






The use of the DWARF Debugging Format for the Identification of Potentially Unwanted Applications (PUAs) in WebAssembly Binaries

Calebe Helpa¹, Tiago Heinrich² ^a, Marcus Botacin³ ^b, Newton C. Will⁴ ^c, Rafael R. Obelheiro⁵ ^d,
and Carlos A. Maziero¹ ^e

¹Computer Science Department, Federal University of Paraná - Curitiba, Brazil 81530-015

²Max Planck Institute for Informatics (MPI) - Saarbrücken, Germany 66123

³Texas A&M University - College Station, TX, USA 77843

⁴Computer Science Department, Federal University of Technology - Paraná - Dois Vizinhos, Brazil 85660-000

⁵Computer Science Department, State University of Santa Catarina - Joinville, Brazil 89219-710
cph19@inf.ufpr.br, theinric@mpi-inf.mpg.de, botacin@tamu.edu, will@utfpr.edu.br, rafael.obelheiro@udesc.br,
maziero@inf.ufpr.br

Keywords: WebAssembly, Intrusion Detection, Security

Abstract: Debugging formats are well-known means to store information from an application, that help developers to find errors, bugs, or unexpected behavior during the development period. The Debugging With Attributed Record Format (DWARF) is an example of a generic format that can be used for a range of programming languages and formats, such as WebAssembly, a low-level binary format that provides a compilation target for high-level languages. Given the use of debugging formats, their potential for intrusion detection is still unknown. Our study consists of evaluating the use of data extracted with the DWARF format, and their respective potential for an intrusion detection solution. In this context, we present a strategy for identifying Potentially Unwanted Application (PUA) in WebAssembly binaries, through feature extraction and static analysis using the DWARF format as a data source from WebAssembly binary. Our results are promising, with an overall f1score performance above 96% for the algorithms.

1 INTRODUCTION


WebAssembly is a low-level binary format that provides a compilation target for high-level languages (Hoffman, 2019). It aims to support web applications, offering fast processing support and decreasing memory usage to load web pages (Falliere, 2018). At the same time, it works in different browsers (Romano et al., 2022).


DWARF is a debugging information file format, supported by different compilers and debuggers to allow developers access to a high-level debugger (DWARF, 2023). It is supported by languages such as C, C++, Fortran, and WebAssembly. The main use of such a format is during the debugging process, where breakpoints can be set, operation data can be


viewed, or the tracing of code sections can be made. The format represents information using a tree, where the nodes represent data, types, and functions (Eager, 2012). While DWARF information is usually produced during compilation from source, one may generate DWARF information from decompiled binaries if source code is lacking.


The use of debugging format outside the development environment is limited. Taking into account the information that this type of format offers from applications, evaluation strategies can benefit. Specifically, security solutions can use this information for a threat investigation and detection process.


Rogue software that may compromise the privacy of a system or weaken its security is denoted Potentially Unwanted Application (PUA) (Pickard and Miladinov, 2012). In most cases, the identification of PUAs involves evaluating binaries using static and/or dynamic analysis. The former depends on the extraction of information found in the binaries, while the latter observes their behavior during execution.

^a  <https://orcid.org/0000-0002-8017-1293>

^b  <https://orcid.org/0000-0001-6870-1178>

^c  <https://orcid.org/0000-0003-2976-4533>

^d  <https://orcid.org/0000-0002-4014-6691>

^e  <https://orcid.org/0000-0003-2592-3664>

WebAssembly applications have the potential for malicious use due to the binary format, which makes it difficult to identify the purpose of each program. In this way, malicious users exploit the design of the WebAssembly format to carry out cryptojacking attacks and the obfuscation of malicious code (Naseem et al., 2021). To mitigate such risks, recent studies focus on correcting design flaws, adding new features and improving already implemented features, evaluating memory-related issues (Michael et al., 2023) and improving the design of the compiler (Bosamiya et al., 2022).

Our proposal consists of evaluating the potential of using information extracted from debugging formats for an intrusion detection solution. We use the DWARF format and identify a set of features that can be used to represent the application and applied in the identification of PUAs. Since WebAssembly applications are being widely adopted by major web browsers, and considering the security problems observed in the field, we focus on the use of WebAssembly binaries as a case study. We extracted information from real applications and evaluated how Machine Learning (ML) models can be used for an intrusion detection process. ML strategies have promising results in the security area, allowing the classification of application behaviors (Ceschin et al., 2024).

Our contributions are:

- The evaluation of the DWARF debugging format for extracting information from WebAssembly binaries, for a subsequent intrusion detection process; and
- A strategy for PUA identification in WebAssembly binaries.

Our proposal presented a better understanding of how information retrieved from debug formats such as DWARF can be used for an intrusion detection process. We achieve interesting results for PUA identification, with f1score above 95% and accuracy above 96% for the algorithms tested with cross-validation.

The remainder of this paper is structured as follows: Section 2 presents the background; Section 3 discusses the proposal; Section 4 presents the evaluation; Section 5 reviews related work; and Section 6 concludes the paper.

2 BACKGROUND

This section presents the background for understanding the work, including the DWARF format, WebAssembly format, intrusion detection, and static analysis.

2.1 DWARF Format

The Debugging With Attributed Record Format (DWARF) is a debugging information file that allows source-level debugging (DWARF, 2023). Debuggers and compilers can use the format to represent the applications in a tree structure, in which types, variables, and functions form the data sections of the DWARF format (Eager, 2012).

In DWARF, a Debugging Information Entry (DIE) describes an attribute in a program. The structure of DIE makes it a parent or property of one. In this way, the structure of the program is maintained for an evaluation process. The description found in a DIE will present information about attributes such as variables, constants, and references.

Code 1 presents a function that returns an integer that is printed as output. The output of this code in DWARF file format is presented in Code 2.

```
int foo() {
    int x=1;
    return x;
}
int main() {
    printf("Value:_%d", foo());
    return 0;
}
```

Code 1: An example of code in C with a function that returns a value to be printed.

As shown in Code 2, the entire structure of the C program is represented through the DWARF format. From the function `foo`, to the variables present throughout the code is presented in a tree-like structure. With each DIE representing the information in the current scope of the application. In this way, the DWARF format presents the information in a way that it is possible to follow the process of executing the C code alongside the DWARF format.

In addition to this information being useful for the debugging process, they appear to have the potential for intrusion detection solutions, since they are capable of representing the key functionalities presented in a program. For other languages, a similar output is expected, with additions for language features. WebAssembly applications can be converted to support the DWARF format even with access only to the application binary.

2.2 WebAssembly

WebAssembly is a binary format targeted at the Web. WebAssembly code can be generated from the

```

<1>:TAG_compile_unit [1] *
  AT_producer("Apple LLVM version 9.0.0 (clang
    -900.0.39.2)")
  AT_language(DW_LANG_C99)
  AT_name("main.c")
  AT_stmt_list(0x00000000)
  AT_comp_dir("/Users/bar/Documents/")
  AT_low_pc(0x0000000000000000)
  AT_high_pc(0x00000041)
<2>: TAG_subprogram [2] *
  AT_low_pc(0x0000000000000000)
  AT_high_pc(0x00000010)
  AT_frame_base(rbp)
  AT_name("foo")
  AT_decl_file("main.c")
  AT_decl_line(7)
  AT_type({0x0000006b})(int)
  AT_external(true)
<3>: TAG_variable [3]
  AT_location(fpreg -4)
  AT_name("x")
  AT_decl_file("main.c")
  AT_decl_line(8)
  AT_type({0x0000006b})(int)
<4>: NULL
<5>: TAG_subprogram [4]
  AT_low_pc(0x0000000000000010)
  AT_high_pc(0x00000031)
  AT_frame_base(rbp)
  AT_name("main")
  AT_decl_file("main.c")
  AT_decl_line(12)
  AT_type({0x0000006b})(int)
  AT_external(true)
<6>: TAG_base_type [5]
  AT_name("int")
  AT_encoding(DW_ATE_signed)
  AT_byte_size(0x04)
<7>: NULL

```

Code 2: DWARF output example, based on the source code presented in Code 1.

WebAssembly Text (WAT) textual format or by compilers that allow the translation of codes from high-level languages such as C, C++, Go, and Rust to WebAssembly (Hoffman, 2019).

A module consists of a WebAssembly application, which contains function definitions, global variables, linear memories, and indirect call tables. Functions and variables, as well as other program elements, are identified by indices represented by integer numbers (Lehmann et al., 2020). A WebAssembly module usually has three sections: `Preamble` with module start information, `Default` which contains all application information such as functions, and `Custom` which has information for debug (Kim et al., 2022). Figure 1 shows in detail the structure of a WebAssembly binary.

Only four primitive types are supported: *i32*, *i64*, *f32*, and *f64*, representing a 32-bit or 64-bit integer or a 32-bit or 64-bit floating point number, respectively.

WebAssembly uses a format of binary code instructions that can be debugged using converters that make the machine code readable. Tools are available and make it more practical to analyze sections of WebAssembly code (Falliere, 2018). The WebAssembly instruction format was designed with a focus on ensuring the safety of its users. Its key features for security are:

Virtualized environment: WebAssembly modules run in a virtual machine based on the stack model. All input/output interactions and access to operating system resources must be performed through functions incorporated by WebAssembly, which must be imported by the module. Therefore, WebAssembly is able to establish security policies for developers and is able to assure users that their environment and system resources are being accessed by modules in a limited and controlled way (Rossberg, 2018).

Linear memory: The linear memory of WebAssembly modules is instantiated in managed buffers. This way, read and write operations are limited to certain areas of memory (Kim et al., 2022).

Control Flow Integrity (CFI): Through a structured control flow generated during the compilation process, WebAssembly modules are protected against attacks such as shellcode injection or the abuse of unrestricted jumps carried out indirectly (Kim et al., 2022). However, the WebAssembly CFI is not as effective as modern CFIs used for native binary defenses, with some calls vulnerable to malicious use (Lehmann et al., 2020).

2.3 Intrusion Detection

Intrusion Detection Systems (IDS) are security mechanisms that have the purpose of monitoring hosts, applications, and networks for signs of attacks and intrusions (Stallings et al., 2012). One way of performing intrusion detection on applications is by analyzing application code using static and/or dynamic analysis (Chandola et al., 2009; Liu et al., 2018; Castanhel et al., 2021; Lemos et al., 2022).

Static analysis is performed by extracting features from the code, but without executing them, thus defining an abstract representation of the program's behavior (Kirchmayr et al., 2016). This technique has been widely used, especially in critical systems such as those used in aviation and air traffic control.

Table 1: Representation of a WebAssembly binary. Structurally describing the expected composition of a WebAssembly binary file.

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

Dynamic analysis is a software analysis technique that allows evaluating the behavior of the program during its execution. Dynamic analysis is performed through tests and simulations, which makes it possible to identify programming errors, unexpected behaviors, and security flaws during the interaction between the code and the environment in which it is executed (Kirchmayr et al., 2016; Lemos et al., 2023).

3 PROPOSAL

This section presents our proposal. Section 3.1 describes the threat model. Section 3.2 presents the studied strategy. Section 3.3 discusses the characteristics used by binaries in the detection process.

3.1 Threat model

In the threat model, we consider that the adversary explores the WebAssembly format for the deployment of malicious content. It is assumed that access to the DWARF data will always exist, since even a WebAssembly binary without the DWARF information, in a later process will be possible to generate the data. Our PUA evaluation will be made by a static strategy, where the information available in the DWARF format will be accessed and used in a detection process.

3.2 Strategy

Our strategy is to evaluate how debug formats, such as DWARF, can be used in an intrusion detection solution. The advantage of using debug formats is directly associated with data access since the format will offer access to information that would not be accessible from other observation points.

Debug formats allow the intrusion detection strategy to have an almost complete understanding of the application and its execution process since all code structures will be taken into account by the debugging tools. Therefore, the use of this data for detection may prove useful in situations in which the source is unknown or code evaluation is possible. Achieving these goals involves two main steps:

- extracting key characteristics from binaries in DWARF format; and

- defining an evaluation strategy to detect PUAs based on these characteristics.

To evaluate the potential of the DWARF format, we selected WebAssembly binaries from well-known datasets (Lehmann and Pradel, 2022; Stiévenart et al., 2022), and extracted debug information from each binary in DWARF format using *llvm-dwarfdump*¹. The information extracted will be discussed in Section 3.3.

For WebAssembly binaries that lack debugging information, we apply the process shown in Figure 1. The binary is decompiled using *wasm2wat*², producing source in WAT format (1). The WAT source is compiled with debugging symbols using *wasmtime*³ (2), and the DWARF information is extracted from the new binary using *llvm-dwarfdump* (3).

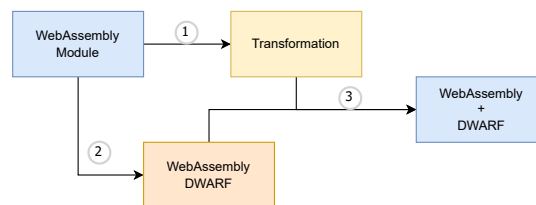


Figure 1: Process of transformation of a binary to WebAssembly to generate a binary supporting the DWARF format.

Our PUA identification strategy uses machine learning algorithms. We selected multi-class algorithms to better understand the impact of the information extracted from the binaries in the models. The choice of algorithms is based on previous related work (Galante et al., 2019; Castanhel et al., 2020; Lemos et al., 2023; Heinrich et al., 2024).

3.3 Binary characteristics

After converting the WebAssembly binaries to support the DWARF format, information can be extracted from them. The DWARF format provides information from a set of tags and attributes present in a WebAssembly binary (as presented in Section 3.2). This information allows a later application analysis process, for the identification of implementation errors or, in the case of this work, for the identification of PUAs that may reflect an intrusion.

¹<https://llvm.org/docs/CommandGuide/llvm-dwarfdump.html>

²<https://github.com/WebAssembly/wabt>

³<https://wasmtime.dev/>

The DWARF format offers different tags according to the format of the language being analyzed (as presented in Section 2.1). Instead of using all the tags available for WebAssembly, we chose to select the relevant attributes for classification. For this selection process, we collected information about all the tags available for the WebAssembly format and evaluated the importance of these features for the models.

We limited our selection based on specific sections of the WebAssembly binary (as presented in Table 1). The data presented in the preamble and standard section presents the key characteristics of the application and the functionality of the WebAssembly module, such as functions, variables, export data, memory allocation, and binary compilation.

After this process, we evaluated the tags that were relevant to our classification strategy (Section 3.2). This process generated a set of key information from the DWARF format that defined six groups covering the extracted information⁴.

General Information: from the application, such as program size, the number of labels in the code, and the source code language. In addition to describing the application, we hypothesize that this information can help classifiers as it offers a view of the language used before the port to WebAssembly;

Routines and subprograms: considering the number of declared variables, the number of declarations of inline subroutines, the number of declarations of subprograms, and the number of parameters of these subprograms. This information is important to describe the operations that a binary can perform;

Variables: which includes the number of type declarations, declared integer types, declared unsigned integer types, declared word types, and declared file pointer types. As WebAssembly has a restricted set of native data types, it is necessary to define these types in WebAssembly modules, consequently, this information also helps in describing the operations performed by a binary;

Parameters: which consist of the information found in each DIE, like operations performed by a function;

Memory Information: frequency of declared members when using structures or classes. We hypothesize that information found in memory can assist in a detection process; and

Shared data: includes data on the number of attributes that determine whether the subroutines are part of an external program or produce externally

⁴We also provide a list of tags used in the Appendix A.1.

accessible information and the number of tags and attributes related to the use of function calls.

When extracting information from the DWARF format, different regions of the application can be accessed in each DIE. The selected groups allow access to 29 tags in WebAssembly DWARF format. The number of tags that can be accessed by the DWARF format will vary according to the language supported.

The tags present specific information about the application behavior or represent some type of operation performed by the program. For an intrusion detection proposal, the raw information extracted can be used, or the frequency of appearance that may present a pattern.

4 EVALUATION

This section presents the evaluation of our proposal. Section 4.1 describes the objective of the evaluation. Section 4.2 presents the dataset. Section 4.3 discusses of the results.

4.1 Objective

Our purpose is to evaluate the usefulness of employing information extracted from the debugging format for PUA identification. We use static information from WebAssembly binaries to identify threats, investigating the feasibility of using information present in WebAssembly binaries through the DWARF format.

For the experiments we selected four machine learning algorithms: *Multi-layer Perceptron (MLP)*, *Random Forest (RF)*, *Support Vector Machines (SVM)* and *XGBoost*. These models aim to demonstrate how the learning process behaves when considering the characteristics selected for the benign and malicious classification classes.

4.2 Dataset

To perform the evaluation, a dataset is needed. Instead of building a completely new dataset, we used already available data from two sources (Lehmann and Pradel, 2022; Stiévenart et al., 2022). The binaries were selected taking into account the available samples to define both benign and malicious samples. To balance the classes of malicious and benign samples, we normalized the number of samples between the two classes. We selected samples to obtain a subset with the same characteristics as the entire dataset (our code is publicly available⁵).

⁵<https://github.com/CalebeHelpa/webassembly-classification>

After the selection process, the binaries were transformed to support the DWARF format (as presented in Section 3.2). In total, 770 samples were selected for the experiment. These samples are divided into 400 benign samples and 370 malicious samples. The purpose of the samples is to present recurring behaviors for benign applications and vulnerabilities or implementation errors for malicious samples.

Before running the experiments information needs to be extracted from the DWARF format of each WebAssembly application. Taking into account the binary characteristics presented in Section 3.3, we extracted all tags in the DWARF format that were incorporated into the six defined groups.

The features extracted from the binaries are saved to a file, which contains the appearance count of the attributes and subsequent encodings of extracted variables. After this process, the information was used for the model training and testing process.

4.3 Results

The classification results obtained by using machine learning algorithms, trained with the information extracted by the DWARF format, are presented in this section. The algorithms were trained and tested in a 1:1 ratio, that is, 50% of the data was used for training and 50% for testing. An exhaustive parameter search was made, aiming to find the best configurations for the models. We also perform the test with 10-fold cross-validation, as it is the standard for this type of assessment.

Table 2 presents the results achieved by the algorithms evaluated, considering the usual metrics in the area of machine learning. The results achieved varied according to the classification strategy used by the machine learning models. However, the results are favorable for a PUA detection strategy using the multi-class algorithms. Multi-class algorithms are trained to classify two or more classes, being capable of defining patterns that represent each of the classes.

The precision highlights that models like MLP and SVM had the biggest impact due to false positives, with the models classifying benign samples as malicious. Despite the small percentage of false positives, the error was responsible for the impact found in the F1Score.

The models were not affected by false negatives, as portrayed by the recall. The best metric to describe the model's result is the F1Score, in which we notice the impact of false positives. For the classification of binaries to perform the identification of PUA, the overall F1Score demonstrates that the features used to train and test the classifiers are sufficient for a classification

process.

The accuracy demonstrates the impact of true positives and true negatives, demonstrating that the models were able to adequately learn the patterns of the binaries through the extracted features. The values achieved by the Balanced Accuracy (BAC) are evidence that the model learning persists.

With these results, we conclude that debugging formats, such as DWARF, have the potential to extract information from binaries that later can be used for intrusion detection solutions. We also presented a static strategy for PUA detection in the WebAssembly application. Although we only explore the use of the DWARF format for WebAssembly applications, the information extracted through the use of the DWARF format showed promising results for the use of information extracted through debugging tools to detect PUA.

5 RELATED WORK

Some well-known strategies are used to collect data from applications and perform intrusion detection. Solutions that use system calls are an example, in which, traces of the application execution are used to define application behavior and identify anomalies that may correspond to an intrusion (Liu et al., 2018).

The use of bytecode is also explored in the intrusion detection field. Bytecode contains a sequence of instructions that represents the program without any redundancy, facilitating the process of compiling or interpreting source code into machine code. The solution proposed by (Ashouri et al., 2021) relies on Java bytecode to intercept runtime attacks. Bytecode is also used to detect malware on Android systems, by extracting features and using convolution neural networks to classify malicious applications (Ding et al., 2020).

In web applications, bytecode sequences are used to detect malicious behavior in JavaScript code, such as cross-site scripting and redirections (Rozi et al., 2020). The use of bytecode allows to bypass JavaScript obfuscation.

Approaches that focus on the static evaluation of WebAssembly binaries are aimed at identifying vulnerabilities in the developed codes that can be exploited or cause an error in production. These tools are also aimed at generating the flow of WebAssembly applications, aiming to identify vulnerabilities that may have been ported from other languages (Quan et al., 2019; Brito et al., 2022).

To the best of our knowledge, our work is the first to use the debugging format to extract features to apply in an intrusion detection system. Our work is able to

Table 2: Performance of algorithms for PUA using DWARF format data.

Classifier	Precision	10-Fold Cross-validation			BAC
		Recall	F1Score	Accuracy	
MLP	93.40%	100%	96.59%	96.62%	96.77%
RandomForest	98.24%	100%	99.11%	99.22%	99.31%
SVM	94.89%	100%	97.38%	97.66%	97.94%
XGBoost	97.34%	100%	98.65%	98.70%	98.76%

demonstrate the potential of using debug formats for intrusion detection.

6 CONCLUSION

In this paper, we present a novel intrusion detection approach using debugging formats to extract features from application code. To validate our proposal, we built a dataset with WebAssembly applications, a binary format that has seen rapid adoption on the Web. The features were extracted from DWARF format, a debugging information file format used by many compilers and debuggers to support source level debugging.

We evaluated our approach with multi-class machine learning algorithms, obtaining promising results, especially with ensemble algorithms. Thus, we showed the potential of using debugging formats to extract information from binaries to perform intrusion detection.

Unfortunately, using debugging formats has some limitations. Debugging symbols increase the size of compiled binaries and thus are usually stripped from distributed binaries to save space. Having a process for dealing with such binaries, as described in Section 3.2, alleviates this problem. The DWARF format may also, on rare occasions, not be capable of debugging the information itself. Some languages/compilers may not support the format, or support only a subset of its functionalities (Bastian et al., 2019); further experimentation is needed to investigate the impact of partial support for DWARF in our proposal.

ACKNOWLEDGEMENTS

This study was financed in part by the *Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Finance Code 001* and *Fundação de Amparo à Pesquisa e Inovação do Estado de Santa Catarina (FAPESC)*. The authors also thank the UDESC, UFPR and UTFPR Computer Science departments.

REFERENCES

- Ashouri, M., Kreitz, C., Austin, T. H., and Bordim, J. L. (2021). JACY: A robust JVM-based intrusion detection and security analysis system.
- Bastian, T., Kell, S., and Zappa Nardelli, F. (2019). Reliable and fast DWARF-based stack unwinding. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–24.
- 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.
- Brito, T., Lopes, P., Santos, N., and Santos, J. F. (2022). Wasmati: An efficient static vulnerability scanner for WebAssembly. *Computers & Security*, 118:102745.
- 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 *XVIII Regional School of Computer Networks*, pages 141–146, Virtual Event. SBC.
- Castanhel, G. R., Heinrich, T., Ceschin, F., and Maziero, C. A. (2021). Taking a peek: An evaluation of anomaly detection using system calls for containers. In *Proceedings of the 26th IEEE Symposium on Computers and Communications*, Athens, Greece. IEEE.
- Ceschin, F., Botacin, M., Bifet, A., Pfahringer, B., Oliveira, L. S., Gomes, H. M., and Grégio, A. (2024). Machine learning (in) security: A stream of problems. *Digital Threats*, 5(1).
- Chandola, V., Banerjee, A., and Kumar, V. (2009). Anomaly detection: A survey. *ACM Computing Surveys*, 41(3):1–58.
- Ding, Y., Zhang, X., Hu, J., and Xu, W. (2020). Android malware detection method based on bytecode image. *Journal of Ambient Intelligence and Humanized Computing*, 14(5):6401–6410.
- DWARF (2023). DWARF debugging information format. <https://dwarfstd.org/>. DWARF Debugging Information Format Committee.
- Eager, M. J. (2012). Introduction to the dwarf debugging format. <https://dwarfstd.org/doc/Debugging%20using%20DWARF-2012.pdf>.
- Falliere, N. (2018). Reverse engineering WebAssembly. <https://www.pnfsoftware.com/reversing-wasm.pdf>.
- 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, São Paulo, SP, Brazil. SBC.
- Heinrich, T., Will, N. C., Obelheiro, R. R., and Maziero, C. A. (2024). A categorical data approach for anomaly detection in WebAssembly applications. In *Proceedings of the 10th International Conference on Information Systems Security and Privacy*, pages 275–284, Rome, Italy. SciTePress.
- Hoffman, K. (2019). *Programming WebAssembly with Rust: Unified Development for Web, Mobile, and Embedded Applications*. The Pragmatic Bookshelf, Raleigh, NC, USA.
- 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.
- Kirchmayr, W., Moser, M., Nocke, L., Pichler, J., and Tober, R. (2016). Integration of static and dynamic code analysis for understanding legacy source code. In *Proceedings of the International Conference on Software Maintenance and Evolution*, pages 543–552, Raleigh, NC, USA. IEEE.
- Lehmann, D., Kinder, J., and Pradel, M. (2020). Everything old is new again: Binary security of WebAssembly. In *Proceedings of the 29th USENIX Security Symposium*, pages 217–234, Boston, MA, USA. USENIX Association.
- 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.
- Lemos, R., Heinrich, T., Maziero, C. A., and Will, N. C. (2022). Is it safe? identifying malicious apps through the use of metadata and inter-process communication. In *Proceedings of the 16th Annual IEEE International Systems Conference*, pages 1–8, Montreal, QC, Canada. IEEE.
- 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.
- 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*, 51(5):98.
- Michael, A. E., Gollamudi, A., Bosamiya, J., Johnson, E., Denlinger, A., Disselkoben, C., 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*, 7(POPL).
- Naseem, F. N., Aris, A., Babun, L., Tekiner, E., and Uluogac, A. S. (2021). MINOS: A lightweight real-time cryptojacking detection system. In *Proceedings of the Network and Distributed System Security Symposium*, Virtual Event. Internet Society.
- Pickard, C. and Miladinov, S. (2012). Rogue software: Protection against potentially unwanted applications. In *Proceedings of the 7th International Conference on Malicious and Unwanted Software*, Fajardo, PR, USA. IEEE.
- Quan, L., Wu, L., and Wang, H. (2019). EVulHunter: Detecting fake transfer vulnerabilities for EOSIO’s smart contracts at WebAssembly-level.
- 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.
- Rosberg, A. (2018). Webassembly specification. https://webassembly.github.io/spec/core/_download/WebAssembly.pdf.
- Rozi, M. F., Kim, S., and Ozawa, S. (2020). Deep neural networks for malicious javascript detection using bytecode sequences. In *Proceedings of the International Joint Conference on Neural Networks*, pages 1–8, Glasgow, UK. IEEE.
- Stallings, W., Brown, L., Bauer, M. D., and Bhattacharjee, A. K. (2012). *Computer Security: Principles and Practice*. Pearson.
- Stiévenart, Q., De Roover, C., and Ghafari, M. (2022). Security risks of porting C programs to WebAssembly. In *Proceedings of the 37th Symposium on Applied Computing*, pages 1713–1722, Virtual Event. ACM.

A Appendix

A.1 Tags Considered in the Training Process

lines, language, dw_tag_subprogram, dw_tag_typedef, dw_tag_member, dw_tag_label, dw_tag_gnu_call_site, dw_at_gnu_all_call_sites, dw_tag_inlined_subroutine, dw_at_external, dw_at_call_file, int_type, uint_type, string_type, fp_type, bool_type, dw_tag_enumerator, dw_tag_variable_int, dw_tag_variable_uint, dw_tag_variable_string, dw_tag_variable_fp, dw_tag_variable_bool, dw_tag_variable, dw_tag_formal_parameter_int, dw_tag_formal_parameter_uint, dw_tag_formal_parameter_string, dw_tag_formal_parameter_fp, dw_tag_formal_parameter, dw_tag_formal_parameter_bool