Identifying Runtime Libraries in Statically Linked Linux Binaries

Javier Carrillo-Mondéjar, **Ricardo J. Rodríguez**

rjrodriguez@unizar.es ※ @RicardoJRdez ※ www.ricardojrodriguez.es

Universidad Zaragoza 1474

Dpto. de Informática e Ingeniería de Sistemas
Universidad de Zaragoza

NoConName 2024 (Barcelona, Spain)

Financiado por la Unión Europea NextGenerationEU

GOBIERNO DE ESPAÑA — MINISTERIO PARA LA TRANSFORMACIÓN DIGITAL Y DE LA FUNCIÓN PÚBLICA — SECRETARÍA DE ESTADO DE DIGITALIZACIÓN E INTELIGENCIA ARTIFICIAL

Plan de Recuperación, Transformación y Resiliencia

incibe_ INSTITUTO NACIONAL DE CIBERSEGURIDAD

$whoami

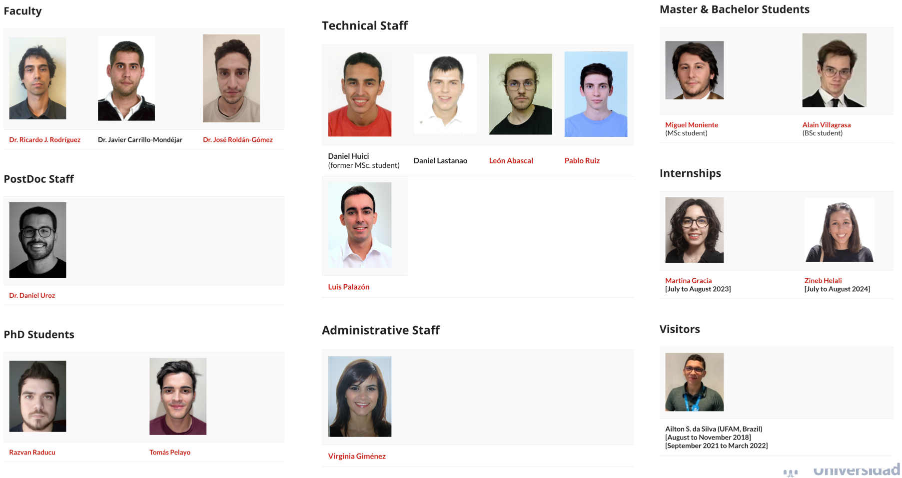
- **Associate Professor (PTU) @ UNIZAR**
- **Research lines**:
  - Software and application security
  - Digital forensics
  - System security
  - Formal methods applied to cybersecurity

$whoami

- **Associate Professor (PTU) @ UNIZAR**
- **Research lines**:
  - Software and application security
  - Digital forensics
  - System security
  - Formal methods applied to cybersecurity
- **Research team** – *we make funny stuff!* 🙂
  - `https://reversea.me`
  - `https://twitter.com/reverseame/`
  - `https://t.me/reverseame`

~~$whoami~~ $whoarewe

https://reversea.me/index.php/people/



Agenda

1 Introduction

2 MANTILLA: System Overview and Description

3 Dataset

4 Experiments and Results

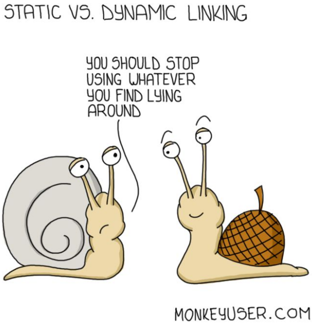
5 Limitations

6 Related Work

7 Conclusions and Future Work

Introduction

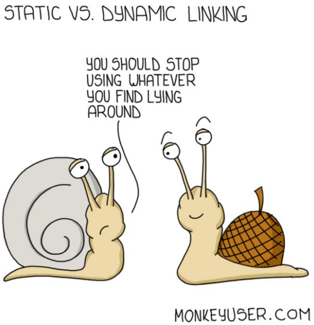
- **Unpatched apps can be exploited via 3rd-party program dependencies**
- **Static linking vs. dynamic linking**
  - Static linking includes all library dependencies in the binary
    - Complicates updates (and security)
  - Dynamic linking, in contrast, relies on external libraries that are linked at runtime
    - Makes updates easier, but introduces runtime dependency risks



Universidad Zaragoza

Introduction

- **Unpatched apps can be exploited via 3rd-party program dependencies**
- **Static linking vs. dynamic linking**
  - Static linking includes all library dependencies in the binary
    - Complicates updates (and security)
  - Dynamic linking, in contrast, relies on external libraries that are linked at runtime
    - Makes updates easier, but introduces runtime dependency risks



Malware developers exploit static linking to guarantee compatibility between platforms (e.g., IoT devices, Linux-based systems)

Universidad Zaragoza

Introduction

Binary code analysis in statically-linking binaries

How do we identify malware-related functions?

Challenges

- **Binary size increased**
- **Difficult to update libraries**
- **Lack of high-level abstractions in binary code**

Introduction

Binary code analysis in statically-linking binaries

How do we identify malware-related functions?

Challenges

- **Binary size increased**
- **Difficult to update libraries**
- **Lack of high-level abstractions in binary code**
- **Mix of malware and third-party binary code**
- **Compiler settings impact binary code generation**: more complex code

Universidad Zaragoza

Introduction
Contributions and results

MANTILLA

- A system to **identify runtime libraries in statically linked Linux binaries**
- **Static analysis** using features such as cyclomatic complexity, number of arguments, and entropy
- **Machine learning (KNN) for classifying binaries by runtime library**
- Evaluation results:
  - High accuracy in identifying runtime libraries and architecture (up to 98.6% on IoT malware)
  - Good performance on real-world IoT malware with diverse runtime libraries (`uClibc`, `glibc`, `musl`)
- MANTILLA **and dataset released for open science** under the GNU/GPLv3
  - Source code: `https://github.com/reverseame/MANTILLA`
  - Dataset: `https://zenodo.org/records/7991325`

Agenda

1 Introduction

2 MANTILLA: System Overview and Description

3 Dataset

4 Experiments and Results

5 Limitations

6 Related Work

7 Conclusions and Future Work

MANTILLA: System Overview and Description

Overview

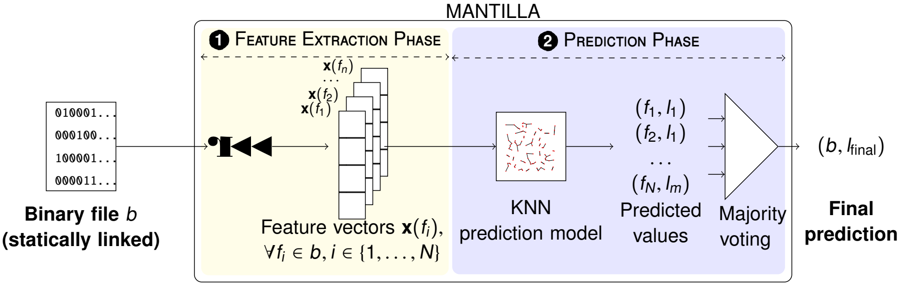
Two-phase workflow

- **Feature extraction phase:** extract features using static binary analysis (agnostic to architecture)
- **Prediction phase:** use KNN-based supervised learning to predict the runtime library
  - Classify the runtime library using KNN and majority voting

Universidad Zaragoza

MANTILLA: System Overview and Description

System Workflow Overview



MANTILLA: System Overview and Description

Extracted features (using `radare2`)

1. **Cyclomatic complexity** metric, $CC(f_i)$
2. **Size (in bytes) of the function**, $S(f_i)$
3. **Reserved stack space**, $SS(f_i)$
4. **Number of basic blocks**, $BB(f_i)$
5. **Number of edges**, $E(f_i)$
6. **Number of individual instructions** in the function, $I(f_i)$
7. **Number of arguments the function takes**, $A(f_i)$

MANTILLA: System Overview and Description

Extracted features (using `radare2`)

8. **Computational cost** of the function, $C(f_i)$
9. **Number of extended basic blocks**, $EBB(f_i)$
10. **Whether the function has an explicit return or not**, noret(f_i)
11. **Number of local variables declared within the function**, $L(f_i)$
12. **Entropy** of the bytes that make up the function, or $H(f_i)$
13. **Number of calls to other functions**
    - **Number of function calls** made within the function ($C_{\text{total}}(f_i)$)
    - **Number of unique functions** called by the function ($C_{\text{unique}}(f_i)$)

Universidad Zaragoza

MANTILLA: System Overview and Description

KNN-Based runtime library prediction

KNN algorithm

- For a new data point *d*, KNN finds the *K* closest examples
- Classifies *d* according to the most frequent label among the nearest neighbors

MANTILLA: System Overview and Description

KNN-Based runtime library prediction

KNN algorithm

- For a new data point *d*, KNN finds the *K* closest examples
- Classifies *d* according to the most frequent label among the nearest neighbors
- In our system, predictions are aggregated using majority voting to determine the final predicted library l_{final} for the entire binary
- **Advantages of KNN:**
  - Distance metric inherent to KNN allows discarding distant functions from the prediction
  - Our system is extensible to other clustering models (e.g., DBScan, K-means)

Universidad Zaragoza

MANTILLA: System Overview and Description

Threat model

Evasion techniques

- Adversaries may use obfuscation, packing, or junk code
- **Mitigation**: robust feature extraction, focusing on intrinsic properties of the binary

Adversarial machine learning attacks

- Adversaries may create adversarial samples to deceive the KNN classifier
- **Mitigation**: model validation, periodic updates, and adversarial training

MANTILLA: System Overview and Description

Threat model

Incompleteness or inaccuracy of extracted features

- Errors in feature extraction by `radare2` may lead to incorrect predictions
- **Mitigation**: verification processes and use of multiple binary analysis tools to cross-check features

Model drift and outdated training data

- New malware techniques may not be represented in training data
- **Mitigation**: Regular model updates and performance degradation detection

Limited scope

- The system may struggle to identify newer or less common runtime libraries.
- **Mitigation**: expand supported libraries and add new libraries through an update mechanism

Agenda

1 Introduction

2 MANTILLA: System Overview and Description

3 Dataset

4 Experiments and Results

5 Limitations

6 Related Work

7 Conclusions and Future Work

Dataset
Generation of ground truth

- Focus on C programming language (due to its popularity in malware)
- Toolchains created for various CPU architectures:
  - `MIPSeb`, `ARMel`, `Intel x86`, `Intel x86-64`
- Multiple runtime libraries considered: `uClibc`, `glibc`, `musl`
- Compilation with `buildroot` and `gcc` 10.2.1
  - Collection of algorithms from "TheAlgorithm" repository
  - Optimization options (specifically, `O0`, `O1`, `O2`, `O3`, and `Os`)
- Dataset of 13,800 statically linked, unstripped binaries
- Public release of the dataset for further research: https://zenodo.org/records/7991325

Universidad Zaragoza

Dataset

Generation of ground truth

Preprocessing steps

1. Extract functions used by the programmer using `cflow`
2. Disassemble binaries and remove standard C library functions
3. Retain external and static functions, excluding internal and library functions

Output: Features of non-discarded functions, labeled by architecture, runtime library, and compiler

Agenda

1 Introduction

2 MANTILLA: System Overview and Description

3 Dataset

4 Experiments and Results

5 Limitations

6 Related Work

7 Conclusions and Future Work

Universidad Zaragoza

Experiments and Results
Evaluation metrics

- $\text{Precision} = \frac{TP}{TP + FP}$
- $\text{Recall} = \frac{TP}{TP + FN}$
- $\text{F1-Score} = 2\frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$
- $\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$

Experiments and Results
Experiments performed

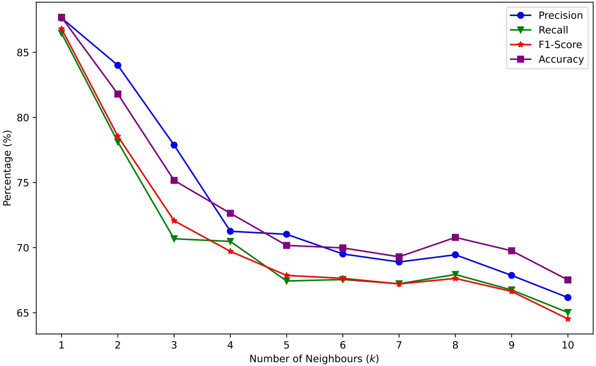
- **Validation**
- **Feature importance**
- **Evaluation on stripped binaries**
- **Architecture distinction**
- **Compiler and runtime library recognition**

All experiments use 5-fold cross-validation and are run with Python3 and Sklearn on a Debian 11 machine

Experiments and Results

Sensitivity of KNN hyperparameters

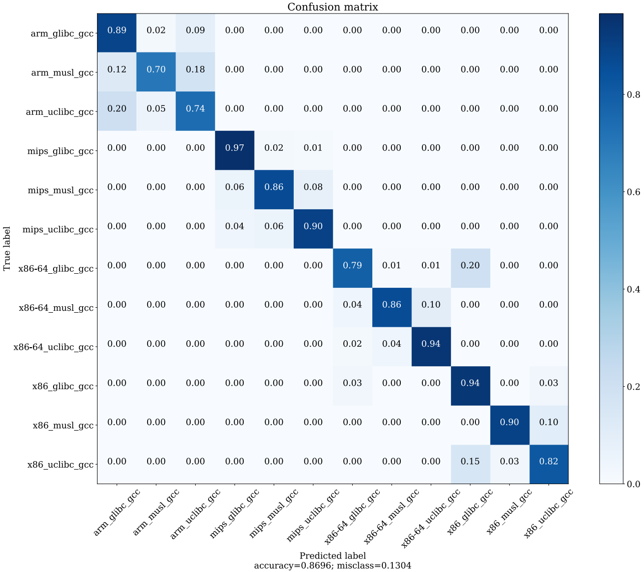
- **Number of neighbors**: $k = \{1, \ldots, 10\}$
- **Distance metrics**: Euclidean, Manhattan, and Minkowski distances
- **All results are very similar, no significant differences between them**



(results before applying the majority voting rule for the final prediction)

Experiments and Results

Sensitivity of KNN hyperparameters



Experiments and Results

Feature importance – Permutation importance technique

- Evaluates the importance of each feature by permuting its values and measuring the impact on model performance

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)

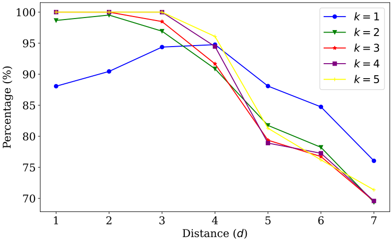
Accuracy of MANTILLA on stripped binaries

Experiment overview

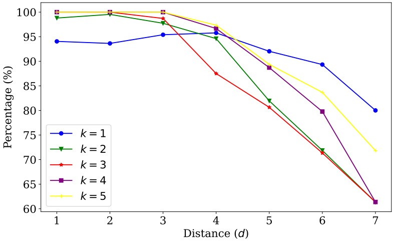
- Stripped and unstripped versions of binaries from the ground truth dataset
- Cross-validation with 80% training and 20% testing
- MANTILLA is trained with unstripped binaries, and tested on stripped ones
- KNN distances are used for prediction, with a threshold *d* to filter out unrelated functions

Experiments and Results

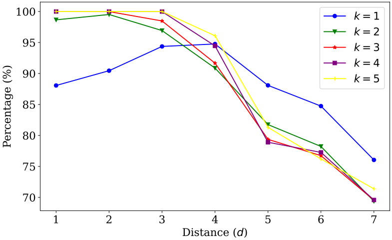
Accuracy of MANTILLA on stripped binaries – results



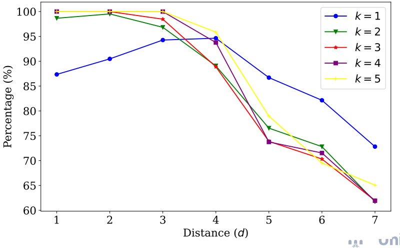
(a) Accuracy



(b) Precision



(c) Recall



(d) F1-Score

Experiments and Results

Architecture distinction

- The system identifies the architecture with very high accuracy before applying majority voting
- Misclassifications mainly occur between different runtime libraries within the same architecture
- The first misclassification occurs with $k = 3, d = 3$, where 19% of `x86-64_uclibc_gcc` binaries are misclassified as `x86-64_glibc_gcc`

Architecture identification is highly accurate, even before voting

Experiments and Results

Compiler provenance – results

- Added `Clang` (version 11.0.1-2) to the toolchain for `Intel x86` and `Intel x86-64` with `glibc`
- Dataset extended by 2,300 binaries
- Removed duplicate functions during training to avoid overfitting and reduce computational load

Experiments and Results

Compiler provenance – results

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

Experiments and Results

Compiler provenance – results

Results

- Compiler prediction accuracy is low, around 50%
- Both `gcc` and `Clang` use the same runtime library (`glibc`), leading to similar or identical neighbor distances
- Prediction performance depends on the order of training data, causing inconsistent results

Conclusion

- MANTILLA is not effective for determining the compiler used to compile a binary when both compilers use the same runtime library

Universidad Zaragoza

Agenda

1 Introduction

2 MANTILLA: System Overview and Description

3 Dataset

4 Experiments and Results

5 Limitations

6 Related Work

7 Conclusions and Future Work

Limitations

Construct validity

- Controlled experiments were conducted to adjust the system and measure evaluation metrics
- **No issues identified in the experimental study design**

Internal validity

- Third-party binary analysis tools (e.g., `radare2`) to extract features
- Accuracy of features influenced by the assumptions of these tools (e.g., instruction or function boundaries)
- **Easy interchangeability of feature extraction component**

Universidad Zaragoza

Limitations

KNN model considerations

- KNN is sensitive to the order of the training data when distances are tied, mitigated by K-fold cross-validation
- KNN is sensitive to high-dimensional data, addressed by limiting features to those common across architectures
- Sensitivity to the distance threshold in the voting phase, tested to assess performance under various settings

Limitations

- Tailored for binaries in the C programming language: errors likely to occur with binaries from other languages
- Statically linked binaries compiled on GNU/Linux systems
- Accuracy may decrease with obfuscated or packed binaries

Limitations

- Tailored for binaries in the C programming language: errors likely to occur with binaries from other languages
- Statically linked binaries compiled on GNU/Linux systems
- Accuracy may decrease with obfuscated or packed binaries

Extensibility

- Can be extended to other operating systems and platforms
- Future work includes additional hardware architectures (e.g., `PowerPC`, `SPARC`)
- Other C runtime libraries (e.g., `bionic`, `dietlibc`) were not considered, but can be added

Agenda

1 Introduction

2 MANTILLA: System Overview and Description

3 Dataset

4 Experiments and Results

5 Limitations

6 Related Work

7 Conclusions and Future Work

Related Work

Compiler provenance

- Early work by Rosenblum et al. uses CRF to identify compiler families
- `BinComp` performs syntactic, structural, and semantic analysis using the Jaccard coefficient for function similarity
- `FOSSIL` identifies free/open-source software (FOSS) packages and compiler provenance in real-world binaries
- `HIMALIA` uses RNNs for identifying optimization levels in binaries
- `Vestige` uses graph neural networks for provenance identification

Related Work

Compiler provenance

- Early work by Rosenblum et al. uses CRF to identify compiler families
- `BinComp` performs syntactic, structural, and semantic analysis using the Jaccard coefficient for function similarity
- `FOSSIL` identifies free/open-source software (FOSS) packages and compiler provenance in real-world binaries
- `HIMALIA` uses RNNs for identifying optimization levels in binaries
- `Vestige` uses graph neural networks for provenance identification

Authorship attribution

- `OBA2` detects software library functions based on syntax/semantics
- `BinAuthor` filters out compiler-related features
- `BinChar` uses CNNs, based on structural/semantic features

Related Work

Library function identification

- `IDA Pro FLIRT` forms signatures for recognizing library functions
- `BinHash` uses semantic hash functions to detect function similarities
- `libv` uses subgraph isomorphism for library function identification
- `discovRE` applies maximum common subgraph isomorphism
- `Genius` detects similar functions using high-level features for IoT firmware
- `BinShape` identifies library functions with robust signatures

Related Work

Library function identification

- `IDA Pro FLIRT` forms signatures for recognizing library functions
- `BinHash` uses semantic hash functions to detect function similarities
- `libv` uses subgraph isomorphism for library function identification
- `discovRE` applies maximum common subgraph isomorphism
- `Genius` detects similar functions using high-level features for IoT firmware
- `BinShape` identifies library functions with robust signatures

Machine learning approaches

- `Asm2Vec`, `DeepBinDiff`, and `PalmTree` leverage ML and neural networks for binary code similarity and diffing

Related Work

Library function identification

- `IDA Pro FLIRT` forms signatures for recognizing library functions
- `BinHash` uses semantic hash functions to detect function similarities
- `libv` uses subgraph isomorphism for library function identification
- `discovRE` applies maximum common subgraph isomorphism
- `Genius` detects similar functions using high-level features for IoT firmware
- `BinShape` identifies library functions with robust signatures

Machine learning approaches

- `Asm2Vec`, `DeepBinDiff`, and `PalmTree` leverage ML and neural networks for binary code similarity and diffing

Comparison with our work:

- Focus on identifying runtime libraries, not binary similarity
- Most related work focuses on function libraries and compiler analysis, not runtime libraries

Agenda

1 Introduction

2 MANTILLA: System Overview and Description

3 Dataset

4 Experiments and Results

5 Limitations

6 Related Work

7 Conclusions and Future Work

Conclusions and Future Work

Conclusions

- MANTILLA: a system for identifying runtime libraries in statically linked ELFs
  - Analyzes binary files, extracting architecture-independent features via `r2`, and uses KNN with majority voting for predictions
- Evaluations with cross-validation show high accuracy, with improved results using relaxed distance thresholds and higher K values
  - 94.4% accuracy on `binutils`
  - 95.5% accuracy on IoT malware datasets
- Achieved 100% and 98.6% accuracy for predicting binary architecture

Future Work

- Support additional architectures, operating systems, and runtime libraries
- Provide it as a web service for integration into third-party analysis workflows

Identifying Runtime Libraries in Statically Linked Linux Binaries

Javier Carrillo-Mondéjar, **Ricardo J. Rodríguez**

rjrodriguez@unizar.es ※ @RicardoJRdez ※ www.ricardojrodriguez.es

Universidad Zaragoza 1474

Dpto. de Informática e Ingeniería de Sistemas
Universidad de Zaragoza

NoConName 2024 (Barcelona, Spain)

Financiado por la Unión Europea NextGenerationEU

GOBIERNO DE ESPAÑA

MINISTERIO PARA LA TRANSFORMACIÓN DIGITAL Y DE LA FUNCIÓN PÚBLICA

SECRETARÍA DE ESTADO DE DIGITALIZACIÓN E INTELIGENCIA ARTIFICIAL

Plan de Recuperación, Transformación y Resiliencia

incibe_ INSTITUTO NACIONAL DE CIBERSEGURIDAD