arteaga, J. C., Malivitsis, O., Perez, O. V., Baudry, B., and Monperrus, M. (2021). CROW: Code diversification for WebAssembly. In *Proceedings of the Workshop on Measurements, Attacks, and Defenses for the Web*, Virtual Event. Internet Society.

Balakrishnan, A. and Schulze, C. (2005). Code obfuscation literature survey. *CS701 Construction of Compilers*, 19.

Balderas, M.-A., Berzal, F., Cubero, J.-C., Eisman, E., and Marn, N. (2005). Discovering hidden association rules. In *Proc. International Workshop on Data Mining Methods for Anomaly Detection (KDD 05)*.

Bandhakavi, S., King, S. T., Madhusudan, P., and Winslett, M. (2010). {VEX}: Vetting browser extensions for security vulnerabilities. In *19th USENIX Security Symposium (USENIX Security 10)*.

Barth, A., Jackson, C., Reis, C., Team, T., et al. (2008). The security architecture of the chromium browser. In *Technical report*. Stanford University.

Bastys, I., Algehed, M., Sjösten, A., and Sabelfeld, A. (2022). SecWasm: Information flow control for WebAssembly. In *Proceedings of the 29th International Static Analysis Symposium*, pages 74–103, Auckland, New Zealand. Springer.

Battagline, R. (2021). *The Art of WebAssembly: Build Secure, Portable, High-Performance Applications*. No Starch Press.

Baumgärtner, L., Höchst, J., and Meuser, T. (2019). B-DTN7: Browser-based disruption-tolerant networking via bundle protocol 7. In *Proceedings of the 6th International Conference on Information and Communication Technologies for Disaster Management*, pages 1–8, Paris, France. IEEE.

Bay, S. D. and Pazzani, M. J. (1999). Detecting change in categorical data: Mining contrast sets. In *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 302–306.

Bernaschi, M., Gabrielli, E., and Mancini, L. V. (2002). Remus: A security-enhanced operating system. *ACM Transactions on Information and System Security (TISSEC)*, 5(1):36–61.

Beyer, C. (2023). Amalgamated webassembly system interface test suite. `https://github.com/caspervonb/wasi-test-suite`.

Bhansali, S., Aris, A., Acar, A., Oz, H., and Uluagac, A. S. (2022). A first look at code obfuscation for WebAssembly. In *Proceedings of the 15th Conference on Security and Privacy in Wireless and Mobile Networks*, pages 140–145, San Antonio, TX, USA. ACM.

Bian, W., Meng, W., and Wang, Y. (2019). Poster: Detecting WebAssembly-based cryptocurrency mining. In *Proceedings of the 26th Conference on Computer and Communications Security*, pages 2685–2687, London, UK. ACM.

Bian, W., Meng, W., and Zhang, M. (2020). Minethrottle: Defending against wasm in-browser cryptojacking. In *Proceedings of The Web Conference 2020*, pages 3112–3118.

Bosamiya, J., Lim, W. S., and Parno, B. (2022). Provably-Safe multilingual software sandboxing using WebAssembly. In *Proceedings of the 31st USENIX Security Symposium*, pages 1975–1992, Boston, MA, USA. USENIX Association.

Botacin, M., Aghakhani, H., Ortolani, S., Kruegel, C., Vigna, G., Oliveira, D., Geus, P. L. D., and Grégio, A. (2021). One size does not fit all: A longitudinal analysis of brazilian financial malware. *ACM Transactions on Privacy and Security (TOPS)*, 24(2):1–31.

Bridges, R. A., Glass-Vanderlan, T. R., Iannacone, M. D., Vincent, M. S., and Chen, Q. G. (2019). A survey of intrusion detection systems leveraging host data. *ACM Comput. Surv.*, 52(6).

Brito, T., Lopes, P., Santos, N., and Santos, J. F. (2022). Wasmati: An efficient static vulnerability scanner for WebAssembly. *Computers & Security*, 118:102745.

Brown, D. J., Suckow, B., and Wang, T. (2002). A survey of intrusion detection systems. *Department of Computer Science, University of California, San Diego*.

Bruyat, J., Champin, P.-A., Médini, L., and Laforest, F. (2021). WasmTree: Web Assembly for the semantic Web. In *Proceedings of the European Semantic Web Conference*, pages 582–597, Virtual Event. Springer.

BytecodeAlliance (2021a). Wasmtime. https://github.com/bytecodealliance/wasmtime/blob/main/docs/WASI-overview.md.

BytecodeAlliance (2021b). Wasmtime. https://docs.wasmtime.dev/introduction.html.

BytecodeAlliance (2023). Lucet has reached End-of-life. https://hacl-star.github.io/.

Canfora, G., Medvet, E., Mercaldo, F., and Visaggio, C. A. (2014). Detection of malicious web pages using system calls sequences. In *International Conference on Availability, Reliability, and Security*, pages 226–238. Springer.

Castanhel, G. R., Heinrich, T., Ceschin, F., and Maziero, C. (2021). Taking a peek: An evaluation of anomaly detection using system calls for containers. In *2021 IEEE Symposium on Computers and Communications (ISCC)*, pages 1–6. IEEE.

Castanhel, G. R., Heinrich, T., Ceschin, F., and Maziero, C. A. (2020). Sliding window: The impact of trace size in anomaly detection system for containers through machine learning. In *Anais da XVIII Escola Regional de Redes de Computadores*, pages 141–146. SBC.

Ceschin, F., Gomes, H. M., Botacin, M., Bifet, A., Pfahringer, B., Oliveira, L. S., and Grégio, A. (2020a). Machine learning (in) security: A stream of problems. *arXiv preprint arXiv:2010.16045*.

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. *CoRR*, abs/2010.16045.

Charu, C. A. (2017). *Outlier Analysis*. Springer Cham.

Chen, W., Sun, Z., Wang, H., Luo, X., Cai, H., and Wu, L. (2022). WASAI: Uncovering vulnerabilities in Wasm smart contracts. In *Proceedings of the 31st International Symposium on Software Testing and Analysis*, pages 703–715, Virtual Event. ACM.

Creech, G. and Hu, J. (2013). A semantic approach to host-based intrusion detection systems using contiguousand discontiguous system call patterns. *IEEE Transactions on Computers*, 63(4):807–819.

Das, K. and Schneider, J. (2007). Detecting anomalous records in categorical datasets. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 220–229.

Das, K., Schneider, J., and Neill, D. B. (2008). Anomaly pattern detection in categorical datasets. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 169–176.

Debar, H., Dacier, M., and Wespi, A. (1999). Towards a taxonomy of intrusion-detection systems. *Computer Networks*, 31(8):805–822.

Denis, F. (2023). webassembly-benchmarks. https://github.com/jedisct1/webassembly-benchmarks.

Disselkoen, C., Renner, J., Watt, C., Garfinkel, T., Levy, A., and Stefan, D. (2019). Position paper: Progressive memory safety for WebAssembly. In *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*, pages 1–8, Phoenix, AZ, USA. ACM.

Durumeric, Z. (2023). Cached chrome top million websites. `https://github.com/zakird/crux-top-lists`.

Emscripten (2021). Emscripten is a complete compiler toolchain to WebAssembly, using LLVM, with a special focus on speed, size, and the Web platform. `https://emscripten.org/`.

EOSIO (2023). EOSIO is a next-generation, open-source blockchain protocol with industry-leading transaction speed and flexible utility. `https://github.com/EOSIO`.

Erlinger, M. A. and Wood, M. (2007). Intrusion Detection Message Exchange Requirements. RFC 4766.

Feldman, R. (2019). Why isn't functional programming the norm? `https://tinyurl.com/yyhtxt74`.

Felker, R. (2023). Musl libc. `https://musl.libc.org/`.

Forrest, S., Hofmeyr, S. A., Somayaji, A., and Longstaff, T. A. (1996). A sense of self for unix processes. In *Proceedings 1996 IEEE Symposium on Security and Privacy*, pages 120–128. IEEE.

Fraiwan, M., Al-Salman, R., Khasawneh, N., and Conrad, S. (2012). Analysis and identification of malicious javascript code. *Information Security Journal: A Global Perspective*, 21(1):1–11.

fstar-lang (2023). Introduction to F*. `https://fstar-lang.org/`.

Fu, W., Lin, R., and Inge, D. (2018). TaintAssembly: Taint-based information flow control tracking for WebAssembly.

Gadepalli, P. K., McBride, S., Peach, G., Cherkasova, L., and Parmer, G. (2020). Sledge: A serverless-first, light-weight Wasm runtime for the edge. In *Proceedings of the 21st International Middleware Conference*, pages 265–279, Delft, Netherlands. ACM.

Galante, L., Botacin, M., Grégio, A., and de Geus, P. (2019). Forseti: Extração de características e classificação de binários elf. In *Anais Estendidos do XIX Simpósio Brasileiro de Segurança da Informação e de Sistemas Computacionais*, pages 5–10. SBC.

Galvin, P. B., Gagne, G., Silberschatz, A., et al. (2003). *Operating system concepts*, volume 10. John Wiley & Sons.

Gao, S., Fu, H., Zhang, H., Zhang, J., and Li, G. (2022). ZAWA: A ZKSNARK WASM emulator. *Proceedings of the ACM on Programming Languages*, 1(1).

Genkin, D., Pachmanov, L., Tromer, E., and Yarom, Y. (2018). Drive-by key-extraction cache attacks from portable code. In *Proceedings of the 16th International Conference on Applied Cryptography and Network Security*, pages 83–102, Leuven, Belgium. Springer.

Golsch, L. (2019). Webassembly: Basics. *Technical University of Braunschweig*.

Goltzsche, D., Nieke, M., Knauth, T., and Kapitza, R. (2019). AccTEE: A WebAssembly-based two-way sandbox for trusted resource accounting. In *Proceedings of the 20th International Middleware Conference*, pages 123–135, Davis, CA, USA. ACM.

Goltzsche, D., Siebels, T., Golsch, L., and Kapitza, R. (2020). Hector: Using untrusted browsers to provision Web applications.

Grier, C., Tang, S., and King, S. T. (2008). Secure web browsing with the op web browser. In *2008 IEEE Symposium on Security and Privacy (SP 2008)*, pages 402–416.

Grosskurth, A. and Godfrey, M. W. (2006). Architecture and evolution of the modern web browser. *Preprint submitted to Elsevier Science*, 12(26):235–246.

Guha, A., Fredrikson, M., Livshits, B., and Swamy, N. (2011). Verified security for browser extensions. In *2011 IEEE symposium on security and privacy*, pages 115–130. IEEE.

Haas, A., Rossberg, A., Schuff, D. L., Titzer, B. L., Holman, M., Gohman, D., Wagner, L., Zakai, A., and Bastien, J. (2017). Bringing the Web up to speed with WebAssembly. In *Proceedings of the 38th Conference on Programming Language Design and Implementation*, pages 185–200, Barcelona, Spain. ACM.

HACL* (2023). A High Assurance Cryptographic Library. `https://hacl-star.github.io/`.

Hancock, J. T. and Khoshgoftaar, T. M. (2020). Survey on categorical data for neural networks. *Journal of Big Data*, 7(1):1–41.

Handa, A., Sharma, A., and Shukla, S. K. (2019). Machine learning in cybersecurity: A review. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 9(4):e1306.

Haßler, K. and Maier, D. (2021). WAFL: Binary-only WebAssembly fuzzing with fast snapshots. In *Proceedings of the 5th Reversing and Offensive-Oriented Trends Symposium*, pages 23–30, Vienna, Austria. ACM.

Hastie, T., Tibshirani, R., Friedman, J. H., and Friedman, J. H. (2009). *The elements of statistical learning: data mining, inference, and prediction*, volume 2. Springer.

He, N., Zhang, R., Wang, H., Wu, L., Luo, X., Guo, Y., Yu, T., and Jiang, X. (2021). EOSAFE: Security analysis of EOSIO smart contracts. In *Proceedings of the 30th USENIX Security Symposium*, pages 1271–1288, Vancouver, BC, Canada. USENIX Association.

He, Z., Deng, S., and Xu, X. (2005). An optimization model for outlier detection in categorical data. In *International Conference on Intelligent Computing*, pages 400–409. Springer.

Heinrich, T., Will, N. C., Obelheiro, R. R., and Maziero, C. A. (2023). Uso de chamadas wasi para a identificação de ameaças em aplicações webassembly. *Anais Estendidos do XXIII Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais. SBC*.

Helpa, C., Heinrich, T., Botacin, M., Will, N. C., Obelheiro, R. R., and Maziero, C. A. (2023). Uma estratégia dinâmica para a detecção de anomalias em binários webassembly. *Anais Estendidos do XXIII Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais. SBC*.

Hilbig, A., Lehmann, D., and Pradel, M. (2021). An empirical study of real-world webassembly binaries: Security, languages, use cases. In *Proceedings of the Web Conference 2021*, pages 2696–2708.

Hoffman, K. (2019). Programming webassembly with rust: unified development for web, mobile, and embedded applications. *Programming WebAssembly with Rust*, pages 1–220.

Internet World Stats (2020). World Internet Usage and Population Statistics.

Jain, K. and Sekar, R. (2000). User-level infrastructure for system call interposition: A platform for intrusion detection and confinement. In *NDSS*.

Karami, S., Ilia, P., Solomos, K., and Polakis, J. (2020). Carnus: Exploring the privacy threats of browser extension fingerprinting. In *Proceedings of the 27th Network and Distributed System Security Symposium (NDSS)*.

Kim, M., Jang, H., and Shin, Y. (2022). Avengers, Assemble! survey of WebAssembly security solutions. In *Proceedings of the 15th International Conference on Cloud Computing*, pages 543–553, Barcelona, Spain. IEEE.

Kitchenham, B. and Charters, S. (2007). Guidelines for performing systematic literature reviews in software engineering. Technical Report EBSE-2007-01, School of Computer Science and Mathematics, Keele University.

Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., et al. (2020). Spectre attacks: Exploiting speculative execution. *Communications of the ACM*, 63(7):93–101.

Kolosick, M., Narayan, S., Johnson, E., Watt, C., LeMay, M., Garg, D., Jhala, R., and Stefan, D. (2022). Isolation without taxation: Near-zero-cost transitions for WebAssembly and SFI. *Proceedings of the ACM on Programming Languages*, 6(POPL).

Koren, I. (2021). A standalone WebAssembly development environment for the internet of things. In *Proceedings of the 21st International Conference on Web Engineering*, pages 353–360, Biarritz, France. Springer.

Koufakou, A. (2009). *Scalable and efficient outlier detection in large distributed data sets with mixed-type attributes*. University of Central Florida.

Koufakou, A., Ortiz, E. G., Georgiopoulos, M., Anagnostopoulos, G. C., and Reynolds, K. M. (2007). A scalable and efficient outlier detection strategy for categorical data. In *19th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2007)*, volume 2, pages 210–217. IEEE.

Kumar, S. (2007). Survey of current network intrusion detection techniques. *Washington Univ. in St. Louis*, pages 1–18.

Kunft, A., Katsifodimos, A., Schelter, S., Breß, S., Rabl, T., and Markl, V. (2019). An intermediate representation for optimizing machine learning pipelines. *Proceedings of the VLDB Endowment*, 12(11):1553–1567.

Lam, A. (2005). New ips to boost security, reliability and performance of the campus network. *Newsletter of Computing Services Center*.

Le Pochat, V., Van Goethem, T., Tajalizadehkhoob, S., Korczyński, M., and Joosen, W. (2018). Tranco: A research-oriented top sites ranking hardened against manipulation. *arXiv preprint arXiv:1806.01156*.

Lee, W., Stolfo, S. J., and Chan, P. K. (1997). Learning patterns from Unix process execution traces for intrusion detection. In *AAAI Workshop on AI Approaches to Fraud Detection and Risk Management*, pages 50–56. New York;.

Lehmann, D., Kinder, J., and Pradel, M. (2020). Everything old is new again: Binary security of webassembly. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 217–234.

Lehmann, D. and Pradel, M. (2019). Wasabi: A framework for dynamically analyzing webassembly. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1045–1058.

Lehmann, D. and Pradel, M. (2022). Finding the dwarf: Recovering precise types from WebAssembly binaries. In *Proceedings of the 43rd International Conference on Programming Language Design and Implementation*, pages 410–425, San Diego, CA, USA. ACM.

Lehmann, D., Torp, M. T., and Pradel, M. (2021). Fuzzm: Finding memory bugs through binary-only instrumentation and fuzzing of WebAssembly.

Lemos, R., Heinrich, T., Will, N. C., Obelheiro, R. R., and Maziero, C. A. (2023). Inspecting binder transactions to detect anomalies in android. In *Proceedings of the 17th Annual IEEE International Systems Conference*, Vancouver, BC, Canada. IEEE.

Li, B., Fan, H., Gao, Y., and Dong, W. (2021). ThingSpire OS: A WebAssembly-based IoT operating system for cloud-edge integration. In *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*, pages 487–488, Virtual Event. ACM.

Li, J., Lan, X., Li, X., Wang, J., Zheng, N., and Wu, Y. (2016). Online variable coding length product quantization for fast nearest neighbor search in mobile retrieval. *IEEE Transactions on Multimedia*, 19(3):559–570.

Li, J., Zhang, J., Pang, N., and Qin, X. (2018). Weighted outlier detection of high-dimensional categorical data using feature grouping. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 50(11):4295–4308.

Li, L. T. and Zhang, M. (2022). Poster: EOSDFA: Data flow analysis of EOSIO smart contracts. In *Proceedings of the Conference on Computer and Communications Security*, pages 3391–3393, Los Angeles, CA, USA. ACM.

Liang, H., Pei, X., Jia, X., Shen, W., and Zhang, J. (2018). Fuzzing: State of the art. *IEEE Transactions on Reliability*, 67(3):1199–1218.

Liao, H.-J., Lin, C.-H. R., Lin, Y.-C., and Tung, K.-Y. (2013). Intrusion detection system: A comprehensive review. *Journal of Network and Computer Applications*, 36(1):16–24.

Liao, Y. and Vemuri, V. R. (2002). Using text categorization techniques for intrusion detection. In *USENIX Security Symposium*, volume 12, pages 51–59.

Linux Programmer's Manual (2021a). intro - introduction to system calls. `https://man7.org/linux/man-pages/man2/intro.2.html`.

Linux Programmer's Manual (2021b). syscalls - linux system calls. `https://man7.org/linux/man-pages/man2/syscalls.2.html`.

Liu, M., Xue, Z., He, X., and Chen, J. (2021a). Scads: A scalable approach using spark in cloud for host-based intrusion detection system with system calls. *arXiv preprint arXiv:2109.11821*.

Liu, M., Xue, Z., Xu, X., Zhong, C., and Chen, J. (2018). Host-based intrusion detection system with system calls: Review and future trends. *ACM Computing Surveys (CSUR)*, 51(5):98.

Liu, R., Garcia, L., and Srivastava, M. (2021b). Aerogel: Lightweight access control framework for WebAssembly-based bare-metal IoT devices. In *Proceedings of the 6th Symposium on Edge Computing*, pages 94–105, San Jose, CA, USA. IEEE.

LLVM (2023). The llvm compiler infrastructure. `https://github.com/llvm/llvm-project`.

LLVM-Team (2023). The llvm project. `https://llvm.org/`.

Marques, F., Fragoso Santos, J., Santos, N., and Adão, P. (2022). Concolic execution for WebAssembly. In *Proceedings of the 36th European Conference on Object-Oriented Programming*, Berlin, Germany. DROPS.

Mazaheri, M. E., Bayat Sarmadi, S., and Taheri Ardakani, F. (2022). A study of timing side-channel attacks and countermeasures on JavaScript and WebAssembly. *The ISC International Journal of Information Security*, 14(1):27–46.

McFadden, B., Lukasiewicz, T., Dileo, J., and Engler, J. (2018). Security chasms of wasm. *NCC Group Whitepaper*.

Ménétrey, J., Pasin, M., Felber, P., and Schiavoni, V. (2021). Twine: An embedded trusted runtime for webassembly. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 205–216. IEEE.

Ménétrey, J., Pasin, M., Felber, P., and Schiavoni, V. (2022). WebAssembly as a common layer for the cloud-edge continuum. In *Proceedings of the 2nd Workshop on Flexible Resource and Application Management on the Edge*, pages 3–8, Minneapolis, MN, USA. ACM.

Michael, A. E., Gollamudi, A., Bosamiya, J., Disselkoen, C., Denlinger, A., Watt, C., Parno, B., Patrignani, M., Vassena, M., and Stefan, D. (2023). MSWasm: Soundly enforcing memory-safe execution of unsafe code. *Proceedings of the ACM on Programming Languages*, 1(1).

Ming, J., Wu, D., Wang, J., Xiao, G., and Liu, P. (2016). StraightTaint: Decoupled offline symbolic taint analysis. In *Proceedings of the 31st International Conference on Automated Software Engineering*, pages 308–319, Singapore. IEEE.

Molnar, C. (2020). *Interpretable Machine Learning: A Guide for Making Black Box Models Interpretable*. Leanpub.

Mozilla (2022a). How CSS works. `https://developer.mozilla.org/en-US/docs/Learn/CSS/First_steps/How_CSS_works`.

Mozilla (2022b). Introduction to the DOM. `https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction`.

Munsters, A., Pupo, A. L. S., Bauwens, J., and Boix, E. G. (2021). Oron: Towards a dynamic analysis instrumentation platform for AssemblyScript. In *Proceedings of the 5th International Conference on the Art, Science, and Engineering of Programming*, pages 6–13, Cambridge, UK. ACM.

Musch, M., Wressnegger, C., Johns, M., and Rieck, K. (2019). New kid on the web: A study on the prevalence of webassembly in the wild. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 23–42. Springer.

Ménétrey, J., Pasin, M., Felber, P., and Schiavoni, V. (2022). WaTZ: A trusted WebAssembly runtime environment with remote attestation for TrustZone. In *Proceedings of the 42nd International Conference on Distributed Computing Systems*, pages 1177–1189, Bologna, Italy. IEEE.

Narayan, S., Disselkoen, C., Garfinkel, T., Froyd, N., Rahm, E., Lerner, S., Shacham, H., and Stefan, D. (2020). Retrofitting fine grain isolation in the Firefox renderer. In *Proceedings of the 29th USENIX Security Symposium*, pages 699–716, Boston, MA, USA. USENIX Association.

Narayan, S., Disselkoen, C., Moghimi, D., Cauligi, S., Johnson, E., Gang, Z., Vahldiek-Oberwagner, A., Sahita, R., Shacham, H., Tullsen, D., et al. (2021). Swivel: Hardening WebAssembly against Spectre. In *Proceedings of the 30th USENIX Security Symposium*, pages 1433–1450, Vancouver, BC, Canada. USENIX Association.

Narayan, S., Garfinkel, T., Lerner, S., Shacham, H., and Stefan, D. (2019). Gobi: Webassembly as a practical path to library sandboxing. *arXiv preprint arXiv:1912.02285*.

Nieke, M., Almstedt, L., and Kapitza, R. (2021). Edgedancer: Secure mobile webassembly services on the edge. In *Proceedings of the 4th International Workshop on Edge Systems, Analytics and Networking*, pages 13–18.

Oates, T. and Cohen, P. R. (1996). Searching for structure in multiple streams of data. In *ICML*, volume 96, pages 346–354.

Ortega, J. L. (2014). *Academic Search Engines: A Quantitative Outlook*. Chandos Publishing (Oxford), Witney, England.

Patil, D. R. and Patil, J. (2017). Detection of malicious javascript code in web pages. *Indian Journal of Science and Technology*, 10(19):1–12.

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.

Pendleton, M. and Xu, S. (2017). A dataset generator for next generation system call host intrusion detection systems. In *MILCOM 2017-2017 IEEE Military Communications Conference (MILCOM)*, pages 231–236. IEEE.

Petrov, I., Invernizzi, L., and Bursztein, E. (2020). CoinPolice: Detecting hidden cryptojacking attacks with neural networks.

Pop, V. A. B., Niemi, A., Manea, V., Rusanen, A., and Ekberg, J.-E. (2022). Towards securely migrating WebAssembly enclaves. In *Proceedings of the 15th European Workshop on Systems Security*, pages 43–49, Rennes, France. ACM.

Powers, D. and Xie, Y. (2008). *Statistical methods for categorical data analysis*. Emerald Group Publishing.

Powers, D. and Xie, Y. (2010). Statistical methods for categorical data analysis. *International Statistical Review*, 78:317–317.

Prevelakis, V. and Spinellis, D. (2001). Sandboxing applications. In *Proceedings of the USENIX Annual Technical Conference*, pages 119–126, Boston, MA, USA. USENIX Association.

Protzenko, J., Beurdouche, B., Merigoux, D., and Bhargavan, K. (2019). Formally verified cryptographic Web applications in WebAssembly. In *Proceedings of the 40th Symposium on Security and Privacy*, pages 1256–1274, San Francisco, CA, USA. IEEE.

Qiang, W., Dong, Z., and Jin, H. (2018). Se-Lambda: Securing privacy-sensitive serverless applications using SGX enclave. In *Proceedings of the 14th International Conference on Security and Privacy in Communication Systems*, pages 451–470, Singapore. Springer.

Quan, L., Wu, L., and Wang, H. (2019). EVulHunter: Detecting fake transfer vulnerabilities for EOSIO's smart contracts at Webassembly-level.

Radovici, A., Rusu, C., and Serban, R. (2018). A survey of IoT security threats and solutions. In *Proceedings of the 17th RoEduNet Conference: Networking in Education and Research*, pages 1–5, Cluj-Napoca, Romania. IEEE.

Rajagopalan, M., Hiltunen, M. A., Jim, T., and Schlichting, R. D. (2006). System call monitoring using authenticated system calls. *IEEE Transactions on Dependable and Secure Computing*.

Reis, C., Barth, A., and Pizano, C. (2009). Browser security: Lessons from google chrome: Google chrome developers focused on three key problems to shield the browser from attacks. *Queue*, 7(5):3–8.

Reis, C., Moshchuk, A., and Oskov, N. (2019). Site isolation: Process separation for web sites within the browser. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1661–1678.

Renner, J., Cauligi, S., and Stefan, D. (2018). Constant-time WebAssembly. In *Proceedings of the 45th Symposium on Principles of Programming Languages*, Los Angeles, CA, USA. ACM.

Romano, A., Lehmann, D., Pradel, M., and Wang, W. (2022). Wobfuscator: Obfuscating JavaScript malware via opportunistic translation to WebAssembly. In *Proceedings of the 43rd Symposium on Security and Privacy*, pages 1574–1589, San Francisco, CA, USA. IEEE.

Romano, A., Liu, X., Kwon, Y., and Wang, W. (2021). An empirical study of bugs in WebAssembly compilers. In *Proceedings of the 36th International Conference on Automated Software Engineering*, pages 42–54, Melbourne, Australia. IEEE.

Romano, A. and Wang, W. (2020a). Wasim: Understanding WebAssembly applications through classification. In *Proceedings of the 35th International Conference on Automated Software Engineering*, pages 1321–1325, Melbourne, Australia. IEEE.

Romano, A. and Wang, W. (2020b). WasmView: Visual testing for WebAssembly applications. In *Proceedings of the 42nd International Conference on Software Engineering*, pages 13–16, Seoul, South Korea. ACM.

Rossberg, A. (2022). Webassembly specification. `https://webassembly.github.io/spec/core/index.html`.

Rourke, M. (2018). *Learn WebAssembly: Build web applications with native performance using Wasm and C/C++*. Packt Publishing Ltd.

Ruth, K., Kumar, D., Wang, B., Valenta, L., and Durumeric, Z. (2022). Toppling top lists: Evaluating the accuracy of popular website lists. In *Proceedings of the 22nd ACM Internet Measurement Conference*, pages 374–387.

Šilić, M., Krolo, J., and Delač, G. (2010). Security vulnerabilities in modern web browser architecture. In *The 33rd International Convention MIPRO*, pages 1240–1245. IEEE.

Spreitzer, R., Moonsamy, V., Korak, T., and Mangard, S. (2017). Systematic classification of side-channel attacks: A case study for mobile devices. *IEEE Communications Surveys & Tutorials*, 20(1):465–488.

stackoverflow (2023). Stackoverflow. `https://stackoverflow.com/`.

Stallings, W., Brown, L., Bauer, M. D., and Bhattacharjee, A. K. (2012). *Computer security: principles and practice*. Pearson Education Upper Saddle River, NJ, USA.

StatCounter (2020). Desktop vs Mobile vs Tablet Market Share Worldwide.

Stephen, A. (2022). Awesome WebAssembly languages. `https://github.com/appcypher/awesome-wasm-langs`.

Stiévenart, Q., Binkley, D. W., and De Roover, C. (2022a). Static stack-preserving intra-procedural slicing of WebAssembly binaries. In *Proceedings of the 44th International Conference on Software Engineering*, pages 2031–2042, Pittsburgh, PA, USA. IEEE.

Stiévenart, Q. and De Roover, C. (2020). Compositional information flow analysis for WebAssembly programs. In *Proceedings of the 20th International Working Conference on Source Code Analysis and Manipulation*, pages 13–24, Adelaide, Australia. IEEE.

Stiévenart, Q., De Roover, C., and Ghafari, M. (2021). The security risk of lacking compiler protection in WebAssembly. In *Proceedings of the 21st International Conference on Software Quality, Reliability and Security*, pages 132–139, Hainan, China. IEEE.

Stiévenart, Q., De Roover, C., and Ghafari, M. (2022b). Security risks of porting C programs to WebAssembly. In *Proceedings of the 37th Symposium on Applied Computing*, pages 1713–1722, Virtual Event. ACM.

Stiévenart, Q. (2023). Sac 2022 dataset. `https://figshare.com/articles/dataset/SAC_2022_Dataset/17297477`.

Stock, B., Johns, M., Steffens, M., and Backes, M. (2017). How the Web Tangled Itself: Uncovering the History of Client-Side Web In Security. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 971–987.

Sudusinghe, C., Charles, S., and Mishra, P. (2021). Denial-of-service attack detection using machine learning in network-on-chip architectures. In *Proceedings of the 15th IEEE/ACM International Symposium on Networks-on-Chip*, pages 35–40.

Sun, J., Cao, D., Liu, X., Zhao, Z., Wang, W., Gong, X., and Zhang, J. (2019). SELWasm: A code protection mechanism for WebAssembly. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing with Applications*, pages 1099–1106, Xiamen, China. IEEE.

Sun, S., Ma, H., Song, Z., and Zhang, R. (2022). WebCloud: Web-based cloud storage for secure data sharing across platforms. *IEEE Transactions on Dependable and Secure Computing*, 19(3):1871–1884.

Szanto, A., Tamm, T., and Pagnoni, A. (2018). Taint tracking for WebAssembly.

Taha, A. and Hadi, A. S. (2016). Pair-wise association measures for categorical and mixed data. *Information Sciences*, 346:73–89.

Taivalsaari, A., Mikkonen, T., Ingalls, D., and Palacz, K. (2008). Web browser as an application platform. In *2008 34th Euromicro Conference Software Engineering and Advanced Applications*, pages 293–302. IEEE.

Ter Louw, M., Lim, J. S., and Venkatakrishnan, V. N. (2008). Enhancing web browser security against malware extensions. *Journal in Computer Virology*, 4(3):179–195.

Titzer, B. L. (2022). A fast in-place interpreter for WebAssembly. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA2).

Tiwari, T., Starobinski, D., and Trachtenberg, A. (2018). Distributed Web mining of Ethereum. In *Proceedings of the International Symposium on Cyber Security Cryptography and Machine Learning*, pages 38–54, Beer-Sheva, Israel. Springer.

Tsoupidi, R. M., Balliu, M., and Baudry, B. (2021). Vivienne: Relational verification of cryptographic implementations in WebAssembly. In *Proceedings of the Secure Development Conference*, pages 94–102, Atlanta, GA, USA. IEEE.

Văduva, J.-A., Chiscariu, R.-E., Culic, I., Florea, I.-M., and Rughinis, R. (2019). Adrem: System call based intrusion detection framework. *eLearning & Software for Education*, 1.

Vassena, M., Disselkoen, C., Gleissenthall, K. v., Cauligi, S., Kıcı, R. G., Jhala, R., Tullsen, D., and Stefan, D. (2021). Automatically eliminating speculative leaks from cryptographic code with Blade. *Proceedings of the ACM on Programming Languages*, 5(POPL).

Vassena, M. and Patrignani, M. (2020). Memory safety preservation for WebAssembly. In *Proceedings of the 47th Symposium on Principles of Programming Languages*, New Orleans, LA, USA. ACM.

W3C Community Group (2022a). WA: Security. `https://webassembly.org/docs/security/`.

W3C Community Group (2022b). WA WebAssembly. `https://webassembly.org`.

Wadlow, T. and Gorelik, V. (2022). Security in the browser. `https://cacm.acm.org/magazines/2009/5/24645-security-in-the-browser/fulltext`.

Wang, S., Ye, G., Li, M., Yuan, L., Tang, Z., Wang, H., Wang, W., Wang, F., Ren, J., Fang, D., and Wang, Z. (2019). Leveraging WebAssembly for numerical JavaScript code virtualization. *IEEE Access*, 7:182711–182724.

Wang, W., Ferrell, B., Xu, X., Hamlen, K. W., and Hao, S. (2018). SEISMIC: SEcure In-lined Script Monitors for Interrupting Cryptojacks. In *Proceedings of the 23rd European Symposium on Research in Computer Security*, pages 122–142, Barcelona, Spain. Springer.

Watt, C., Renner, J., Popescu, N., Cauligi, S., and Stefan, D. (2019a). CT-Wasm: Type-driven secure cryptography for the Web ecosystem. *Proceedings of the ACM on Programming Languages*, 3(POPL).

Watt, C., Rossberg, A., and Pichon-Pharabod, J. (2019b). Weakening WebAssembly. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–28.

Wazlawick, R. S. (2009). *Metodologia de pesquisa para ciência da computação*, volume 2. Elsevier, Rio de Janeiro, RJ, Brazil.

WebAssembly (2023a). Wasi tests. `https://github.com/WebAssembly/wasi-testsuite`.

WebAssembly (2023b). Webassembly proposals. `https://github.com/WebAssembly/proposals`.

Wen, E. and Weber, G. (2020). Wasmachine: Bring the edge up to speed with a WebAssembly OS. In *Proceedings of the 13th International Conference on Cloud Computing*, pages 353–360, Beijing, China. IEEE.

Wen, E., Weber, G., and Nanayakkara, S. (2021). WasmAndroid: A cross-platform runtime for native programming languages on Android (WIP paper). In *Proceedings of the 22nd International Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 80–84, Virtual Event. ACM.

Will, N. C., Heinrich, T., Viescinski, A. B., and Maziero, C. A. (2021). Trusted inter-process communication using hardware enclaves. In *2021 IEEE International Systems Conference (SysCon)*, pages 1–7. IEEE.

Wong, W.-K., Moore, A., Cooper, G., and Wagner, M. (2002). Rule-based anomaly pattern detection for detecting disease outbreaks. In *AAAI/IAAI*, pages 217–223.

Wu, S. and Wang, S. (2011a). Information-theoretic outlier detection for large-scale categorical data. *IEEE transactions on knowledge and data engineering*, 25(3):589–602.

Wu, S. and Wang, S. (2011b). Parameter-free anomaly detection for categorical data. In *International Workshop on Machine Learning and Data Mining in Pattern Recognition*, pages 112–126. Springer.

Xie, Q., Tang, S., Zheng, X., Lin, Q., Liu, B., Duan, H., and Li, F. (2022). Building an open, robust, and stable voting-based domain top list. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 625–642.

Yan, Y., Tu, T., Zhao, L., Zhou, Y., and Wang, W. (2021). Understanding the performance of WebAssembly applications. In *Proceedings of the 21st ACM Internet Measurement Conference*, pages 533–549.

Yassin, W., Udzir, N. I., Muda, Z., Sulaiman, M. N., et al. (2013). Anomaly-based intrusion detection through k-means clustering and naives Bayes classification. In *Proc. 4th Int. Conf. Comput. Informatics, ICOCI*.

Yu, G., Yang, G., Li, T., Han, X., Guan, S., Zhang, J., and Gu, G. (2020). MinerGate: A novel generic and accurate defense solution against Web based cryptocurrency mining attacks. In *Proceedings of the 17th China Cyber Security Annual Conference*, pages 50–70, Beijing, China. Springer.

Zhang, C., Niknami, M., Chen, K., Song, C., Chen, Z., and Song, D. (2015). Jitscope: Protecting web users from control-flow hijacking attacks.

Zhang, Y., Luo, S., Pan, L., and Zhang, H. (2021). Syscall-bsem: Behavioral semantics enhancement method of system call sequence for high accurate and robust host intrusion detection. *Future Generation Computer Systems*, 125:112–126.

Zheng, S., Wang, H., Wu, L., Huang, G., and Liu, X. (2020). VM matters: A comparison of WASM VMs and EVMs in the performance of blockchain smart contracts.

Šilić, M., Krolo, J., and Delač, G. (2010). Security vulnerabilities in modern web browser architecture. In *The 33rd International Convention MIPRO*, pages 1240–1245.

APPENDIX A – LIST OF ATTACKS PRESENTED IN THE DATASET

Table A.1 presents the Common Weakness Enumeration (CWE) of the portion of attacks selected from the different data sources to represent the anomaly class of the dataset.

CWE	**Description**
CWE 121	Stack-based Buffer Overflow
CWE 122	Heap-based Buffer Overflow
CWE 123	Write-what-where Condition
CWE 124	Buffer Underwrite (‘Buffer Underflow’)
CWE 126	Buffer Over-read
CWE 127	Buffer Under-read
CWE 129	Improper Validation of Array Index
CWE 134	Use of Externally-Controlled Format String
CWE 135	Incorrect Calculation of Multi-Byte String Length
CWE 193	Off-by-one Error
CWE 194	Unexpected Sign Extension
CWE 195	Signed to Unsigned Conversion Error
CWE 252	Unchecked Return Value
CWE 253	Incorrect Check of Function Return Value
CWE 390	Detection of Error Condition Without Action
CWE 391	Unchecked Error Condition
CWE 400	Uncontrolled Resource Consumption
CWE 401	Missing Release of Memory after Effective Lifetime
CWE 404	Improper Resource Shutdown or Release
CWE 416	Use After Free
CWE 457	Use of Uninitialized Variable
CWE 475	Undefined Behavior for Input to API
CWE 476	NULL Pointer Dereference
CWE 511	Logic/Time Bomb
CWE 805	Buffer Access with Incorrect Length Value
CWE 806	Buffer Access Using Size of Source Buffer

Table A.1: List of Attacks presented at the Dataset.

APPENDIX B – USING CATEGORICAL DATA FOR CYBERSECURITY SOLUTIONS

We did not find a systematic literature review concerning the use of categorical data for cybersecurity solutions. Thus, considering the definitions presented in Section 4.2, we performed a systematic review on this subject, based on the following research questions (RQ):

- **RQ1:** How categorical data is being used in cybersecurity solutions? and
- **RQ2:** How categorical data is being used for anomaly detection?

The research string consists of *Categorical Data*, as the primary topic, and as the secondary topic *Cybersecurity* or *Anomaly Detection*. The search string was: *categorical data AND (Cybersecurity OR Anomaly Detection)*.

The search engines used follow the criteria set out in Section 4.2. The Selection Criteria (SC) were selected for this research:

- **SC1:** Written in English;
- **SC2:** Primary studies (i.e., not surveys, meta-analysis, systematic mappings or reviews);
- **SC3:** Not duplicated;
- **SC4:** The paper was widely available for download; and
- **SC5:** Central theme is associated with Categorical Data.

B.1 RESEARCH USING CATEGORICAL DATA

In a classification pipeline, the data model used for representation is an important aspect, and using categorical data is one of the options (Charu, 2017). Within the scope of our review, categorical data is used mostly for anomaly and outlier detection. We also found discussions about the use of categorical data with a statistical focus.

Overall, proposals in this scope focus on the optimization of previous data mining solutions (He et al., 2005), the comparison of different detection strategies (Koufakou et al., 2007), and the definition of new models based on this type of data (Wu and Wang, 2011a).

B.1.1 Categorical data for Anomaly Detection

A frequent goal in categorical data analysis is to identify differences between groups, which according to the type of data may contain different classes of objects. (Bay and Pazzani, 1999) present a strategy for mining categorical data, to identify significant differences in the data. The contribution of the work is a methodology for the identification of contrasting groups, which allows for the distinction and comparison of such groups.

(Liao and Vemuri, 2002) apply text categorization for intrusion detection using system calls. Instead of focusing on the order in which the calls are presented, the proposal

considers the frequency that system calls are issued by the application to characterize whether an application has anomalous behavior or not. The second contribution of the work is the use of text processing categorization strategies to handle system calls. Finally, the K-Nearest Neighbors (KNN) algorithm is used as a classifier for intrusion detection using the set of proposed strategies.

(Wong et al., 2002) presents an algorithm for the identification of disease outbreaks, through the identification of anomalies. The identification process was carried out based on a set of predefined rules, which do not correlate the features to carry out the identification of an anomaly. The focus of the strategy is on the grouping of specific features that help in the process of identifying anomalies. The proposed *What's strange about recent events (WSARE)* algorithm was able to identify new patterns, and overfitting caused by intensive multiple testing.

The use of strategies based on association rules are recurrently applied in data mining tasks. The problem when using such strategies is the volume of rules obtained when evaluating a data set. Thus, evaluating strategies to minimize this problem is the approach proposed in (Balderas et al., 2005). In addition to presenting a set of correlated solutions for this problem, an algorithm is proposed to identify such rules.

(He et al., 2005) propose a local search algorithm to find outliers in categorical data. Their work main contribution is the focus on the optimization of models for outlier detection in categorical data, and the use of this strategy in real datasets.

(Koufakou et al., 2007) introduce *Attribute Value Frequency (AVF)* for outlier detection in categorical data. AVF is a linear time solution that provided an improvement over existing methods, which were more complex. The scalable outlier detection algorithm proposed ends up being faster and more accurate than other proposals.

Taking the focus away from identifying anomalies, (Das and Schneider, 2007) focuses on identifying unusual factors. This focus on unusual factors already is the objective of systems such as network monitoring systems. However, it has a strong link with the definition of what this unusual group would be. The proposal starts from an unlabeled dataset, and assumes that the anomalous set has a minor presence in the dataset. Using a probability-based approach and an association between attributes, the work manages to present a method for identifying anomalies in categorical data without a label.

A new methodology for detecting patterns in categorical data is presented by (Das et al., 2008). The proposed strategy is taking into account that an anomaly in a process is a subset of altered data. This identification process is carried out from the moment when operations outside the expected are identified in the sample being evaluated, and later this anomaly is used for searches in other data samples.

Focusing on the identification of outliers in categorical data sets, (Wu and Wang, 2011b) discusses and presents a strategy for the identification of significant similarity measures for this problem. The proposed methodology uses entropy and total correlation to characterize outliers effectively. This strategy turns out to be relevant when considering the difficulty in defining that an outlier exists in a categorical dataset, since a categorical set does not allow the use of a distance measure and outlier classifications for categorical data do not have a formal definition.

Taking into account multidimensional datasets, (Akoglu et al., 2012) uses pattern-based compression to perform anomaly identification. The authors propose *CompreX*, a parameter-free method that uses a dictionary to encode data and, through this encoding, identifies anomalies. The proposed method is scalable and adequate results were found when dealing with different types of categorical data. The work is pioneer in the use of

discrete data for the identification of anomalies, having a valid solution for both numerical and categorical data.

Considering the relation between categorical variables, (Taha and Hadi, 2016) propose two measurements used in mixed variables. The main contribution of the work is the study of relationship in categorical variables, and ways of representation that could be used for this type of data.

Outlier mining it is being widely used to solve problems in intrusion detection, fraud, and fault detection. (Li et al., 2018) focus on correlation among features when considering outlier mining. The work propose WATCH, a weighted outlier mining method that uses feature grouping for categorical data. The results achieved by WATCH are similar to the ideal results of the dataset, and WATCH outperform others proposal like AVF (Koufakou, 2009).

B.1.2 Cybersecurity and Categorical Data

An expansion of the work by (Forrest et al., 1996) is presented by (Lee et al., 1997). The work applies a machine learning strategy to identify security anomalies using system calls. Two contributions that can be (highlighted|mentioned) are a discussion of the impact of non-generic models for the definition of normal behaviors and an investigation of the impact of the size of sliding windows on the threat identification process. Another contribution of the study consists of a brief presentation of related works that present new approaches. (Oates and Cohen, 1996) is one of these works, which presents a Multi-Stream Dependency Detection (MSDD) algorithm that is responsible for performing a systematic search to identify dependencies in a categorical dataset.

We also found research that uses categorical data, however, no discussion about the use of this type of data representation is presented (Bernaschi et al., 2002; Castanhel et al., 2021; Lemos et al., 2023). The classification was used for pattern recognition, anomaly detection, and feature classification.

B.2 CONSIDERATIONS

The references reviewed in Section B.1 show how anomalies and outliers can be addressed when working with categorical data, and how this approach can be used for anomaly detection strategies. We can thus propose the following answers to the research questions defined in Section B:

RQ1: How categorical data is being used in cybersecurity solutions? There are not many references that use categorical data in cybersecurity solutions. Those that use categorical data for classification lack in-depth discussion about this choice of representation.

RQ2: How categorical data is being used for anomaly detection? Overall, anomaly detection strategies take advantage of categorical data, with more studies published. The area takes advantage of categorical data models that can be used for data analysis and are well-fitted to handle this data type. Despite being limited to specific areas where categorical data representation appears more often, anomaly and outlier detection are subjects with more discussion.

APPENDIX C – SYSTEM CALLS AND WASI CALLS CLASSIFICATION

#	#	System Calls	WASI Calls
A	1	chmod, chown, chown32, execveat, fanotify_init, fanotify_mark, fchmod, fchmodat, fchown, fchown32, fchownat, fsetxattr, fsmount, fsopen, lchown, lchown32, link, linkat, lsetxattr, mount, perf_event_open, pivot_root, rename, renameat, renameat2, setxattr, splice, symlink, symlinkat, tee, unlink, unlinkat, userfaultfd, utimensat, vmsplice	path_link, path_rename, path_symlink, path_unlink_file, path_remove_directory, path_filestat_set_times, args_get, environ_get
	2	execve, getresgid, getresgid32, getresuid, getresuid32, ioperm, keyctl, rseq, seccomp, set_thread_area, set_tid_address, setfsgid, setfsgid32, setfsuid, setfsuid32, setgid, setgid32, setgroups, setgroups32, setpgid, setregid, setregid32, setresgid, setresgid32, setresuid, setresuid32, setreuid, setreuid32, setsid, setuid, setuid32, unshare	
	3	delete_module, finit_module, init_module, inotify_init, inotify_init1, request_key	
	4	modify_ldt, move_mount, move_pages, pkey_mprotect, process_vm_writev	
	5	getitimer	
	6	accept, accept6, mq_getsetattr	
	7	pkey_alloc, pkey_free, restart_syscall, semctl, setns, setpriority, sethostid	
	9	create_module	
	10	ioctl, iopl, epoll	
B	1	_llseek, chdir, chroot, epoll_ctl, fcntl, fcntl64, ftruncate, ftruncate64, lseek, mkdir, mkdirat, mknod, mknodat, pread, pread64, preadv, preadv2, pwritev, pwritev2, readv, sync_file_range, sync_file_range2, syncfs, truncate, truncate64, umask, utime, utimes, write, writev, epoll_ctl_old	fd_fdstat_set_flags, fd_tell, fd_seek, path_create_directory, fd_pread, fd_pwrite, sock_recv, fd_read, sock_send, fd_write, fd_filestat_get, path_create_directory
	2	arch_prctl, capset, exit, exit_group, fadvise64, fadvise64_64, fchdir, personality, prctl, rt_sigaction, rt_sigpending, rt_sigprocmask, rt_sigqueueinfo, rt_sigreturn, rt_sigsuspend, rt_sigtimedwait, rt_tgsigqueueinfo, sched_yield, sigaction, sigaltstack, signal, signalfd, signalfd4, sigpending, sigprocmask, sigreturn, sigsuspend, ssetmask, uselib, wait4, waitid, waitpid	fd_advise, sched_yield
	3	add_key, inotify_add_watch, inotify_rm_watch	
	4	brk, mprotect, mremap, munmap, sbrk	
	5	clock_nanosleep, clock_settime, io_pgetevents, nanosleep, nice, setitimer, timer_create, timer_delete, timer_settime, timerfd_create, timerfd_settime	
	6	connect, mq_notify, mq_open, msgctl	
	7	semget, semop, semtimedop	
	9	_sysctl	
	10	io_cancel, io_destroy, io_getevents, io_setup, io_submit, io_uring_enter, io_uring_register, io_uring_setup, msync, poll, ppoll	
C	1	creat, dup, dup2, dup3, epoll_create, epoll_create1, eventfd, eventfd2, fallocate, flock, fremovexattr, lremovexattr, name_to_handle_at, open, open_by_handle_at, open_tree, openat, openat2, pidfd_getfd, pidfd_open, quotactl, read, readahead, removexattr, rmdir, sendfile, sendfile64, umount, umount2	fd_renumber, fd_allocate, path_open, random_get
	2	bind, clone2, clone, clone3, futex, getrlimit, kill, prlimit64, ptrace, reboot, sched_setaffinity, sched_setattr, sched_setparam, sched_setscheduler, set_robust_list, setrlimit, fork, vfork, vhangup, vm86old, vm86, _exit, __clone2	
	4	madvise, mbind, mincore, mlock, mlock2, mlockall, mmap, mmap2, munlock, munlockall, process_vm_readv, set_mempolicy, swapoff, swapon	
	5	adjtimex, clock_adjtime, pause, settimeofday, stime, ntp_adjtime	
	6	bpf, listen, mq_timedreceive, mq_timedsend, mq_unlink, msgrcv, msgsnd, pidfd_send_signal, recv, recvfrom, recvmsg, recvmmsg, send, sendmmsg, sendmsg, sendto, setsockopt, socket, socketcall, socketpair, killpg, raise	sock_recv, sock_send, proc_raise
	7	fspick, setdomainname, sethostname, syslog, tgkill, tkill, gethostid, ssize_t	
	9	nfsservctl	
	10	pselect6, select	
D	1	close, copy_file_range, fgetxattr, getdents, getdents64, getxattr, lgetxattr, listxattr, llistxattr, lookup_dcookie, lstat, lstat64, newfstatat, oldfstat, oldlstat, oldstat, pipe, pipe2, readdir, readlink, readlinkat, stat, stat64, statfs, statfs64, statx, sync, sysfs, ustat	fd_close, path_readlink, path_filestat_get, args_sizes_get, environ_sizes_get, fd_prestat_get fd_prestat_dir_name fd_readdir
	2	access, acct, alarm, capget, cmpxchg_badaddr, faccessat, flistxattr, fstat, fstat64, fstatat64, fstatfs, fstatfs64, get_robust_list, getcpu, getcwd, getegid, getegid32, geteuid, geteuid32, getgid, getgid32, getgroups, getgroups32, getpgid, getpgrp, getpid, getppid, getrandom, getrusage, getsid, gettid, getuid, getuid32, sched_get_priority_max, sched_get_priority_min, sched_getaffinity, sched_getattr, sched_getparam, sched_getscheduler, sched_rr_get_interval, sgetmask, ugetrlimit, pdflush	
	4	cacheflush, get_mempolicy, get_thread_area	
	5	clock_getres, clock_gettime, gettimeofday, time, timer_getoverrun, timer_gettime, timerfd_gettime, times	clock_res_get, clock_time_get
	6	getpeername, getsockname, getsockopt, msgget	
	7	fsconfig, getpriority, oldolduname, olduname, shutdown, sysinfo, uname	sock_shutdown
	8	afs_syscall, break, ftime, getpmsg, gtty, idle, lock, madvise1, mpx, phys, prof, profil, putpmsg, security, stty, tuxcall, ulimit, vserver, unimplemented, ftime, ulimit	
	9	bdflush, futimesat, get_kernel_syms, query_module, remap_file_pages, xtensa	fd_filestat_set_times
	10	epoll_pwait, epoll_wait, fdatasync, fsync, epoll_wait_old	fd_datasync, fd_sync, fd_fdstat_get
E	1	oldumount, pwrite, pwrite64, s390_guarded_storage, spu_create, spu_run	
	2	arc_gettls, arc_settls, arc_usr_cmpxchg, atomic_barrier, atomic_cmpxchg_32, bfin_spinlock, breakpoint, execv, old_getrlimit, perfctr, perfmonctl, s390_runtime_instr, sched_get_affinity, sched_set_affinity, setpgrp, switch_endian	
	3	s390_sthyi	
	4	dma_memcpy, getpagesize, memory_ordering, metag_get_tls, metag_set_fpu_flags, metag_set_tls, metag_setglobalbit, membarrier, memfd_create, migrate_pages, riscv_flush_icache, s390_pci_mmio_read, s390_pci_mmio_write, shmat, shmctl, shmdt, shmget, spill, sram_alloc, sram_free, subpage_prot	
	5	old_adjtimex	
	7	getdomainname, gethostname, getxgid, getxpid, getxuid, rtas, sethae, swapcontext, makecontext, sys_debug_setcontext, syscall, usr26, usr32, utrap_install	
	9	alloc_hugepages, free_hugepages, get_tls, getdtablesize, getunwind, set_tls, setup, sysmips	
	10	__newselect, or1k_atomic, pciconfig_iobase, pciconfig_read, pciconfig_write	

Table C.1: System Calls and WASI Calls classification.