Predicting the Success of x86-64 Binary Rewriters

Akshay Sood
GrammaTech, Inc.
asood@grammatech.com

Keara Hill*
Binghamton University
khill9@binghamton.edu

Kimble D. Houck*
kimble.d.houck@khouck.net

Zachary P. Fry
GrammaTech, Inc.
zfry@grammatech.com

Jonathan Dorn
GrammaTech, Inc.
jdorn@grammatech.com

ABSTRACT

Binary rewriters aim to modify the behavior of binary executables without having access to their source code. While there has been significant research on binary rewriting tools and techniques, assessing the effectiveness of binary rewriters remains a time consuming and challenging task that impedes wider adoption by the community. Differences in capabilities among rewriters, as well as varying but generally large computational costs, make it difficult for users to select rewriters appropriate to their specific workloads. In this paper, we propose a machine learning-based approach to predict the success of binary rewriters for x86-64 binaries, allowing users to quickly judge the likelihood of various binary transforms. We compare various learning algorithms for modeling a range of binary rewriting tools, explore different feature representations, and examine the performance of models trained on a diverse set of programming languages. Our findings provide insights into learning algorithms, feature representations, and the generalizability of models across programming languages for this task. We are able to model rewriting success with a high level of predictive performance, as measured by the area under the curve (AUC), enabling users to make much faster judgment of rewriter effectiveness on a given binary, at a fraction of the computational cost of actual rewriting. Our work also contributes to the understanding of the strengths and weaknesses of different binary rewriters and empowers tool developers and researchers to make informed decisions regarding the application of rewriting tools.

CCS CONCEPTS

• **Security and privacy → Software reverse engineering**.

KEYWORDS

Binary Analysis, Binary Recompilation, Binary Rewriting

ACM Reference Format:
Akshay Sood, Keara Hill, Kimble D. Houck, Zachary P. Fry, and Jonathan Dorn. 2023. Predicting the Success of x86-64 Binary Rewriters. In *Proceedings of The 7th Annual Symposium on Machine Programming (MAPS '23)*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

*Work done at GrammaTech, Inc.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MAPS '23, December 3, 2023, San Francisco, CA, USA
© 2023 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Binary rewriting refers to the process of modifying the behavior or properties of a compiled binary executable without access to its source code. It has gained significant attention due to a wide range of potential applications, including patching security vulnerabilities, code obfuscation, software instrumentation, performance optimization, cross-platform adaptation, among others.

Researchers have proposed a multitude of approaches for binary rewriting, and recent works have performed comprehensive surveys of binary rewriting approaches [18] and empirical evaluations of existing rewriting tools [14]. However, there continue to exist gaps in our understanding of the generalizability and limitations of binary rewriters. For instance, rewriters are primarily developed for and tested on commercial off-the-shelf (COTS) binaries, which are often programmed in C and C++, especially for applications requiring low-level access to hardware or high-performance capabilities. However, different languages exhibit unique coding styles, idioms, and semantic nuances that may affect the applicability of rewriting techniques. Rewriting tools and techniques can also differ widely in their computational costs, and some rewriters may be prohibitively slow or resource-intensive for binaries with specific characteristics. For these reasons, it is challenging for users to select rewriters appropriate to their setting and difficult for researchers and tool developers to identify weaknesses that may need to be addressed through new techniques and tool improvements.

In this work, we use a machine-learning-based approach for modeling binary rewriter success to address some of these challenges. Machine-learning techniques have been applied to various areas of binary analysis, such as disassembly [20], function identification [15], similarity detection [12], and malware detection [2]. Our work focuses on the application of machine-learning to predicting rewriting success across a range of binary rewriting tools.

The main contributions of this paper are threefold. First, we extract an informative set of features from binaries and perform a comparison of various learning algorithms for predicting rewriting success across a range of binary rewriting tools. Second, we evaluate the performance of models trained on a diverse set of programming languages and investigate the generalizability of these models across different language ecosystems. Third, we compare the predictive value of features that we have engineered for this task against a baseline set of features. We further examine which features are most important for rewriting viability and for predicting rewriting success to inform user decision-making.

2 RELATED WORK

Our work builds upon the comparative evaluation of rewriting tools by Schulte et al. [14]. The authors evaluate the performance of several rewriters on two rewriting tasks over x86-64 binaries compiled from a set of thirty-four C/C++ programs. They also train decision tree models over simple features extracted from the binaries in order to identify features predictive of rewriting success. In our work, we select a largely overlapping set of rewriters and model their ability at performing the same rewriting tasks. However, our work focuses on predictive modeling of the rewriters' success rather than on evaluating the rewriters themselves. We perform a comprehensive investigation of modeling approaches for predicting rewriting success that includes (i) collecting data for an augmented set of programs, including several non-C/C++ languages, (ii) extracting and engineering more advanced binary features in order to gain more predictive and explanatory value, (iii) evaluating several classes of models, varying in their complexity, for predicting rewriting success, and (iv) investigating the generalizability of models trained on different language sub-groups. We also make improvements to the quality of the data collected and generate more accurate estimates of rewriter performance on the set of programs in our data set.

3 METHODS

3.1 Data

3.1.1 Tool Selection. We select seven rewriting tools for predictive modeling: *ddisasm* [8], *dyninst* [4], *e9patch* [7], *egalito* [19], *reopt* [11], *retrowrite* [6], and *zipr* [10]. Except for *dyninst*, the tools selected are a subset of the tools evaluated by Schulte et al. [14], and are chosen based on rewriting ability, tool maturity, adequate robustness and performance for data collection, in addition to ensuring coverage of a breadth of popular rewriting techniques. The techniques represented include direct rewriting (*dyninst*, *egalito* and *zipr*), reassemblable disassembly (*ddisasm* and *retrowrite*), LLVM rewriting (*reopt*), and trampoline rewriting (*e9patch*). Note that while *dyninst* supports both static and dynamic instrumentation, we only model its static rewriting capabilities in this work. More generally, we focus on modeling static rewriters and exclude dynamic rewriters due to scalability and performance reasons.

3.1.2 Tasks. We treat the problem of predicting rewriting success as a binary classification task, where for each rewriting tool considered, a classifier is trained to predict whether the tool will succeed or fail at a given rewriting task. Following the evaluation in Schulte et al. [14], we consider two types of rewriting tasks:

Relowering. Relowering represents the ability of a rewriting tool to process the input binary, transform it into an internal or external intermediate representation, and generate a new binary. The tool is considered to be successful at this task if it generates a new binary that passes a smoke test to validate that the binary is executable. While this represents an identity transform, it is non-trivial for most tools, and the rewritten binaries are generally distinct from the original. However, some tools handle analysis errors by incorporating reasonable defaults (such as by linking code from the original binary on rewriting failure, or by preserving unidentified symbols which continue to resolve correctly if the code is left undisturbed in memory). Three of the seven tools considered, namely, *dyninst*, *e9patch*, and *zipr*, achieve near-perfect success rates for this task, and are thus excluded from the set of rewriters modeled for relowering. For these reasons, we also model the tools' rewriting ability at a more challenging task that better represents a real-world application of rewriting.

AFL++ Instrumentation. This task evaluates the tool's ability to transform a binary with AFL++ instrumentation [1]. This process represents a rewriting task that is more complex but of greater interest than relowering, since it measures the ability of the tool to transform a binary into a representation that is useful for downstream rewriting applications, such as gray-box fuzzing. Although all the tools that we have selected for modeling claim to support instrumentation with AFL++, *egalito* and *reopt* failed to generate any executable binaries instrumented with AFL, and are excluded from predictive models trained for this task.

3.1.3 Binaries. We collect data on 2,650 binary samples generated using permutations of compilation variants of 58 benchmark programs, compiled on Ubuntu 20.04 for x86-64. We include the 34 C/C++ programs used in the comparative evaluation by Schulte et al. [14], along with two other C/C++ programs: the VLC media player [17], as well as a trivial "Hello, World!" program in C++ to serve as a low-water mark for program complexity.

In addition, we collect data on several programs from different open-source projects in non-C/C++ languages, in order to model rewriting ability over a diverse set of source programming languages. We compile binaries for, and evaluate rewriters on, 7 FORTRAN programs, 5 Go programs, 5 Haskell programs, and 5 OCaml programs.

Table 1: Compilation variants used to generate binaries.

Compiler	Flags	Relocation (position-)	Symbols
clang gcc icx gfortran	O0 O1 O2 O3 Os Ofast	Independent Dependent	Present Stripped
OLLVM	fla sub bcf	Independent Dependent	Present Stripped
go	O0	Independent Dependent	Present Stripped
ghc	O0 O1 O2	Dependent	Present Stripped
ocaml	O1	Independent	Present Stripped

Table 2: Benchmark programs along with associated source language and number of samples in the data set.

Program	Language	Samples	Program	Language	Samples	Program	Language	Samples
alscal	FORTRAN	24	libreoffice	C/C++	22	proftpd	C/C++	84
anope	C/C++	80	libzmq	C/C++	40	qmail	C/C++	72
asterisk	C/C++	72	lighttpd	C/C++	42	redis	C/C++	42
bind	C/C++	48	magic-trace	OCaml	2	samba	C/C++	78
bitcoin	C/C++	56	memcached	C/C++	36	satisfy	FORTRAN	24
cli	Go	4	mirage	OCaml	2	satisfy_openmp	FORTRAN	24
compose	Go	4	monero	C/C++	72	sendmail	C/C++	80
crowdsec	Go	4	mosh	C/C++	80	shellcheck	Haskell	6
dijkstra	FORTRAN	24	mysql	C/C++	48	sipwitch	C/C++	80
dijkstra_openmp	FORTRAN	24	nginx	C/C++	80	snort3	C/C++	72
dnsmasq	C/C++	80	nsq	Go	4	sqlite	C/C++	84
dune	OCaml	2	openssh	C/C++	80	squid	C/C++	80
f77_to_f90	FORTRAN	24	openvpn	C/C++	84	unison	OCaml	2
filezilla	C/C++	72	osv-scanner	Go	4	unrealircd	C/C++	84
frama-c	OCaml	2	pandoc	Haskell	6	vim	C/C++	84
gnome-calculator	C/C++	10	pidgin	C/C++	84	vlc	C/C++	72
hello	C/C++	60	pks	C/C++	80	wordsnake	FORTRAN	24
implicitcad	Haskell	6	poppler	C/C++	36	xmonad	Haskell	6
kmonad	Haskell	6	postfix	C/C++	84	zip	C/C++	80
leafnode	C/C++	84						

We compile the programs in our data set using several compilation configurations, varying the choice of compiler, optimization flags, whether the binary is a position-independent executable (PIE), and whether the binary is stripped of debug and symbol information. Table 1 lists all the compilation variants used to generate the samples in our data set, and Table 2 lists the programs along with source language and number of samples for each program. We use and extend the open-source project *lifter-eval* [9] to execute binary rewriters on our suite of binary samples and collect labels for both relowering and AFL tasks.

3.1.4 Features. The compiled binaries are stored in the ELF file format, and we parse these files using the LIEF parser [16] to extract most of the features in our data set. For a baseline set of features, we collect all the features extracted by Schulte et al. [14], representing whether the binary is a position-independent executable, whether it contains debug and symbol information, and whether it contains any of 11 standard ELF sections. We also extract several other features, including various measures of binary and section size, and one feature (`num_indirect_instructions`) derived from a fast linear disassembly of the binary, performed using the Capstone disassembler [5]. Our feature engineering is driven by an iterative process: we investigate unexpected or interesting experimental findings, such as variations in rewriting success between programs of similar apparent complexity, and design new features that could potentially explain these findings. We focus on engineered rather than learned feature representations, since the structured nature of the binary file format lends itself to high-performing hand-crafted features in other domains such as malware detection [2]. Table 3 lists all the features that we extract from the binaries.

Table 3: List of features extracted from binaries. Features marked with an asterisk are included in Schulte et al. [14].

Feature	Description
`eh_frame_size`	Size of .eh_frame section
`has_section_data_rel_ro`*	Binary has section .data_rel_ro
`has_section_debug_str`*	Binary has section .debug_str
`has_section_gcc_except_table`*	Binary has section .gcc_except_table
`has_section_got_plt`*	Binary has section .got.plt
`has_section_interp`*	Binary has section .interp
`has_section_note_abi_tag`*	Binary has section .note.ABI-tag
`has_section_note_gnu_build_id`*	Binary has section .note.gnu.build-id
`has_section_plt_got`*	Binary has section .plt.got
`has_section_rela_plt`*	Binary has section .rela.plt
`has_section_strtab`*	Binary has section .strtab
`has_section_symtab`*	Binary has section .symtab
`is_debug_section_present`	Binary has any debug section
`is_pie`*	Position-independent executable
`is_stripped`*	Binary is stripped
`num_indirect_instructions`	Number of indirect instructions
`num_nonstandard_sections`	Number of non-standard sections
`size`	Size of binary
`text_section_size`	Size of .text section
`uses_dynamic_libraries`	Binary uses dynamic libraries
`uses_multithreading`	Binary uses multithreading
`uses_nonstandard_section`	Binary has any non-standard section
`uses_setjmp`	Binary uses setjmp for control flow

3.2 Model Training

We train and compare several modeling approaches, varying in their complexity, for predicting rewriting success. We train simpler models, namely L1-regularized logistic regression and decision trees, to establish baseline levels of performance as well as for their

explanatory value. We also train more complex models, namely multilayer perceptrons, random forests, and gradient boosted trees, in order to learn richer representations and establish upper bounds on predictive performance for this task and data set. We train the models using the machine-learning Python library scikit-learn [13].

We use k-fold cross-validation with $k = 4$ for model selection and evaluation. We attempt to mitigate risks of overfitting associated with a small data set and large variations in program size, complexity, language, and other characteristics in several key ways. We also try to ensure that the evaluation results are robust to variance in model performance arising from the choice of random seeds used to group binaries into cross-validation folds and to optimize the learning objectives. To these ends, we undertake the following measures:

- Binaries that correspond to different compilation variants of a single program are grouped together while splitting data for cross-validation. This ensures that related binaries for a given program are only used to either train, validate or test a given model, mitigating the risks of overfitting to specific programs or achieving overoptimistic assessments of generalization performance.
- Cross-validation folds are stratified so that each fold contains a similar proportion of binaries for each target label and language, as well as a similar proportion of binaries that represent libraries.
- Since the number of compilation variants can vary significantly by program, the samples are weighted in order to assign equal weight to each program, rather than each binary, in order to avoid biasing models towards programs for which we have more data available.
- Repeated k-fold cross-validation is performed for both model selection and evaluation, where each repetition corresponds to a different random seed used for splitting data into cross-validation folds and training the model. Randomized hyperparameter search [3] is used to identify 10 candidate sets of hyperparameters, and repeated k-fold cross-validation with 10 repetitions is used to select the optimal set of hyperparameters from these. Then, models are trained and evaluated using the optimal set of hyperparameters over 50 repetitions of k-fold cross-validation. To compute the final performance results, the model selection and evaluation procedure is repeated with five different random seeds used for randomized hyperparameter search.

Feature Selection. We perform two feature selection transformations in the modeling pipeline. First, variance thresholding is used to remove features with low or zero variance. Second, real or integer-valued features are standardized to have zero mean and unit variance to assist learning algorithms that are sensitive to differential feature scales (i.e., logistic regression and multilayer perceptrons).

Hyperparameters. We perform randomized hyperparameter search for model selection, with search spaces and budgets customized for each type of model. The following hyperparameters are optimized for each model:

- Decision trees: maximum depth, split strategy (best vs. random), minimum number of samples required for splits/leaves, minimum decrease in impurity required for splits, class weight
- Logistic regression: L1 regularization penalty, class weight
- Multilayer perceptron: hidden layer size, initial learning rate, L2 regularization penalty
- Random forests: number of trees in ensemble, maximum depth per tree, maximum number of features considered per split, class weight
- Gradient boosted trees: number of trees in ensemble, maximum depth per tree, maximum number of features considered per split, learning rate

4 EXPERIMENTAL RESULTS

In this section, we present results of experiments formulated to investigate several key questions for the predictive modeling of binary rewriters:

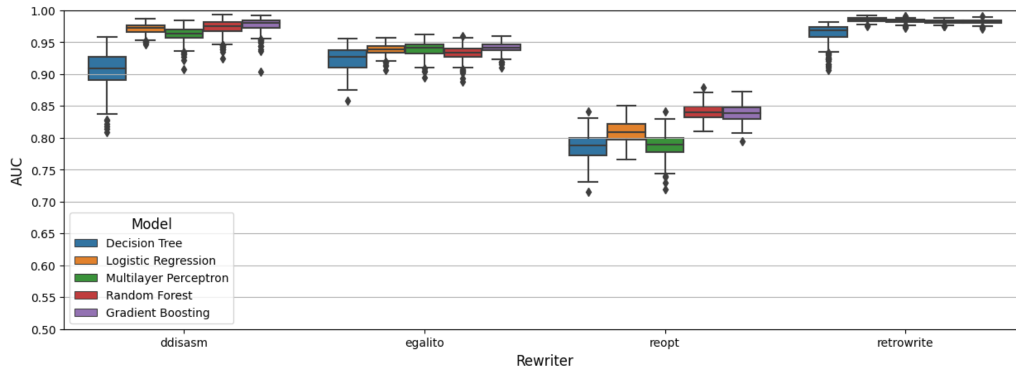
- What learning algorithms allow us to achieve the highest level of predictive performance for modeling binary rewriters? Do we gain any benefits from using more complex, black-box models over simpler, more interpretable models for predicting rewriting success?
- Do binary rewriters perform equally well on C/C++ and non-C/C++ languages? Does our ability to predict rewriting success depend on which languages are included in the training data?
- Can we design features to improve model performance for predicting rewriting success? What features provide the greatest predictive value for this task?

4.1 Comparative Evaluation of Models

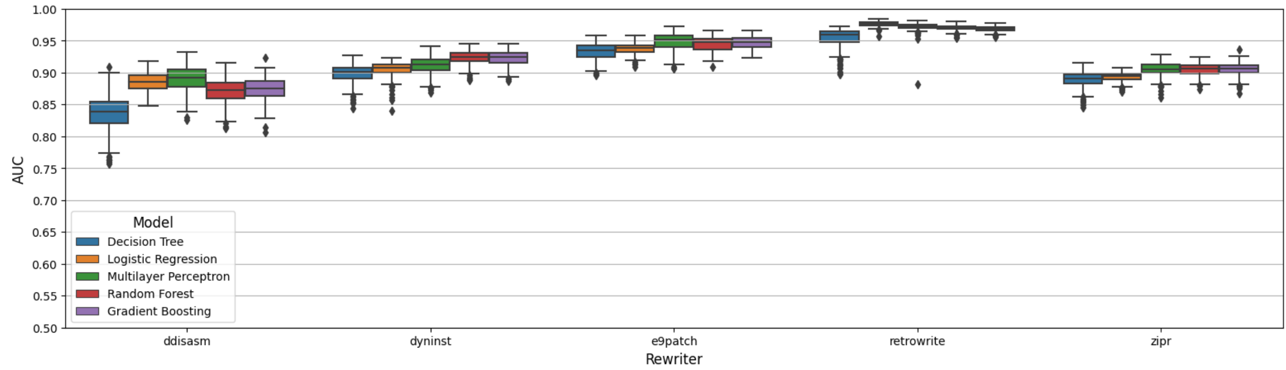
In order to identify which models are best-suited for predicting rewriting success, and to establish upper bounds of predictive performance, we perform a comparative evaluation of various models, ranging in complexity, for predicting rewriting success of several rewriting tools for both tasks. Figures 1a and 1b show model performance for predicting relowering and AFL success respectively, measured in terms of area under the curve (AUC). While decision trees generally perform worse than other models for most rewriters for both tasks, no model consistently outperforms the rest. We therefore restrict ourselves to logistic regression models for other experiments owing to their ease of interpretability and lower computational costs for training.

4.2 Effect of Languages on Rewriting Success and Predictive Performance

Tables 4 and 5 show rewriting success rates grouped by language for relowering and AFL tasks, respectively. Note that success rates for each language are computed using equal weights assigned to each binary for that language. The rewriters exhibit varying levels of success across different languages, and this variation is not consistent across the two rewriting tasks. This suggests two possibilities: (i) the differences in rewriting performance across languages arise due to differing distributions of binaries for each language, with no language-specific effects on rewriting ability, or (ii) rewriting



(a) Model performance for predicting relowering success.



(b) Model performance for predicting AFL success.

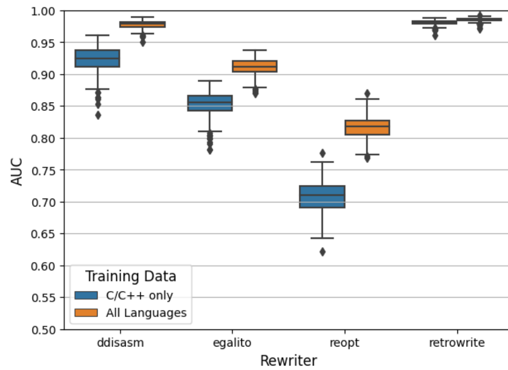
Figure 1: Comparisons of model performance across multiple rewriters for (a) relowering and (b) AFL prediction tasks. Each box plot represents AUCs for a given model, rewriter and prediction task, aggregated over 250 points, corresponding to 5 sets of hyperparameters identified after model selection and 50 random seeds used for model evaluation. Each point represents the average AUC across four cross-validation folds. *dyninst*, *e9patch*, and *zipr* are not modeled for relowering and *egalito* and *reopt* are not modeled for AFL due to ~100% and 0% rewriting success rates, respectively. AUC values below 0.50 are omitted.

Table 4: Rewriting success rates for relowering.

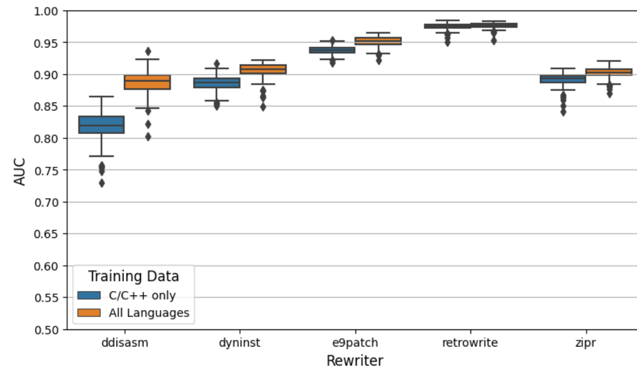
Language	Binaries	*ddisasm*	*dyninst*	*e9patch*	*egalito*	*reopt*	*retrowrite*	*zipr*
FORTRAN	168	1.00	1.00	1.00	0.49	0.91	0.25	1.00
C	2422	0.88	1.00	1.00	0.19	0.46	0.08	0.99
Ocaml	10	1.00	1.00	1.00	0.00	0.00	0.00	0.00
Go	20	0.00	0.60	0.20	0.00	0.00	0.00	0.00
Haskell	30	0.00	1.00	1.00	0.00	0.00	0.00	0.00

Table 5: Rewriting success rates for AFL.

Language	Binaries	*ddisasm*	*dyninst*	*e9patch*	*egalito*	*reopt*	*retrowrite*	*zipr*
FORTRAN	168	0.71	0.45	0.36	0.00	0.00	0.18	0.36
C	2422	0.80	0.42	0.44	0.00	0.00	0.09	0.29
OCaml	10	0.90	0.00	1.00	0.00	0.00	0.00	0.00
Go	20	0.05	0.00	0.00	0.00	0.00	0.00	0.00
Haskell	30	0.00	0.30	0.00	0.00	0.00	0.00	0.00

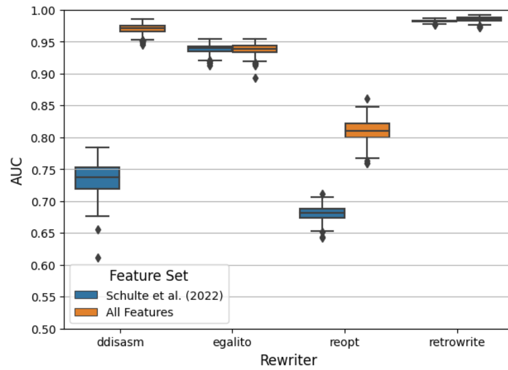


(a) Model performance for predicting relowering success.

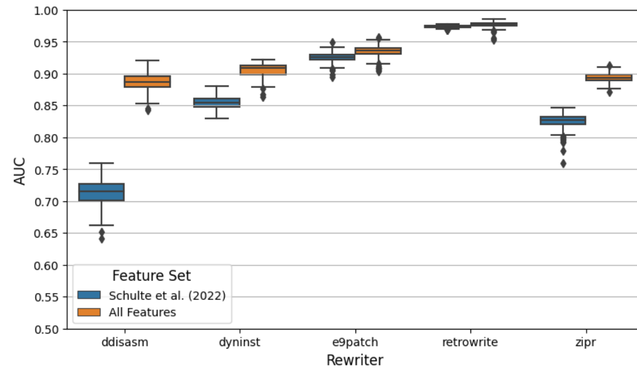


(b) Model performance for predicting AFL success.

Figure 2: Performance of logistic regression models across multiple rewriters for (a) relowering and (b) AFL prediction tasks, showing comparisons between models trained on C/C++-only programs and models trained on all languages. Both types of models are evaluated on all languages. AUCs are computed as in Figure 1, and AUC values below 0.50 are omitted.



(a) Model performance for predicting relowering success.



(b) Model performance for predicting AFL success.

Figure 3: Performance of logistic regression models across multiple rewriters for (a) relowering and (b) AFL prediction tasks, showing comparisons between models trained on features from Schulte et al. [14] and the set of all features listed in Table 3. AUCs are computed as in Figure 1, and AUC values below 0.50 are omitted.

performance is affected by language-specific characteristics of the binaries.

To better understand the reasons behind this variation, we compare the performance of logistic regression models trained on programs in all languages with models trained on programs in C/C++ only. If the source language has little or no effect on rewriting performance, we might expect models trained on programs in C/C++ to generalize equally well on both C/C++ and non-C/C++ programs.

Figures 2a and 2b show model performance for predicting relowering and AFL success, respectively. For all the rewriters examined, models that are trained on programs in all languages perform better than models that are trained on C/C++ programs only, when evaluated on programs in all languages. This suggests that certain characteristics of programs in non-C/C++ languages may provide novel information for modeling these languages, and that variations in rewriting ability may be associated with these characteristics.

4.3 Identifying Important Features

To understand the impact of the extracted features shown in Table 3, we compare the performance of models trained on the entire set of extracted features with models trained on the baseline set of features from Schulte et al. [14]. Figures 3a and 3b show model performance for predicting relowering and AFL success, respectively. For some rewriters (e.g., *ddisasm*), models that are trained on all features perform significantly better than models trained on the baseline features, suggesting that we are able to identify salient properties of the binaries that are not captured by the baseline features for these rewriters.

In order to identify the features that are most useful for predicting rewriting success, we further examine feature coefficients in the trained logistic regression models. The magnitude of a given feature's coefficient serves as an indicator of the feature's importance to the model, while the sign of the coefficient indicates whether the feature is positively or negatively correlated with predicting

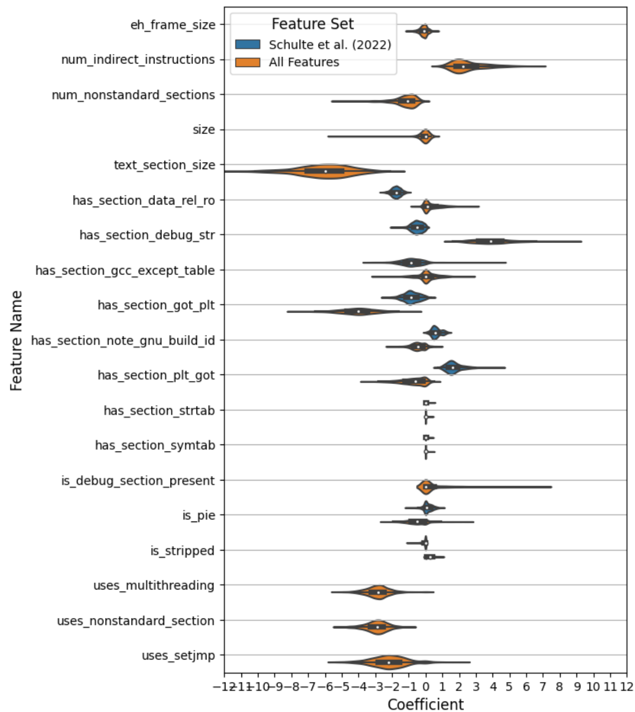


Figure 4: Violin plots showing the distribution of feature coefficients for logistic regression models trained to predict relowering success for *ddisasm*. Quartiles are indicated using miniature box plots inside the violins, shown in black. Features from Schulte et al. [14] are only shown when included in the models.

rewriter success. Figure 4 shows distributions of feature coefficients for models trained to predict relowering success for *ddisasm*. Features `text_section_size`, `uses_multithreading`, `num_nonstandard_sections`, `uses_nonstandard_section`, and `uses_setjmp` are negatively associated with predicting relowering success, suggesting that binaries that are larger, use multithreading, include non-standard sections, or have unusual control flow are harder for *ddisasm* to rewrite. On the other hand, coefficients for `is_stripped` and `is_pie` have small variance and are centered around zero, suggesting that *ddisasm* is robust to binaries that are stripped or non-PIE, unlike other rewriters such as *retrowrite* [14]. Features with larger variance that are centered at or near zero, e.g., `is_debug_section_present`, may indicate overfitting. Coefficients for certain features tend to change signs going from baseline features to all features, e.g., `has_section_plt_got`. A possible explanation for this is that new features that are correlated with `has_section_plt_got` may provide the model with some information that was previously extracted from `has_section_plt_got`, leading to the model extracting novel information from `has_section_plt_got` when all features are present.

5 DISCUSSION

We make the following observations based on these results:

- We are able to predict rewriting success with a high level of performance. The performance varies by rewriter, choice of model, and rewriting task, with AUC ≥ 0.80 for the best-performing models for all rewriters and prediction tasks, and AUC ≥ 0.95 in many cases.
- No model consistently outperforms other models across the range of rewriters and rewriting tasks. Simpler and more interpretable logistic regression models perform comparably to more complex, black-box models for predicting binary rewriting success. This suggests that richer representations do not yield additional predictive value for this task, given features that are largely extracted from parsing the binaries and for the amount of data available.
- For most rewriters, models trained on programs in all languages perform better than models trained on C/C++ programs only, suggesting that both rewriting performance, and the performance of models trained to predict rewriting success, are affected by language-specific characteristics of the binaries. Since rewriters are predominantly tested on COTS binaries compiled from programs in C/C++, investigating features of binaries specific to non-C/C++ languages may be a promising line of research for improving cross-language performance of binary rewriters.
- The set of features that we extract from the binaries provides sufficient predictive value to train performant models for predicting rewriting success. In many cases, our features outperform the set of baseline features from Schulte et al. [14]. Feature importance analysis of the trained models shows which features are most useful for predicting rewriting success. These features may be used to elicit strengths and weaknesses of different rewriters in order to drive improvements in rewriting methods.

6 CONCLUSION

In this work, we investigated the problem of predicting the success of binary rewriters across a diverse set of binaries. We compiled binaries using various compilation configurations for a set of 58 benchmark programs written in a range of popular languages. We collected data on rewriting success for these binaries using a representative set of tools for two rewriting tasks, relowering and AFL++ instrumentation.

We extracted features from the binaries by parsing them and performing fast linear disassembly. We used these features to train models to predict rewriting success at both the aforementioned tasks. We performed a comparative evaluation of models of varying complexity for these tasks. We examined rewriting success rates on binaries compiled from different source languages, and investigated the performance of models both with and without access to languages other than C/C++ in the training data. Finally, we examined the predictive value of our features in comparison to a baseline set of features and conducted a feature importance analysis to identify features that are most useful for predicting the success of a given rewriter.

Our results show that we were able to learn performant models for predicting rewriting success, and that simpler, more interpretable models performed comparably to more complex models.

We observed that models trained on programs in all languages performed better than models trained only on C/C++ programs, suggesting that both rewriting and model performance are affected by language-specific characteristics of the binaries. We showed that models trained using our feature set performed better than those trained on the baseline set of features for many rewriters. By analyzing these models, we were also able to identify salient binary features associated with rewriting ability for a given rewriter. Ultimately, the developed models can be used to aid users' decisions related to binary rewriters in diverse contexts, speeding up the vetting process for popular transforms.

7 ACKNOWLEDGMENTS

We would like to thank Vlad Folts for his help in data collection. This material is based upon work supported by the Office of Naval Research (ONR) under Contract No. N00014-21-C-1032. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the ONR.

REFERENCES

[1] AFL++. 2023. The AFL++ Fuzzing Framework. https://aflplus.plus/.

[2] Hyrum S. Anderson and Phil Roth. 2018. EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models. https://doi.org/10.48550/arXiv.1804.04637 arXiv:cs/1804.04637

[3] James Bergstra and Yoshua Bengio. 2012. Random Search for Hyper-Parameter Optimization. *Journal of Machine Learning Research* 13, 10 (2012), 281–305.

[4] Andrew R. Bernat and Barton P. Miller. 2011. Anywhere, Any-Time Binary Instrumentation. 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, 9–16. https://doi.org/10.1145/2024569.2024572

[5] Capstone. 2023. The Ultimate Disassembly Framework. https://www.capstone-engine.org/.

[6] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. 2020. RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization. In *2020 IEEE Symposium on Security and Privacy (SP)*. 1497–1511. https://doi.org/10.1109/SP40000.2020.00009

[7] Gregory J. Duck, Xiang Gao, and Abhik Roychoudhury. 2020. Binary Rewriting without Control Flow Recovery. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 151–163. https://doi.org/10.1145/3385412.3385972

[8] Antonio Flores-Montoya and Eric Schulte. 2020. Datalog Disassembly. In *29th USENIX Security Symposium (USENIX Security 20)*. 1075–1092.

[9] GrammaTech. 2023. Lifter Eval. https://gitlab.com/GrammaTech/lifter-eval.

[10] William H. Hawkins, Jason D. Hiser, Michele Co, Anh Nguyen-Tuong, and Jack W. Davidson. 2017. Zipr: Efficient Static Binary Rewriting for Security. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 559–566. https://doi.org/10.1109/DSN.2017.27

[11] Galois Inc. 2023. Reopt. https://github.com/GaloisInc/reopt

[12] Andrea Marcelli, Mariano Graziano, Xabier Ugarte-Pedrero, Yanick Fratantonio, Mohamad Mansouri, and Davide Balzarotti. 2022. How Machine Learning Is Solving the Binary Function Similarity Problem. In *31st USENIX Security Symposium (USENIX Security 22)*. 2099–2116.

[13] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[14] Eric Schulte, Michael D. Brown, and Vlad Folts. 2022. A Broad Comparative Evaluation of X86-64 Binary Rewriters. In *Cyber Security Experimentation and Test Workshop*. 129–144. https://doi.org/10.1145/3546096.3546112 arXiv:cs/2203.13231

[15] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. 2015. Recognizing Functions in Binaries with Neural Networks. In *24th USENIX Security Symposium (USENIX Security 15)*. 611–626.

[16] Romain Thomas. 2017. LIEF - Library to Instrument Executable Formats. https://lief.quarkslab.com/.

[17] VideoLan. 2006. VLC media player. https://www.videolan.org/vlc/index.html

[18] Matthias Wenzl, Georg Merzdovnik, Johanna Ullrich, and Edgar Weippl. 2019. From Hack to Elaborate Technique—A Survey on Binary Rewriting. *Comput. Surveys* 52, 3 (June 2019), 49:1–49:37. https://doi.org/10.1145/3316415

[19] David Williams-King, Hidenori Kobayashi, Kent Williams-King, Graham Patterson, Frank Spano, Yu Jian Wu, Junfeng Yang, and Vasileios P. Kemerlis. 2020. Egalito: Layout-Agnostic Binary Recompilation. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 133–147. https://doi.org/10.1145/3373376.3378470

[20] Sheng Yu, Yu Qu, Xunchao Hu, and Heng Yin. 2022. DeepDi: Learning a Relational Graph Convolutional Network Model on Instructions for Fast and Accurate Disassembly. In *31st USENIX Security Symposium (USENIX Security 22)*. 2709–2725.