Future Generation Computer Systems 164 (2025) 107602

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Identifying runtime libraries in statically linked linux binaries

Javier Carrillo-Mondéjar [a,*], Ricardo J. Rodríguez [a,b]

[a] Dpto. de Informática e Ingeniería de Sistemas, Universidad de Zaragoza, Spain
[b] Aragón Institute of Engineering Research (I3A), Spain

ARTICLE INFO

Keywords:
Binary code analysis
Statically linked binaries
Runtime library identification
IoT
Malware

ABSTRACT

Vulnerabilities in unpatched applications can originate from third-party dependencies in statically linked applications, as they must be relinked each time to take advantage of libraries that have been updated to fix any vulnerability. Despite this, malware binaries are often statically linked to ensure they run on target platforms and to complicate malware analysis. In this sense, identification of libraries in malware analysis becomes crucial to help filter out those library functions and focus on malware function analysis. In this paper, we introduce MANTILLA, a system for identifying runtime libraries in statically linked Linux-based binaries. Our system is based on `radare2` to identify functions and extract their features (independent of the underlying architecture of the binary) through static binary analysis and on the K-nearest neighbors supervised machine learning model and a majority rule to predict final values. MANTILLA is evaluated on a dataset consisting of binaries built for different architectures (`MIPSeb`, `ARMel`, `Intel x86`, and `Intel x86-64`) and different runtime libraries (`uClibc`, `glibc`, and `musl`), achieving very high accuracy. We also evaluate it in two case studies. First, using a dataset of binary files belonging to the `binutils` collection and second, using an IoT malware dataset. In both cases, good accuracy results are obtained both in terms of runtime library detection (94.4% and 95.5%, respectively) and architecture identification (100% and 98.6%, respectively).

1. Introduction

For an attacker, the existence of a vulnerable unpatched application is the stuff of dreams. These vulnerabilities can originate from third-party program dependencies, which can inadvertently remain unfixed in the application when the binary is *statically linked*. When a binary is statically linked, all its library dependencies are included in the resulting binary as part of the compile and link process [1].

Static linking offers several advantages, as it ensures that binaries are self-contained (i.e., more portable) and makes it more difficult to reverse engineer (i.e., binary code from the programmer and from the libraries are mixed together) [2]. However, it increases the size of the binary and makes updating libraries more difficult, as the program must be relinked to take advantage of a library that was updated to fix a vulnerability. Despite these drawbacks, malware developers targeting Linux-based systems [3] or Internet-of-Things (IoT) devices [4] take advantage of static linking to ensure that their malware runs on the target platform, regardless of which libraries are installed. Furthermore, it complicates the tasks of malware analysts, making it difficult to correlate malware samples or identify reused code between different variants [4].

Therefore, library identification is essential for understanding functionality, assessing vulnerabilities, or detecting code reuse, among other purposes. In this sense, binary code analysis is a common approach to identify them [5], ranging from copyright infringement (for example, a commercial software inadvertently using library code from an open-source project [6]) to malware analysis [7]. For instance, it helps malware analysts to filter out those library functions and focus on malware function analysis. However, it is extremely challenging because binary code lacks high-level abstractions, such as data types and functions that are easily found in source code [8,9]. Also, compiler settings can drastically change the generated binary code [10].

To assist in this process, in this paper we present MANTILLA, a system for *runtiMe librAries ideNtification in sTatIcally-Linked Linux binAries*. MANTILLA identifies functions within a given binary and extracts features that are present in it, that are not specific to the underlying architecture of the binary (e.g., cyclomatic complexity, number of arguments, or entropy, to name a few, in contrast to assembly code bytes or mnemonics) via static binary analysis [5] using `radare2` [11], a popular reverse engineering framework. These features are then processed by the K-Nearest Neighbors (KNN) [12] supervised machine learning model to predict values, applying a majority rule for the final prediction. To validate it, we create a dataset made up of binaries built for different architectures (`MIPSeb`, `ARMel`, `Intel x86`, and

* Corresponding author.
E-mail addresses: jcarrillo@unizar.es (J. Carrillo-Mondéjar), rjrodriguez@unizar.es (R.J. Rodríguez).

https://doi.org/10.1016/j.future.2024.107602
Received 8 June 2024; Received in revised form 30 September 2024; Accepted 3 November 2024
Available online 13 November 2024

Intel x86-64) and different runtime libraries (uClibc [13,14], glibc [15], and musl [16]), obtaining very high accuracy in identifying the runtime library . As a real case study, we also test it on IoT malware, getting also good accuracy results (95.5% and 98.6% on runtime library and architecture identification, respectively).

In summary, the contribution of this paper is three-fold:

- We present MANTILLA, a system that automatically identifies the runtime library within a given binary using static binary analysis and KNN classification. Our system is specially designed for binary files of the C programming language with static linking and with or without symbols.
- We evaluate MANTILLA on a dataset of real-world (statically linked) IoT malware, observing that majority of them prefer using uClibc to glibc and musl, with the latter the most underrepresented runtime library. Our system achieves accuracy results close to 95.5% on runtime library identification. Furthermore, we also evaluate our system on a dataset created with tools from the binutils collection, achieving an accuracy of 94.4%.
- For the sake of open science and to encourage further research, we release MANTILLA and the dataset used in evaluation under the GNU/GPLv3 license [17,18].

The rest of the paper is structured as follows. Section 2 reviews related work. Section 3 introduces the threat model and the workflow of our system, detailing the features extracted from the binaries. The evaluation of MANTILLA is presented in Section 4, while Section 5 describes a real case study with IoT malware. Section 6 discusses the limitations of our system. Finally, Section 7 concludes the paper.

2. Related work

This section describes the most relevant and close works to ours. In particular, we discuss significant contributions in three areas of the field of static binary code analysis [5,13]: compiler provenance, authorship attribution, and library function identification. For an interested reader, a recent review of the state of the art in binary code analysis is provided in [19].

2.1. Compiler provenance

Seminal work in this area was presented by Rosenblum et al. in [20]. Based on a linear Conditional Random Field (CRF) model, the approach identifies the compiler family regardless of code stripping. It uses a dataset of diverse binaries compiled with different compilers (gcc, icc, and Microsoft Visual C++ (MSVC)) for Windows and Linux on Intel x86 architectures. This work was extended in [21] with Origin, a tool that identifies compiler family, compiler version, and optimization level using a Support Vector Machine (SVM) classifier model.

BinComp [22] performs a syntactic, structural, and semantic analysis of disassembled functions to extract compiler provenance. The Jaccard coefficient [23] is used to calculate function similarity scores. The dataset consists of diverse unstripped binaries compiled with various versions of gcc, icc, MSVC, and Clang using different optimization levels on Intel x86 and x86-64 architectures.

FOSSIL [24] utilizes syntactic and semantic analysis to identify free and open-source software (FOSS) packages and to determine compiler provenance in real-world projects and malware binaries. An adaptive hidden Markov and Bayesian models are used to calculate function similarity. The dataset includes FOSS packages compiled with different versions of MSVC and gcc.

Recent work has explored neural networks for compiler provenance identification. HIMALIA [25] uses recurrent neural networks to identify optimization levels in binaries. As a dataset, it uses binaries compiled with different versions of gcc and optimization levels for Linux on Intel x86-64 architecture. Likewise, BinEye [26] uses convolutional neural networks to identify optimization levels in ARM binaries compiled with buildroot (which uses gcc by default). NeuralCI [27] utilizes convolutional or recurrent neural networks to identify compiler family, version, and optimization level in binaries. The dataset consists of a set of binaries compiled with different versions of compilers (namely, gcc, ICC, and Clang) for Linux systems. Similarly, [28] presents a hybrid system that uses convolutional and long-short term memory networks to identify the compiler family and optimization level in code snippets on different architectures. As a dataset, it uses binaries compiled with gcc and Clang for Linux on x86-64 and AArch64, and gcc for Linux on multiple architectures, with different optimization settings.

Vestige [29] relies on attributed function call graphs and a graph neural network model for provenance identification. The dataset comprises stripped binaries compiled with various versions of gcc and clang. PassTell [30] is an approach that uses specially crafted features and gradient boosting decision trees for provenance identification. The evaluation involved unstripped binaries from popular open-source repositories compiled with Clang, gcc, and icc. Finally, BinProv [31] leverages a BERT-based embedding model for compiler and optimization level identification, fine-tuning the classifier model with embedding inputs from binary code. As a dataset, it uses a subset of the BINKIT benchmark [19] composed of binaries compiled with gcc and Clang for Linux on Intel x86-64 binaries. Apart from the BINKIT benchmark, [19] also presents an interpretable model that is based on syntactic and structural analysis, and calculates binary similarity through the relative difference between feature values.

2.2. Authorship attribution

Although the works described in this section primarily focus on authorship attribution, they also involve identifying compiler-embedded functions and other library functions.

The work described in [32] presents an approach that uses SVM models to automatically detect stylistic features in binary code and identify stylistic similarities between programs from unknown authors. The Mahalanobis distance is used as a similarity method. The dataset used for experimentation includes binaries from graduate courses and the Google Code Jam competition. The ParseAPI parser library is used to process the binaries, but no information is available on the specific build toolchain used.

OBA2 [33] is a three-layered approach that uses syntax and semantic features to automatically detect software library functions and other functions known to be unrelated to a specific author's style. It focuses on the C/C++ binaries from the 2014 Google Code Jam programming competition, but the compilers and the underlying architecture used are unknown.

In [34], the authors presented a set of code features (based on structural and semantic analysis at the basic block level) that are used to identify inline code from popular C++ template libraries (namely, STL and Boost). The dataset consists of C and C++ binaries compiled with gcc and statically linked for Intel x86-64 architectures. The approach does not rely on symbol information and uses supervised machine learning algorithms [35], such SVM and linear CRF, for function similarity.

BinAuthor [36] introduces a compiler-agnostic method to identify authors of program binaries by filtering out compiler-related features and tagging library-related features. As similarity methods, it uses the Mahalanobis [37] and the normalized compression [38] distances. The dataset includes binaries compiled with gcc and MSVC on Intel x86 architecture, using IDA Pro as a disassembler.

An approach to deanonymization using syntactic and structural analysis and employing the Random Forest classification is presented in [39]. The approach is evaluated using data from the Google Code Jam competition. The dataset used for evaluation consists of binaries that were compiled with gcc or g++ for Linux on Intel x86 architecture.

BinChar [40] identifies the characteristics of program authors based on structural and semantic characteristics using a convolutional neural network [41]. It combines annotated CFG and dataflow graphs and considers CFG level numerical features. Syntactic analysis is done with `IDA Pro`. It uses various compilers, including `g++`, `Clang`, `gcc`, `icc`, and `MSVC`, and can handle multiple architectures as it relies on some features that are independent between architectures.

2.3. Library function identification

A popular technique for library function identification is `IDA Pro` FLIRT [42], which forms signatures from the initial sequence of bytes of a function to recognize library functions. However, it has signature collision issues [43].

In [44], an approach based on syntactical and structural analysis is introduced. It extends UNSTRIP [45] to detect GNU C library wrapper functions in stripped binaries examining their interactions with system calls via pattern matching. It uses a relaxed pattern matching criterion where inexact matches are allowed and returns all matching fingerprints when multiple matches are found. Three versions of `glibc`, compiled with different versions of `gcc`, are used to build the test dataset.

BinHash [46] introduces a semantic hash function that operates at the function level to detect function similarities. It is tested against real-world functions from the CERT artifact catalog, but it does not specify the compilers, runtime libraries, or architectures used in the evaluation.

Another structural analysis approach, `libv` [47,48], applies subgraph isomorphism for library function identification. It employs the `IDA Pro` disassembler and different versions of `MSVC` compiler link libraries to build the test dataset. Similarly, discovRE [49] applies maximum common subgraph isomorphism to locate similar functions and tests it with `gcc`, `Clang`, `Intel ICC`, and `MSVC` binaries for Windows and Linux on `Intel x86`, `Intel x86-64`, and `ARM` architectures (when supported). `BinDiff` [50] and `BinSlayer` [51] are also based on function similarity detection by means of graph isomorphism but are not designed for labeling library functions.

Genius [52] is a structural and semantic bug-search approach that converts a CFG into high-level numeric features for similarity detection using Euclidean distance and the cosine distance [53]. Focused on IoT and firmware, the dataset consists of binaries compiled for three different 32-bit architectures (`Intel x86`, `ARM`, and `MIPS`) and using three compilers (two versions of `gcc` and `Clang`).

BinShape [54] is a system for identifying standard library functions in binaries using robust signatures and efficient matching techniques. It considers features (such as function call graphs) that are less affected by optimization settings. The evaluation uses C/C++ binaries compiled with `gcc` and `MSVC` compilers on `Intel x86` and `x86-64` architectures, linked with their respective standard C/C++ libraries.

Finally, it is worth mentioning Asm2Vec [55], DeepBinDiff [56], and PalmTree [57], which are state-of-the-art approaches that leverage machine learning and neural networks to generate sophisticated code representations, improving tasks such as binary code similarity and binary diffing even in the presence of code obfuscation and optimization transformations. In contrast, our primary goal is to provide insights into the build toolchain and compilation processes, which are essential for comprehensive software analysis and understanding of binary provenance. Consequently, our system addresses a different aspect of binary analysis, so direct comparisons with these approaches are not entirely applicable.

Comparison with our work. The works on compiler provenance and authorship attribution are complementary to ours and can help MANTILLA improve its accuracy in identifying the compiler or performing authorship attribution. As for the identification of library function, none of the works (but [44]) explicitly mention the runtime library that is used. In addition, they specifically focus on detecting function libraries other than runtime libraries, even filtering them out under the assumption that *they are from the compiler*.

We emphasize that our work does not focus on binary code similarity, but rather on recognizing the runtime library with which the binary under analysis has been compiled. Identifying runtime libraries is of interest to gaining a comprehensive understanding of the entire build toolchain used in a binary. This knowledge helps software analysts identify the specific environment and tools used during binary creation, which can be instrumental in reverse engineering, vulnerability assessment, and threat attribution. Additionally, accurate identification of runtime improves the ability to track potential security flaws due to unpatched vulnerabilities.

3. Mantilla: Threat model and system description

This section introduces the main components of MANTILLA. First, we briefly describe the threat model of this system. We then provide an overview of the system workflow, and finally describe each step in our pipeline. A high-level overview of our system is shown in Fig. 1.

3.1. Threat model

Our system aims to identify the runtime library used in statically linked binaries via a machine-learning approach based on features extracted from static analysis. This identification is useful to know if a statically linked stripped binary contains an outdated vulnerable library, as well as to help malware analyst focus their analysis on functions inherent to the malware's functionality. In this sense, adversaries may intentionally apply several actions that can undermine our system's effectiveness, which we detail below.

Static analysis evasion techniques. Adversaries may use various techniques to obfuscate the binary code or alter the structure of the binary to evade detection by static analysis tools (e.g., code obfuscation, packing, or adding junk code, among others) [58]. MANTILLA can be adapted to handle these evasion techniques, such as through robust feature extraction methods that focus on intrinsic properties of the binary rather than the obfuscated elements.

Adversarial machine learning attacks. MANTILLA uses KNN, a machine learning classification model. Therefore, adversaries can introduce adversarial samples designed to fool the underlying classifier (i.e., they can create adversarial inputs that lead to incorrect predictions) [59,60]. In this regard, strategies such as model validation, periodic updates, and adversarial training can improve the robustness of the machine learning model.

Incompleteness or inaccuracy of extracted features. MANTILLA relies on `radare2` [11] to analyze binary inputs and extract features. If the feature extraction process misses critical features or extracts incorrect data, it can lead to incorrect classification results. To address this, verification processes can be implemented to ensure accuracy in feature extraction. Similarly, using multiple binary analysis tools to cross-check extracted features can also help mitigate this issue.

Model drift and outdated training data. Emerging malware that use new techniques may be underrepresented in the dataset used to train our system. Therefore, the effectiveness of the machine learning model used in MANTILLA may degrade if the training data becomes outdated [61]. Strategies such as a regular model update cycle with new and diverse training data, as well as mechanisms to detect when model performance begins to degrade, can help minimize the impact of this issue.

Limited scope of runtime library identification. MANTILLA focuses on a specific set of runtime libraries and may not be able to identify newer (or less common) libraries. Therefore, adversaries may use new or custom runtime libraries in their malware to avoid detection. To address this, the supported runtime libraries can be expanded. Similarly, an update mechanism can also be incorporated into our system to add new libraries as they are identified in the wild.

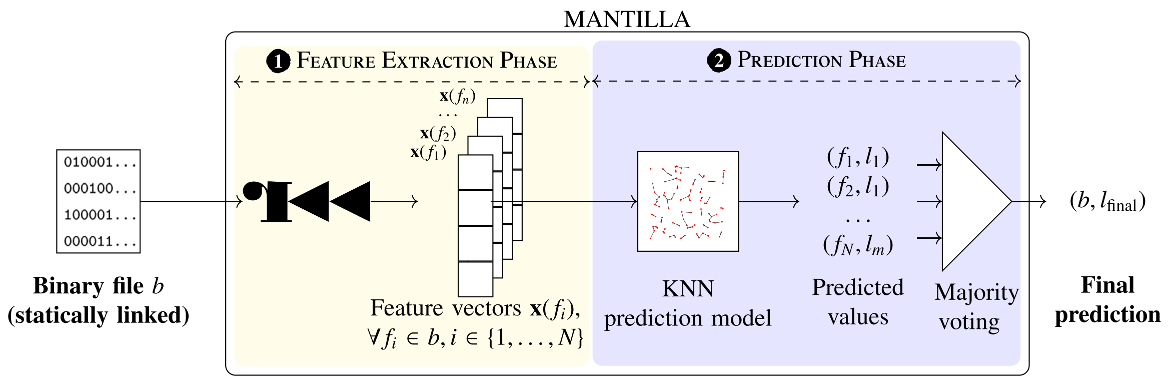


Fig. 1. High-level system overview of MANTILLA.

3.2. System overview

MANTILLA performs a two-phase workflow: the FEATURE EXTRACTION PHASE (❶ in Fig. 1), which extracts a set of features that are agnostic to the underlying architecture through static binary analysis [5]. All these features are obtained from the structural and semantic analysis of the binary [19]; and the PREDICTION PHASE (❷ in Fig. 1), which predicts the runtime library using a supervised learning algorithm and a majority voting. Both phases are described in more detail below.

3.3. Feature extraction phase

Let b a statically linked binary file composed of N functions. We use the `radare2` framework [11] (version 5.8.3 29989 git.5.8.2-187-g22d71f931a commit) to recognize the functions $f_i, i \in \{1,\dots,N\}$, of b and extract its features through static binary analysis. `radare2` is a widely used open-source reverse engineering framework that supports multiple CPU architectures and allows us to easily analyze binary files and extract features from their internal structure through the Python `r2pipe` library. Since we want our system to work with binaries compiled for different CPU architectures, we focus on features that are present on all architectures and that are not unique to the underlying architecture (i.e., features such as assembly code bytes or mnemonics are not considered). In particular, the extracted features are:

- The **cyclomatic complexity** metric, $CC(f_i)$, used in software engineering, quantitatively calculates the complexity of a function at the logical level. It is calculated based on the number of linearly independent paths through the function's control flow graph.
- The **size (in bytes) of the function**, $S(f_i)$. This feature gives insight into the footprint of the function within the binary.
- The **reserved stack space**, or $SS(f_i)$, which considers the amount of stack memory reserved by the function in its prologue, specifically for storing local variables and temporary data.
- The **number of basic blocks**, $BB(f_i)$, that the function contains. A *basic block* is a straight-line sequence of instructions with no jumps or branches except at the end (e.g., `call`, `jmp`, or `ret` in Intel assembly code), and no entry points except at the beginning.
- The **number of edges**, $E(f_i)$, that connect the existing basic blocks in the function. This represents the number of execution paths that the function may take. A function with more edges has a greater number of potential execution paths, indicating more dynamic or conditional behavior within the function.
- The total **number of individual instructions** in the function, $I(f_i)$.This feature provides a direct measure of the complexity of the function at the code level.
- The **number of arguments the function takes**, $A(f_i)$.
- The **computational cost** of the function, $C(f_i)$, which is calculated by `radare2` based on the instructions of the function. This feature provides insight into the complexity and processing demands of the function.
- The **number of extended basic blocks**, $EBB(f_i)$, within the function. An extended basic block is defined a group of basic blocks where only the first block can have multiple predecessors, while the others have only one predecessor, providing important structural information about the control flow of the function.
- A boolean value to indicate **whether the function has an explicit return or not**, $\text{noret}(f_i)$, which is useful for identifying functions that do not terminate in the standard way, providing additional granularity to the analysis.
- The total **number of local variables declared within the function**, $L(f_i)$.
- The **entropy** of the bytes that make up the function, or $H(f_i)$, that make up the function. A higher entropy value typically indicates a more complex or obfuscated function, while a lower entropy might suggest simpler, more repetitive patterns in the instructions.
- The **number of calls to other functions**. This feature captures how frequently the function invokes other functions. We split into two parts: the total number of function calls made within the function ($C_{\text{total}}(f_i)$, which includes multiple calls to the same function, and the number of unique functions called by the function ($C_{\text{unique}}(f_i)$).

As a result, we obtain a feature vector $\mathbf{x}(f_i) = [CC(f_i), S(f_i), SS(f_i), BB(f_i), E(f_i), I(f_i), A(f_i), C(f_i), EBB(f_i), \text{noret}(f_i), L(f_i), H(f_i), C_{\text{total}}(f_i), C_{\text{unique}}(f_i)]$ for each function $f_i \in b$ that will feed our runtime library identification model in the next phase. Note that none of these features depend on the symbols of b, and therefore MANTILLA can identify the runtime library regardless of whether b has symbols or not. We will show this in Section 4.3.

3.4. Prediction phase

As a model to identify the runtime library used in the linking process, we use the KNN [12] supervised learning algorithm. For a new data point d, this algorithm looks for the number of examples (K) closest to d and classifies it according to the most frequent label.

In our case, the KNN model (as shown in Fig. 1) generates a predicted runtime library $l_j, j \in \{1,\dots,m\}$, for each function f_i in a binary b based on its extracted features $\mathbf{x}(f_i)$. These individual predictions are fed into a majority voting system, which aggregates the predictions for all functions f_i in the binary b. The system then selects the most frequent label l_{final} as the final prediction for the runtime library of the entire binary.

We use KNN instead of other machine learning algorithms because the distance metric between clusters is inherent to the chosen algorithm, and therefore, we can discard from the prediction those functions whose distance to the nearest cluster is very large. In any case, our system is extensible to the use of other clustering models (such as DBScan or K-means). The evaluation of different clustering models is not within the scope of this work. Next, we describe the creation of the ground truth dataset used to train the KNN model and the evaluation metrics we use to measure its effectiveness.

3.4.1. Generation of ground truth dataset

Specifically, we focus on the C programming language due to its popularity and adoption by malware authors [62,63]. To create the dataset, we build toolchains for different CPU architectures (in particular, `MIPSeb`, `ARMel`, `Intel x86`, and `Intel x86-64`) and with different runtime libraries (in particular, `uClibc` [14], `glibc` [15], and `musl` [16]) using `buildroot` [64] and the `gcc` version 10.2.1 20210110 as compiler.

We use the collection of different algorithms implemented in C that are publicly available in the "TheAlgorithm" repository [65]. We compile each of these source codes with static linking and with different optimization options (specifically, `O0`, `O1`, `O2`, `O3`, and `Os`), using the toolchains described above. After the build and cross-compile process, our dataset consists of 13,800 statically linked unstripped binaries. To encourage further research, we have released all our dataset [18]. Finally, to assist in the feature extraction phase, we perform a preprocessing phase to remove those functions written by the programmer or from the standard C library. For each binary, this preprocessing phase works as follows:

(i) First, we get all the functions that are used or added by the programmer and that appear in the source code. To do this, we rely on `cflow` [66] (version 1.6) to get the name of all the functions that are used in each source code file, including all functions beginning with an underscore character;
(ii) Second, we disassemble the binary and extract the name of all its functions. To do this, we rely on their symbols and discard those functions that appear in the source code and that were found by `cflow`. We also removed functions that appear in the GNU C function list [67,68] and in the source code of `uClibc`, `glibc`, and `musl` to reduce the number of candidate functions. In this sense, only external and static functions remain, as well as aliases;
(iii) Finally, we extract the features of each of the candidate functions from the binary using `radare2` [11] as described in Section 3.3.

Note that at this stage we use the source code of the binary and the binary itself as inputs to our system. As output, we get the features of each function that was not discarded in the preprocessing phase. Each function in b is also labeled with the architecture, the runtime library, and the compiler used to create b. This dataset will be used to train the KNN prediction model of MANTILLA, as discussed later.

3.4.2. Evaluation metrics

As model evaluation metrics, we use: $Precision = \frac{TP}{TP+FP}$, where TP is the number of binaries whose runtime library is correctly identified (true positive) and FP is the number of binaries whose runtime library is falsely identified (false positive); $Recall = \frac{TP}{TP+FN}$, where FN is the number of binaries whose runtime library is wrongly identified (false negative); $F1-Score\ (F1-S) = 2\frac{Precision \cdot Recall}{Precision + Recall}$, which combines precision and recall into a single metric, providing a measure of balance between both metrics; and $Accuracy\ (Acc) = \frac{TP+TN}{TP+TN+FN+FP}$, where TN is the number of binaries whose runtime library is correctly misidentified (true negative) and measures the proportion of all samples that the model classified correctly.

4. Evaluation

This section evaluates the effectiveness of MANTILLA. We carry out different experiments to check if our system allows us to identify the runtime libraries used in the linking process. For all experiments in this work, we perform 5-fold cross-validation on each test (i.e., we split the dataset into five equally sized parts and perform five separate training procedures). The experiments were run with Python3 and the `Sklearn` library on a Debian 11 on top of a machine with an Intel i7-9700K 3.60 GHz CPU and 32 GB of RAM.

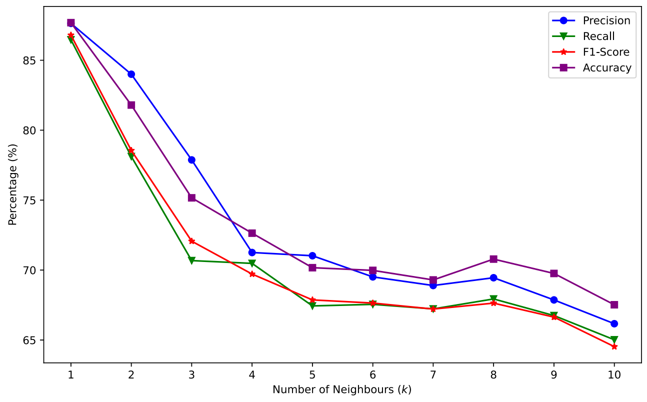


Fig. 2. Evaluation metrics for $k = \{1, \dots, 10\}$ in the KNN model and using the Euclidean distance.

We conducted five experiments. First, we validate the model by evaluating the accuracy of MANTILLA towards the whole dataset. We then measure the importance of the features we selected (see Section 3.3). Next, we evaluate its accuracy specifically on stripped binaries. We then test whether our system is able to distinguish the architecture for which a binary file is compiled, simply from the extracted features. Finally, we test whether it can recognize for a given binary file the compiler used to compile it, as well as its runtime library.

4.1. Validation

In this experiment, we evaluate the accuracy of MANTILLA towards the whole dataset. We first perform a sensitive analysis to find the best parameter settings of the KNN model (i.e., the number of k-nearest neighbors and the metric used to calculate the distance). As the number of neighbors, we consider $k = \{1, \dots, 10\}$. As distance metrics, we consider Euclidean (the default in the KNN prediction model of the software library we use), Manhattan, and Minkowski distances. All results are very similar, with no significant differences between them. Fig. 2 depicts the results of this evaluation for the default (i.e., Euclidean) metric, before applying the majority voting rule for the final prediction. We see that the model performance degrades as the neighbor size increases. This makes sense since KNN is using the k nearest neighbors to predict, without considering whether the distance between them is small or not.

The confusion matrix in Fig. 3 details the percentage of correct and incorrect labels for the best case ($k = 1$), giving a more accurate picture of the classification performance. The confusion matrix reveals that the classification model achieves an overall accuracy of 86.96%, indicating a good ability to differentiate binaries compiled using different architectures, compilers, and runtime libraries. The model performs particularly well with `mips_glibc_gcc` and mips_uclibc_gcc, achieving correct classification rates of 97% and 90%, respectively. However, some confusion is observed among binaries compiled for the same architecture but with different runtime libraries, especially within the `Arm` and `Intel x86` architectures. For instance, 12% of `arm_glibc_gcc` binaries are misclassified as `arm_musl_gcc`, and 20% of `x86_64_glibc_gcc` binaries are misclassified as `x86_64_uclibc_gcc`. These misclassifications suggest challenges in distinguishing between binaries compiled on similar architectures but different runtime libraries, likely due to similarities in their compiled structures. Despite these challenges, the model exhibits

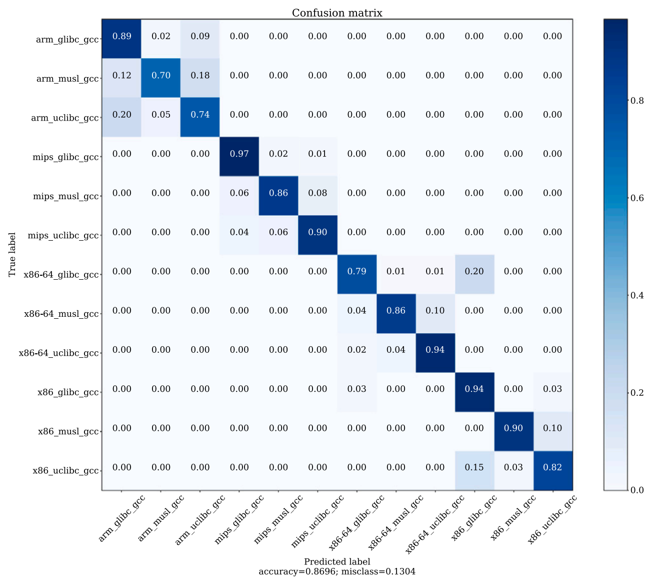


Fig. 3. Confusion matrix for the $K = 1$ and Euclidean distance settings without majority voting towards the whole dataset.

a generally strong performance, with only 13.04% of instances misclassified, providing a solid foundation for runtime library and architecture identification.

Recall that in these results we are not applying the majority voting rule for the final prediction (see Section 3.4). When applied, we obtain a hit rate of 100%. In this case, MANTILLA considers the rest of the functions of a binary before giving a prediction, instead of only a single function, thus giving better results.

4.2. Measuring feature importance

In this experiment we measured the importance of each feature in identifying runtime libraries. Since the KNN algorithm has no inherent function to measure feature importance, we use the *permutation importance technique* [69] to measure how much the model performance is affected when the values of a feature are randomly permuted, while keeping the rest of the features at their original values. This removes the relationship that such a feature has with the target variable, allowing us to evaluate the model with the permuted feature and calculate the difference with the original model. This difference indicates the importance of that feature to the model performance.

Table 1 shows the importance of each of the features used by MANTILLA. The feature that contributes the most is the size of the function, $S(f_i)$. According to the results, when this feature is randomly permuted, the model performance of the model drops by around 0.66.

4.3. Accuracy of MANTILLA on stripped binaries

For this experiment, we get rid of the symbols of all the compiled binaries from our ground truth dataset (described in Section 3.4), extracting the features of all the functions without performing any type of filtering. Therefore, for each binary file, we have its stripped and unstripped version.

We use the same cross-validation strategy, considering 80% of the binaries as training and 20% as testing. The ground truth dataset (unstripped binaries) is used to train the system, while all the functions of the stripped binaries are used for the test phase. Note that in this case, each binary contains standard C library functions, programmer functions, and any other library function. Therefore, it is necessary to filter out the functions unrelated to the runtime library because the

Table 1
Importance of the features used in MANTILLA (sorted by weight).

Weight	σ	Feature
0.6581	0.0030	$S(f_i)$ (size)
0.5621	0.0034	$C(f_i)$ (cost)
0.4239	0.0025	$SS(f_i)$ (stackframe)
0.3796	0.0031	$I(f_i)$ (ninst)
0.1349	0.0017	$E(f_i)$ (edges)
0.0597	0.0004	$H(f_i)$ (entropy)
0.0356	0.0009	$BB(f_i)$ (nbbs)
0.0349	0.0020	$A(f_i)$ (nargs)
0.0227	0.0011	$CC(f_i)$ (cc)
0.0191	0.0021	$L(f_i)$ (nlocals)
0.0110	0.0013	$C_{\text{total}}(f_i)$ (outdegree)
0.0022	0.0003	$EBB(f_i)$ (ebbs)
0.0005	0.0013	$C_{\text{unique}}(f_i)$ (unique outdegree)
0.0004	0.0003	$noret(f_i)$ (noreturn)

KNN prediction model returns the neighbors with the closest distance, regardless of how far they are from those neighbors. To do so, we work directly with the KNN distances instead of the prediction labels.

In particular, we obtain the distances to the K nearest neighbors, establishing a threshold d for the maximum distance and using the Euclidean distance metric.[1] Fig. 4 depicts the results obtained for different parameter settings. We have tested $k = \{1, \dots, 5\}$ and $d \in \{1, \dots, 7\}$. The results show that the system performs better with more neighbors and a lower distance threshold. According to the results, the performance is reduced as the distance threshold is relaxed. When we increase the distance threshold, MANTILLA starts to recognize functions not related to the runtime library as related, which degrades overall system performance. Based on these results, we conclude that a configuration with $K > 1$ and a low distance metric is preferable. As we see, it achieves a hit rate of 100% due to the majority voting rule, being the best performance for $k = 5$, which maintains a better hit rate even with a more relaxed threshold d.

4.4. Determining the architecture from the extracted features

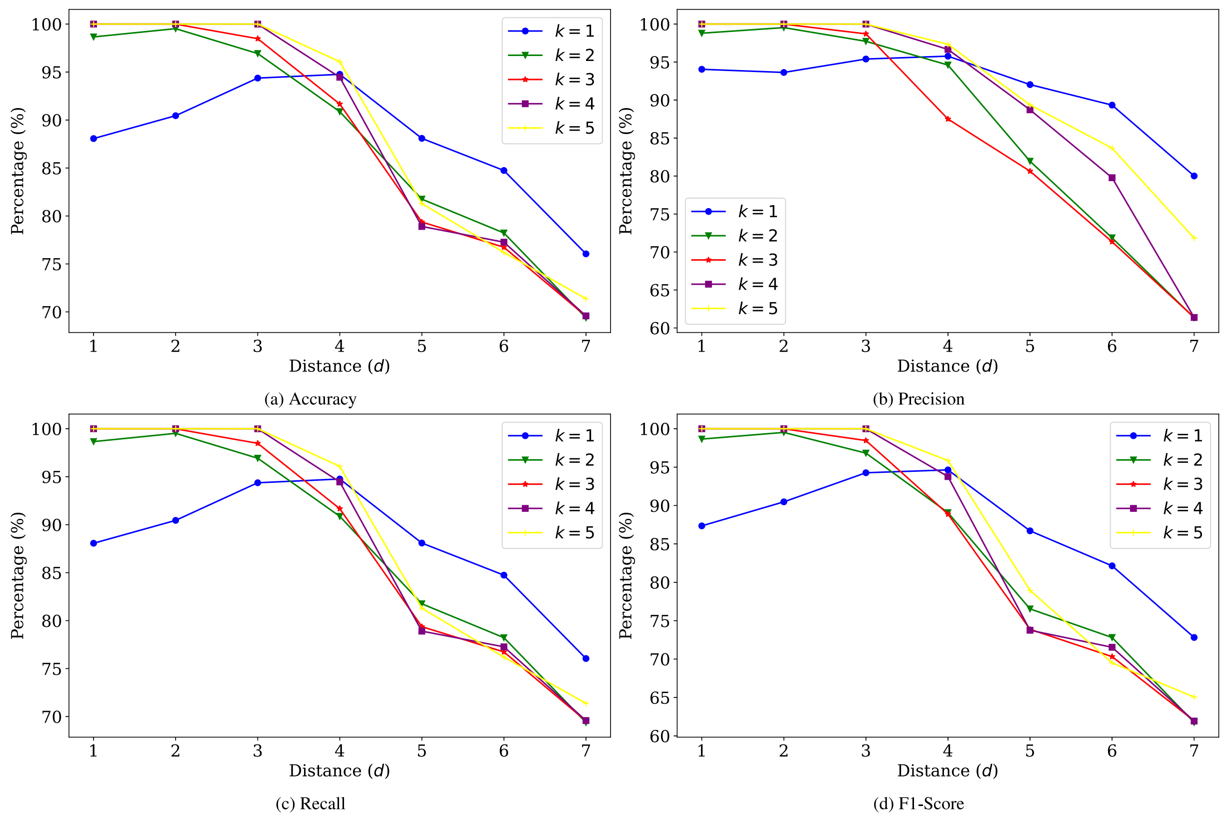
Incorrect identification of the architecture for which a binary file is compiled can lead to incorrect analysis [70]. Therefore, correct architecture identification can help analyze binary files and then lead to faster discovery of more vulnerabilities. In this section, we further explore if MANTILLA can determine the underlying architecture for which a given binary file is compiled simply by using the extracted features.

Fig. 3 already shows that the mismatches on predicted values are mostly within the same architecture. That is, before applying the majority voting rule, MANTILLA can identify the architecture with very high accuracy. Recall that none of the features extracted by our system are specific to the underlying architecture (see Section 3.3). When majority voting is applied, the first time an incorrect classification starts to appear in a preferred configuration ($K > 1$ and a low d) is with $k = 3, d = 3$. In this case, we obtain a confusion matrix with all the diagonal elements (except the `x86-64_uclibc_gcc` label) have a value of 1 (we omit this figure for the sake of space). Specifically, some data from `x86-64_uclibc_gcc` is misclassified into `x86-64_glibc_gcc` (in particular, 19%), that is, the same architecture and compiler, but a different runtime library. Based on these results, we conclude that MANTILLA can determine the architecture from the extracted features with very high accuracy.

4.5. Compiler provenance

Finally, we further explore whether MANTILLA can be used for compiler provenance, as has been studied before in many works

[1] As before, we have also evaluated the Manhattan and Minkowski distances, but the results obtained were very similar.



(a) Accuracy (b) Precision (c) Recall (d) F1-Score

Fig. 4. Evaluation metrics for the KNN prediction model considering $K = \{1, \dots, 5\}$ and distance thresholds $d \in \{1, \dots, 7\}$, on stripped binaries.

Table 2
Average results of evaluation metrics for the compiler provenance experiment.

Predicted label	Precision	Recall	F1-Score
x86-64_glibc_clang	0.56	0.47	0.51
x86-64_glibc_gcc	0.54	0.62	0.58
x86_glibc_clang	0.54	0.47	0.50
x86_glibc_gcc	0.53	0.60	0.56

(see Section 2.1). Since our dataset only contains one compiler (in particular, `gcc`; see Section 3.4), we add `Clang` version 11.0.1-2 to our build toolchain. In particular, `Clang` is only used for `Intel x86` and `Intel x86-64` architectures with `glibc`. As a result, we get another 2300 binaries.

To avoid overfitting during the training phase, we removed duplicate functions. That is, we eliminate those functions that have the same feature vector and the same label after the feature extraction phase. This allows, in addition to reducing the computational and memory load of the model, to prevent the distance to different neighbors from being the same because there could be several functions in the training phase that are in fact the same function.

Table 2 shows the average results of this experiment. As shown, the compiler prediction is not good enough. Note that in this case, the extracted functions belong to the same runtime library (i.e., `glibc`), and both compilers are using it. Therefore, the distance to the neighbors is very similar or even identical, which causes the prediction to fail. In particular, the results in Table 2 shows that the percentage of evaluation metrics is, on average, close to 50%. This is mainly because if the KNN prediction model finds neighbors with the same distance but different labels, the results will mainly depend on the order used in the training data. Therefore, based on these results, we conclude that MANTILLA cannot determine the compiler used to compile a given binary file.

5. Case study

In this section we present the use of MANTILLA with two real case studies. First, we use MANTILLA to recognize the runtime library in applications from the `binutils` collection. Then, we use MANTILLA to identify the runtime library in IoT malware.

5.1. Runtime libraries in `binutils` applications

The GNU `binutils` [71] collection is a set of tools for creating, manipulating, and analyzing binary files. Considering the source code of this collection (version 2.38), we created the toolset for the different architectures supported by our system (i.e., `MIPSeb`, `ARMel`, `Intel x86`, and `Intel x86-64`) using different runtime libraries (i.e., `uClibc`, `glibc`, and `musl`) and different optimization modes (in particular, `O0`, `O1`, `O2`, `O3`, and `Os`). All the binaries were statically linked and any debug symbols were stripped (using `strip`). In total, the dataset for this case study consists of 960 binaries evenly distributed for each of the architectures supported by MANTILLA.

We use the model created in Section 4 and feed it with this data set, setting $k = 5$, $d = 1$ as configuration parameters and using the Euclidean distance as the distance function. Note that this dataset has not been used in the evaluation phase explained in Section 4 (that is, it is *unknown* for MANTILLA).

Fig. 5 shows the confusion matrix of this experiment. The model performs exceptionally well on `MIPS` and `Inte x86` architectures, achieving 100% accuracy on all evaluated runtime libraries (`glibc`, `musl`, and `uclibc`) for these architectures. However, there is notable confusion within the `Arm` architecture, where 41% of `arm_musl_gcc` binaries are misclassified as `arm_glibc_gcc`, and 19% of `arm_uclibc_gcc` binaries are also misclassified as `arm_glibc_gcc`. These results suggest that the model struggles to differentiate binaries compiled with different runtime libraries for the Arm architecture, likely due to feature similarities between these configurations. Despite this, the low misclassification rate of 5.56% indicates that the model performs strongly

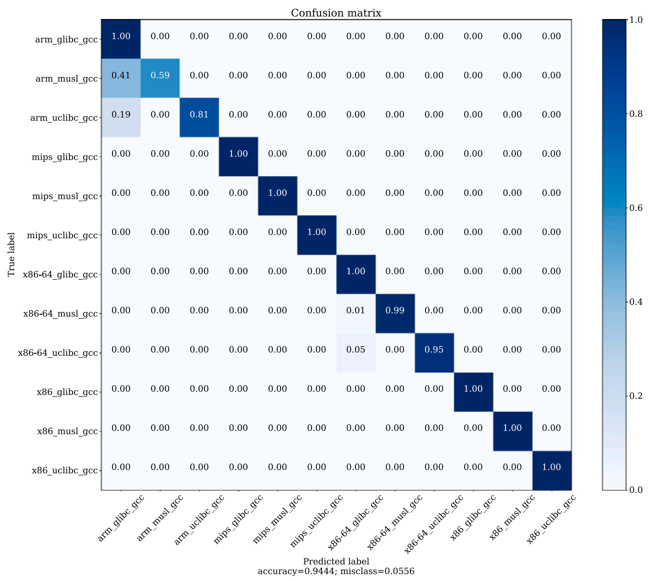


Fig. 5. Confusion matrix for the $k = 5$, $d = 1$, and Euclidean distance as configuration settings for detection of runtime libraries in the `binutils` programs.

overall, with room for improvement particularly in distinguishing Arm binaries compiled with different runtime libraries.

Let us note that, as we pointed out in Section 4.3, the results improve by decreasing the distance threshold. We repeated this experiment by reducing the distance threshold to $d = 0.5$ and $d = 0.1$, and the accuracy obtained was 96% and 99.2%, respectively. On the other hand, observing the identification results of the architecture for which the binary was compiled, we see that no misclassification between architectures has occurred.

5.2. *Runtime libraries of IoT malware*

This section presents a real case study where we use MANTILLA for runtime library identification. We focus on malware that targets IoT devices as variants exist for different architectures and are linked with different runtime libraries. To build the dataset, we collect samples from the VirusShare repository [72]. The original dataset consists of 56,502 samples of Linux-based malware targeting different hardware architectures, where 50,286 samples are linked statically and 6216 dynamically. In total, around 54% of the samples are stripped. To build our final dataset and be able to verify the hit rate obtained by MANTILLA we filter the dataset as follows. First, we remove from the dataset those samples that do not belong to the architectures currently supported by our system (see Section 3). We then remove those samples that are not statically compiled or stripped. Keep in mind that although MANTILLA does not rely on symbols, we need them to simply check its accuracy. Finally, we discard those samples that we identify as not being developed with the C programming language (such as C++ and Golang). To do this, we rely on the presence of certain strings (e.g., `golang`) or mangled names in symbols.

In total, our malware dataset consists of 9553 samples distributed almost equally between the `ARMel` and `Intel x86` architectures, with the remainder being `Intel x86-64`. No samples were found for the `MIPSeb` architecture that met the selection criteria. To classify the samples into malware families, we collect the VirusTotal reports for each sample and use AVClass [73]. Table 3 summarizes the distribution of the top 5 malware families, as well as the number of samples per runtime library and per architecture. As shown, most of the samples are distributed between the `Mirai` (27%) and `Gafgyt` (67%) families, and about 96% of the total samples were compiled using `uClibc` as the runtime library.

We fed MANTILLA with this dataset of malware samples, using $k = 5, d = 1$, and Euclidean distance as configuration settings to reduce any potential bias in the order of the training data (as discussed later). Our system provides the runtime library (and, indirectly, architecture information) for 9501 samples, that is, only 99.46% of the 9553 malware samples have functions with a distance $d \leq 1$.

Table 4 summarizes the results obtained for both the architecture and the runtime library by malware family. Specifically, the table shows the number of class samples and the percentage of successes and failures, for each library and architecture. In percentage, the total success rate for the architecture is around 98.6%, while for the runtime library it is around 95.5%, a similar result to the previous case study.

Looking at the individual hit and miss rates in identifying the runtime library for each family, we see that the top misclassifications occur in specific families (such as `Xorddos`, `Tsunami`, or `Others`). However, these samples make up a small percentage of the total sample set that appears in the dataset. As for the architecture identification, we see that misclassifications occur between the `Intel x86` and `Intel x86-64` architectures. A similar result is discussed in Section 4.4. This may be caused because the `x86-64` architecture is a 64-bit version of the `x86` architecture. Thus, they have certain similarities.

6. Threats to validity

Although MANTILLA identifies the runtime libraries on a never-before-seen IoT malware dataset with high accuracy, it still has some limitations. In this section, we discuss the validity of our work according to construct, internal, and external validity [74].

Construct validity. We have conducted controlled experiments that allowed us to adjust our system and measure evaluation metrics accordingly. Therefore, no issue should arise from our experimental study.

Internal validity. MANTILLA relies on third-party binary analysis tools (particularly `radare2` [11]) to obtain the functions that make up a binary and extract their features. Thus, the quality of these features largely depends on the binary analysis tool used. Therefore, the accuracy of our system may be affected by assumptions made by these types of tools during binary analysis [75,76] (for instance, for the detection of instruction boundary, function boundary, or the function signatures [77]). Keeping this in mind, our system has been designed in such a way that the component of the feature extraction phase (see Section 3.3) is easily interchangeable. A comparative study using different binary analysis tools (similar to [76], but not focused on ARM architecture) for this phase deserves further investigation and we plan to do so as immediate future work.

On the other hand, MANTILLA uses KNN as a model for supervised learning. Recall that KNN computes the distances to all training data and returns the K nearest neighbors (that is, those K neighbors with the shortest distance to the sample to predict). However, when there are more than K neighbors with the same distance, the return values are highly dependent on the order used in the training dataset. To overcome this limitation, we have tested it on a previously unseen dataset with K-fold cross-validation, thus reducing any potential bias in the order of the training data. Furthermore, KNN is sensitive to high-dimensional data, commonly referred to as the *curse of dimensionality* [12]. As the number of features increases, the distance between data points becomes less informative, which can degrade the performance of the model. To overcome this, we limit the number of extracted features to those that are present in all architectures. Finally, MANTILLA is also sensitive to the distance chosen as a threshold to consider the prediction in the voting phase. To mitigate this, we performed different tests to assess the performance of the system as a function of the chosen distance.

Table 3
Distribution of the malware dataset by families, architecture, and runtime library.

Family	#	Runtime library			Architecture		
		`uClibc`	`glibc`	`musl`	`ARMel`	`Intel x86`	`Intel x86-64`
`Gafgyt`	6,409	6,398 (99.83%)	11 (0.17%)	0 (0.00%)	1,907 (29.76%)	2,922 (45.59%)	1,580 (24.65%)
`Mirai`	2,558	2,490 (97.34%)	68 (2.66%)	0 (0.00%)	1,964 (76.78%)	444 (17.36%)	150 (5.86%)
`Tsunami`	285	271 (95.09%)	14 (4.91%)	0 (0.00%)	77 (27.02%)	125 (43.86%)	83 (29.12%)
`Xorddos`	79	0 (0.00%)	79 (100.00%)	0 (0.00%)	0 (0.00%)	79 (100.00%)	0 (0.00%)
`Ddostf`	46	0 (0.00%)	46 (100.00%)	0 (0.00%)	4 (1.40%)	42 (14.74%)	0 (0.00%)
`Others`	176	4 (1.40%)	171 (60.00%)	1 (0.35%)	6 (2.11%)	153 (53.68%)	17 (5.96%)
	9,553	**9,163** (95.92%)	**389** (4.07%)	**1** (0.01%)	**3,958** (41.43%)	**3,765** (39.41%)	**1,830** (19.16%)

Table 4
Percentage of successes and failures in identifying (a) runtime libraries and (b) architecture for the top 5 malware families present in the dataset.

Family	`uClibc`			`glibc`			`musl`		
	#	✓	✗	#	✓	✗	#	✓	✗
`Gafgyt`	6,398	97.47%	2.53%	11	90.91%	9.09%	0	–	–
`Mirai`	2,490	99.88%	0.12%	60	60.00%	40.00%	0	–	–
`Tsunami`	271	80.44%	19.56%	14	42.86%	57.14%	0	–	–
`Xorddos`	0	–	–	79	0.00%	100.00%	0	–	–
`Ddostf`	0	–	–	46	100.00%	0.00%	0	–	–
`Others`	4	25.00%	75.00%	127	29.92%	70.08%	1	0.00%	100.00%

(a) Identification of runtime libraries

Family	`ARMel`			`Intel x86`			`Intel x86-64`		
	#	✓	✗	#	✓	✗	#	✓	✗
`Gafgyt`	1,907	100.00%	0.00%	2,922	96.89%	3.11%	1,580	100.00%	0.00%
`Mirai`	1,960	100.00%	0.00%	440	98.86%	1.14%	150	100.00%	0.00%
`Tsunami`	77	100.00%	0.00%	125	79.20%	20.80%	83	100.00%	0.00%
`Xorddos`	0	–	–	79	100.00%	0.00%	0	–	–
`Ddostf`	4	100.00%	0.00%	42	100.00%	0.00%	0	–	–
`Others`	6	100.00%	0.00%	109	95.41%	4.59%	17	88.24%	11.76%

(b) Identification of architecture

External validity. MANTILLA is specifically designed to work with binaries developed in the C programming language. Therefore, prediction errors are very likely to occur when identifying the runtime library in binaries developed in other programming languages. On the other hand, our work focuses on the identification of runtime libraries in statically linked binaries, compiled on a GNU/Linux machine. The accuracy of MANTILLA may be affected in obfuscated or packaged binaries, leading to inappropriate results as runtime functions may be affected due to obfuscation or packaging itself. However, the use of these types of techniques is not very common in malware that targets IoT devices, and if it exists, the authors mainly use packers that are widely known in the community (i.e., UPX) and whose unpacking is trivial [3,78,79].

Fortunately, the methodology used here is extensible to other operating systems (e.g., Microsoft Windows). The idea behind our methodology is to obtain the functions used in the source code to identify the linking functions used in statically linked binaries so that we can calculate the distance to those functions in stripped binaries (i.e., binaries whose symbols have been removed). This approach can also be used to identify the runtime linked library on other platforms. Identifying runtime libraries on other platforms requires more research and we plan to do so as future work.

Furthermore, in this work we have considered different hardware architectures, such as `MIPSeb`, `ARMel`, `x86`, and `x86-64`. However, other architectures commonly found on IoT malware samples are not currently covered here (e.g., `PowerPC`, `MC68000`, or `SPARC`) [79]. As mentioned, our methodology can be easily extensible by generating toolchains for other architectures and following the same process discussed above. Finally, this work focuses on the most popular runtime libraries on GNU/Linux-based systems. Other C runtime libraries for Linux (such as `bionic`, `dietlibc`, or `newlib`) are not considered here.

7. Conclusions and future work

In this paper, we have introduced a new system, dubbed MANTILLA, for identifying runtime libraries in statically linked binaries. Our system first analyzes the functions of a given binary and extracts a set of features that are independent of the underlying binary architecture. It then uses a KNN supervised learning model to obtain predicted values (for each function) and a majority voting rule to make the final prediction. We evaluated it using a K-fold cross-validation and different configuration parameters on a ground truth dataset generated for different runtime libraries (`uClibc`, `glibc`, and `musl`) and architectures (`ARMel`, `Intel x86`, and `Intel x86-64`). The results show that our system is accurate, obtaining better results when we use a relaxed distance threshold to the neighbors and higher values of K to make the final prediction. To encourage further research in this field, we have released it publicly, along with the dataset used.

Additionally, we conducted two case studies, one with applications from the `binutils` collection and the other with malware targeting IoT devices, achieving a success rate of 94.4% and 95.5% in runtime library prediction, respectively. Furthermore, MANTILLA achieved a success rate of 100% and 98.6% in predicting the architecture of the binaries, respectively. Although our system achieves good accuracy, more research is needed. As future work, our immediate goal is to explore other types of architectures and operating systems, as well as other runtime libraries. We also plan to give our system access via a web service, to facilitate its integration into other (third-party) analysis workflows.

CRediT authorship contribution statement

Javier Carrillo-Mondéjar: Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Ricardo J.**

Rodríguez: Writing – review & editing, Writing – original draft, Visualization, Supervision, Project administration, Methodology, Funding acquisition, Formal analysis, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This research was supported in part by TED2021-131115A-I00 (MIMFA), funded by MCIN/AEI/10.13039/501100011033, by the Recovery, Transformation and Resilience Plan funds, financed by the European Union (Next Generation), by the Spanish National Cybersecurity Institute (INCIBE) under *Proyecto Estratégico CIBERSEGURIDAD EINA UNIZAR*, and by the University, Industry and Innovation Department of the Aragonese Government under *Programa de Proyectos Estratégicos de Grupos de Investigación* (DisCo research group, ref. T21-23R).

Data availability

Data will be available in a repository.

References

[1] A. Aho, J. Ullman, R. Sethi, M. Lam, Compilers: Principles, Techniques, and Tools, Addison Wesley, 2006.

[2] C.S. Collberg, J.H. Hartman, S. Babu, S.K. Udupa, SLINKY: Static linking reloaded, in: Proceedings of the 2005 USENIX Annual Technical Conference, April 10-15, 2005, Anaheim, CA, USA, USENIX, 2005, pp. 309–322.

[3] E. Cozzi, M. Graziano, Y. Fratantonio, D. Balzarotti, Understanding linux malware, in: 2018 IEEE Symposium on Security and Privacy, SP, 2018, pp. 161–175.

[4] E. Cozzi, P.-A. Vervier, M. Dell'Amico, Y. Shen, L. Bilge, D. Balzarotti, The tangled genealogy of IoT malware, in: Annual Computer Security Applications Conference, ACSAC '20, Association for Computing Machinery, New York, NY, USA, 2020, pp. 1–16.

[5] K. Liu, H.B.K. Tan, X. Chen, Binary code analysis, Computer 46 (2013) 60–68.

[6] H. Welte, The gpl-violations.org project, 2004, Online; https://gpl-violations.org/. (Accessed 26 May 2023).

[7] M. Yong Wong, M. Landen, M. Antonakakis, D.M. Blough, E.M. Redmiles, M. Ahamad, An inside look into the practice of malware analysis, in: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, CCS '21, Association for Computing Machinery, New York, NY, USA, 2021, pp. 3053–3069.

[8] D. Andriesse, Practical Binary Analysis: Build Your Own Linux Tools for Binary Instrumentation, Analysis, and Disassembly, No Starch Press, 2018.

[9] S. Alrabaee, M. Debbabi, P. Shirani, L. Wang, A. Youssef, A. Rahimian, L. Nouh, D. Mouheb, H. Huang, A. Hanna, Binary analysis overview, in: Binary Code Fingerprinting for Cybersecurity: Application to Malicious Code Fingerprinting, Springer International Publishing, Cham, 2020, pp. 7–44.

[10] G. Balakrishnan, T. Reps, WYSINWYX: What you see is not what you execute, ACM Trans. Program. Lang. Syst. 32 (2010).

[11] Radare2 Team, Radare2 book, 2021, Online; https://book.rada.re/. (Accessed 11 October 2023).

[12] O. Kramer, K-nearest neighbors, in: Dimensionality Reduction with Unsupervised Nearest Neighbors, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 13–23.

[13] S. Alrabaee, M. Debbabi, L. Wang, A survey of binary code fingerprinting approaches: Taxonomy, methodologies, and features, ACM Comput. Surv. 55 (2022).

[14] E. Andersen, Uclibc, 2012, Online; https://www.uclibc.org/. (Accessed 26 May 2024).

[15] glibc, The GNU C library (glibc), 2023, Online; https://www.gnu.org/software/libc/. (Accessed 24 May 2023).

[16] musl, musl libc, 2023, Online; https://musl.libc.org/. (Accessed 24 May 2023).

[17] J. Carrillo-Mondéjar, R.J. Rodríguez, MANTILLA: A system for runtime libraries identification in sTatIcally-linked linux binaries, 2023, Online; https://github.com/reverseame/MANTILLA. (Accessed 26 May 2023).

[18] J. Carrillo-Mondéjar, R.J. Rodríguez, Identifying runtime libraries in statically linked linux binaries [dataset], 2023, http://dx.doi.org/10.5281/zenodo.7991325, [Online]. (Accessed 31 May 2023).

[19] D. Kim, E. Kim, S.K. Cha, S. Son, Y. Kim, Revisiting binary code similarity analysis using interpretable feature engineering and lessons learned, IEEE Trans. Softw. Eng. 49 (2023) 1661–1682.

[20] N.E. Rosenblum, B.P. Miller, X. Zhu, Extracting compiler provenance from program binaries, in: Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE '10, ACM, New York, NY, USA, 2010, pp. 21–28.

[21] N. Rosenblum, B.P. Miller, X. Zhu, Recovering the toolchain provenance of binary code, in: Proceedings of the 2011 International Symposium on Software Testing and Analysis, ISSTA '11, Association for Computing Machinery, New York, NY, USA, 2011, pp. 100–110.

[22] A. Rahimian, P. Shirani, S. Alrbaee, L. Wang, M. Debbabi, BinComp: A stratified approach to compiler provenance attribution, Digit. Investig. 14 (2015) S146–S155, The Proceedings of the Fifteenth Annual DFRWS Conference.

[23] R. Real, J.M. Vargas, The probabilistic basis of Jaccard's index of similarity, Syst. Biol. 45 (1996) 380–385.

[24] S. Alrabaee, P. Shirani, L. Wang, M. Debbabi, FOSSIL: A resilient and efficient system for identifying foss functions in malware binaries, ACM Trans. Priv. Secur. 21 (2018).

[25] Y. Chen, Z. Shi, H. Li, W. Zhao, Y. Liu, Y. Qiao, HIMALIA: Recovering compiler optimization levels from binaries by deep learning, in: K. Arai, S. Kapoor, R. Bhatia (Eds.), Intelligent Systems and Applications, Springer International Publishing, Cham, 2019, pp. 35–47.

[26] S. Yang, Z. Shi, G. Zhang, M. Li, Y. Ma, L. Sun, Understand code style: Efficient CNN-based compiler optimization recognition system, in: ICC 2019 - 2019 IEEE International Conference on Communications, ICC, 2019, pp. 1–6.

[27] Z. Tian, Y. Huang, B. Xie, Y. Chen, L. Chen, D. Wu, Fine-grained compiler identification with sequence-oriented neural modeling, IEEE Access 9 (2021) 49160–49175.

[28] D. Pizzolotto, K. Inoue, Identifying compiler and optimization level in binary code from multiple architectures, IEEE Access 9 (2021) 163461–163475.

[29] Y. Ji, L. Cui, H.H. Huang, Vestige: Identifying binary code provenance for vulnerability detection, in: K. Sako, N.O. Tippenhauer (Eds.), Applied Cryptography and Network Security, Springer International Publishing, Cham, 2021, pp. 287–310.

[30] Y. Du, R. Court, K. Snow, F. Monrose, Automatic recovery of fine-grained compiler artifacts at the binary level, in: 2022 USENIX Annual Technical Conference, USENIX ATC 22, USENIX Association, Carlsbad, CA, 2022, pp. 853–868.

[31] X. He, S. Wang, Y. Xing, P. Feng, H. Wang, Q. Li, S. Chen, K. Sun, BinProv: Binary code provenance identification without disassembly, in: Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses, RAID '22, Association for Computing Machinery, New York, NY, USA, 2022, pp. 350–363.

[32] N. Rosenblum, X. Zhu, B.P. Miller, Who wrote this code? Identifying the authors of program binaries, in: V. Atluri, C. Diaz (Eds.), Computer Security – ESORICS 2011, Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 172–189.

[33] S. Alrabaee, N. Saleem, S. Preda, L. Wang, M. Debbabi, OBA2: An onion approach to binary code authorship attribution, Digit. Investig. 11 (2014) S94–S103, Proceedings of the First Annual DFRWS Europe.

[34] X. Meng, B.P. Miller, K.-S. Jun, Identifying multiple authors in a binary program, in: S.N. Foley, D. Gollmann, E. Snekkenes (Eds.), Computer Security – ESORICS 2017, Springer International Publishing, Cham, 2017, pp. 286–304.

[35] T. Jo, Machine Learning Foundations, first ed., Springer Cham, 2021.

[36] S. Alrabaee, P. Shirani, L. Wang, M. Debbabi, A. Hanna, On leveraging coding habits for effective binary authorship attribution, in: J. Lopez, J. Zhou, M. Soriano (Eds.), Computer Security – ESORICS 2018, Springer International Publishing, Cham, 2018, pp. 26–47.

[37] P.C. Mahalanobis, On the generalized distance in statistics, in: Proceedings of the National Institute of Science of India, 1936, pp. 49–55.

[38] R. Cilibrasi, P.M.B. Vitanyi, Clustering by compression, IEEE Trans. Inform. Theory 51 (2005) 1523–1545.

[39] A. Caliskan, F. Yamaguchi, E. Dauber, R.E. Harang, K. Rieck, R. Greenstadt, A. Narayanan, When coding style survives compilation: De-anonymizing programmers from executable binaries, in: 25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018, The Internet Society, 2018.

[40] S. Alrabaee, M. Debbabi, L. Wang, On the feasibility of binary authorship characterization, Digit. Investig. 28 (2019) S3–S11.

[41] I. Goodfellow, Y. Bengio, A. Courville, Deep learning, first ed., Adaptive Computation and Machine Learning series, The MIT Press, 2016.

[42] Hex-Rays, IDA F.L.I.R.T. Technology: In-depth, Online; https://hex-rays.com/products/ida/tech/flirt/in_depth/. (Accessed 11 May 2023).

[43] J. McMaster, Issues with FLIRT aware malware, 2011, Online; https://siliconpr0n.org/uv/issues_with_flirt_aware_malware.pdf. (Accessed 11 May 2023).

[44] E.R. Jacobson, N. Rosenblum, B.P. Miller, Labeling library functions in stripped binaries, in: Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools, PASTE '11, Association for Computing Machinery, New York, NY, USA, 2011, pp. 1–8.

[45] X. Meng, J.T. Gu, B. Williams, Dyninst project – examples, 2023, Online; https://github.com/dyninst/examples/tree/master/unstrip. (Accessed 11 May 2023).

[46] W. Jin, S. Chaki, C. Cohen, A. Gurfinkel, J. Havrilla, C. Hines, P. Narasimhan, Binary function clustering using semantic hashes, in: Proceedings of the 2012 11th International Conference on Machine Learning and Applications, vol. 1, 2012, pp. 386–391.
[47] J. Qiu, X. Su, P. Ma, Library functions identification in binary code by using graph isomorphism testings, in: 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering, SANER, 2015, pp. 261–270.
[48] J. Qiu, X. Su, P. Ma, Using reduced execution flow graph to identify library functions in binary code, IEEE Trans. Softw. Eng. 42 (2016) 187–202.
[49] S. Eschweiler, K. Yakdan, E. Gerhards-Padilla, discovRE: Efficient cross-architecture identification of bugs in binary code, in: 23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016, The Internet Society, 2016.
[50] T. Dullien, R. Rolles, Graph-based comparison of executable objects, in: Proceedings of the Symposium sur la Securite des Technologies de l'Information et des Communications, 2015.
[51] M. Bourquin, A. King, E. Robbins, BinSlayer: Accurate comparison of binary executables, in: Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop, PPREW '13, Association for Computing Machinery, New York, NY, USA, 2013.
[52] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, H. Yin, Scalable graph-based bug search for firmware images, in: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, CCS '16, Association for Computing Machinery, New York, NY, USA, 2016, pp. 480–491.
[53] G. Qian, S. Sural, Y. Gu, S. Pramanik, Similarity between Euclidean and cosine angle distance for nearest neighbor queries, in: Proceedings of the 2004 ACM Symposium on Applied Computing, SAC '04, Association for Computing Machinery, New York, NY, USA, 2004, pp. 1232–1237.
[54] P. Shirani, L. Wang, M. Debbabi, BinShape: Scalable and robust binary library function identification using function shape, in: M. Polychronakis, M. Meier (Eds.), Detection of Intrusions and Malware, and Vulnerability Assessment, Springer International Publishing, Cham, 2017, pp. 301–324.
[55] S.H.H. Ding, B.C.M. Fung, P. Charland, Asm2Vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization, in: 2019 IEEE Symposium on Security and Privacy, SP, 2019, pp. 472–489, http://dx.doi.org/10.1109/SP.2019.00003.
[56] Y. Duan, X. Li, J. Wang, H. Yin, et al., Deepbindiff: Learning program-wide code representations for binary diffing, 2020.
[57] X. Li, Y. Qu, H. Yin, PalmTree: Learning an assembly language model for instruction embedding, in: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, CCS '21, Association for Computing Machinery, New York, NY, USA, 2021, pp. 3236–3251, http://dx.doi.org/10.1145/3460120.3484587.
[58] A. Moser, C. Kruegel, E. Kirda, Limits of static analysis for malware detection, in: Proceedings of the 23rd Annual Computer Security Applications Conference, ACSAC, 2007, pp. 421–430.
[59] C. Sitawarin, D. Wagner, On the robustness of deep K-nearest neighbors, in: 2019 IEEE Security and Privacy Workshops, SPW, 2019, pp. 1–7.
[60] C. Sitawarin, D. Wagner, Minimum-norm adversarial examples on KNN and KNN based models, in: 2020 IEEE Security and Privacy Workshops, SPW, 2020, pp. 34–40.
[61] K. Nelson, G. Corbin, M. Anania, M. Kovacs, J. Tobias, M. Blowers, Evaluating model drift in machine learning algorithms, in: 2015 IEEE Symposium on Computational Intelligence for Security and Defense Applications, CISDA, 2015, pp. 1–8.
[62] A. Calleja, J. Tapiador, J. Caballero, The MalSource dataset: Quantifying complexity and code reuse in malware development, IEEE Trans. Inf. Forensics Secur. 14 (2019) 3175–3190.
[63] M. Lindorfer, A. Di Federico, F. Maggi, P.M. Comparetti, S. Zanero, Lines of malicious code: Insights into the malicious software industry, in: Proceedings of the 28th Annual Computer Security Applications Conference, ACSAC '12, Association for Computing Machinery, New York, NY, USA, 2012, pp. 349–358.
[64] buildroot, Buildroot - Making Embedded Linux Easy, 2023, Online; https://buildroot.org. (Accessed 24 May 2023).
[65] T. Algorithms, The algorithms - C, 2023, Online; https://github.com/TheAlgorithms/C/. (Accessed 12 May 2023).
[66] S. Poznyakoff, GNU cflow, 2005, Online; https://www.gnu.org/software/cflow/. (Accessed 26 May 2023).
[67] GNU, Function and macro index, 2023, Online; https://www.gnu.org/software/libc/manual/html_node/Function-Index.html. (Accessed 12 May 2023).
[68] I. Corporation, Standard C library functions table, 2023, Online; https://www.ibm.com/docs/en/i/7.3?topic=extensions-standard-c-library-functions-table-by-name. (Accessed 12 May 2023).
[69] M. Korobov, K. Lopuhin, ELI5, 2016, Online; https://github.com/TeamHG-Memex/eli5. (Accessed 4 September 2023).
[70] A. Costin, J. Zaddach, A. Francillon, D. Balzarotti, A large-scale analysis of the security of embedded firmwares, in: Proceedings of the 23rd USENIX Conference on Security Symposium, SEC '14, USENIX Association, USA, 2014, pp. 95–110.
[71] C. Solutions, GNU binutils, 2024, Online; https://www.gnu.org/software/binutils/. (Accessed 14 April 2024).
[72] Corvus Forensics, VirusShare.com - Because sharing is caring, Online; https://virusshare.com/. (Accessed 12 May 2023).
[73] S. Sebastián, J. Caballero, AVclass2: Massive malware tag extraction from AV labels, in: Annual Computer Security Applications Conference, ACSAC '20, Association for Computing Machinery, New York, NY, USA, 2020, pp. 42–53.
[74] D.S. Cruzes, L. ben Othmane, Empirical research for software security, in: Threats to Validity in Empirical Software Security Research, CRC Press, 2017, p. 26.
[75] C. Pang, R. Yu, Y. Chen, E. Koskinen, G. Portokalidis, B. Mao, J. Xu, SoK: All you ever wanted to know about x86/x64 binary disassembly but were afraid to ask, in: 2021 IEEE Symposium on Security and Privacy, SP, 2021, pp. 833–851.
[76] M. Jiang, Q. Dai, W. Zhang, R. Chang, Y. Zhou, X. Luo, R. Wang, Y. Liu, K. Ren, A comprehensive study on ARM disassembly tools, IEEE Trans. Softw. Eng. 49 (2023) 1683–1703.
[77] X. Meng, B.P. Miller, Binary code is not easy, in: Proceedings of the 25th International Symposium on Software Testing and Analysis, in: ISSTA 2016, Association for Computing Machinery, New York, NY, USA, 2016, pp. 24–35.
[78] O. Alrawi, C. Lever, K. Valakuzhy, R. Court, K. Snow, F. Monrose, M. Antonakakis, The circle of life: A Large-Scale study of the IoT malware lifecycle, in: 30th USENIX Security Symposium, USENIX Security 21, USENIX Association, 2021, pp. 3505–3522, URL: https://www.usenix.org/conference/usenixsecurity21/presentation/alrawi-circle.
[79] A.A. Al Alsadi, K. Sameshima, J. Bleier, K. Yoshioka, M. Lindorfer, M. van Eeten, C.H. Gañán, No spring chicken: Quantifying the lifespan of exploits in IoT malware using static and dynamic analysis, in: Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security, ASIA CCS '22, Association for Computing Machinery, New York, NY, USA, 2022, pp. 309–321.

Javier Carrillo Mondéjar is an Assistant Professor at the University of Zaragoza. He received his Ph.D. in Advanced Information Technology from the University of Castilla-La Mancha. He has also been a visiting researcher at King's College London for 5 months and the University of Jyväskylä for 3 months. His research interests are related to malware detection and classification techniques, with a particular focus on IoT/firmware cybersecurity.

Ricardo J. Rodríguez is an Associate Professor at the University of Zaragoza, Spain, with a Ph.D. in Computer Science and Systems Engineering from the same institution. He has participated in multiple EU and national projects, serving as Principal Investigator on both locally and industry-funded initiatives. He is currently leading several R&D projects, including those funded by the Spanish Ministry of Science, Innovation, and Universities (MICIU) and the Spanish National Cybersecurity Institute (INCIBE). His research interests include digital forensics, program binary analysis, and system security. Dr. Rodríguez has also served in various roles in international conferences and journals, including as co-Conference Chair for DFRWS EU 2024.