2023 IEEE/ACM 31st International Conference on Program Comprehension (ICPC) | 979-8-3503-3750-1/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICPC58990.2023.00042

Revisiting Deep Learning for Variable Type Recovery

Kevin Cao
Vanderbilt University
Nashville, USA
kevin.cao@vanderbilt.edu

Kevin Leach
Vanderbilt University
Nashville, USA
kevin.leach@vanderbilt.edu

***Abstract*—Compiled binary executables are often the only available artifact in reverse engineering, malware analysis, and software systems maintenance. Unfortunately, the lack of semantic information like variable types makes comprehending binaries difficult. In efforts to improve the comprehensibility of binaries, researchers have recently used machine learning techniques to predict semantic information contained in the original source code. Chen et al. implemented DIRTY, a Transformer-based Encoder-Decoder architecture capable of augmenting decompiled code with variable names and types by leveraging decompiler output tokens and variable size information. Chen et al. were able to demonstrate a substantial increase in name and type extraction accuracy on Hex-Rays decompiler outputs compared to existing static analysis and AI-based techniques. We extend the original DIRTY results by re-training the DIRTY model on a dataset produced by the open-source Ghidra decompiler. Although Chen et al. concluded that Ghidra was not a suitable decompiler candidate due to its difficulty in parsing and incorporating DWARF symbols during analysis, we demonstrate that straightforward parsing of variable data generated by Ghidra results in similar retyping performance. We hope this work inspires further interest and adoption of the Ghidra decompiler for use in research projects.**

***Index Terms*—Ghidra, Hex-Rays, Machine Learning, Transformers**

I. Introduction

Program comprehension is a critical part of developing and maintaining large software systems. Many analysis and comprehension tools operate on program source code, such as code similarity comparison [1], automated program repair [2], and fault localization [3]. However, when dealing with legacy systems or proprietary software, researchers and reverse engineers often have access only to the distributed compiled binaries. Binary files are frequently viewed as more difficult to analyze than source code files, in part because assembly and machine language lack the semantics and structure present in a higher-level programming language. Although debugging formats such as DWARF allow compilers to build binaries with debugging symbols to integrate semantic information within the binary, release builds often omit these symbols to improve performance and conceal intellectual property.

To analyze binary files, reverse engineers use decompilers to transform a binary file (or single function) into an equivalent source code representation, which in turn helps engineers comprehend the original semantics of the file or function in question. This transformation augments the sequential, type-agnostic nature of assembly programming with abstractions such as variables names and types. However, without the original semantic information, decompilers are unable to provide meaningful names to these variables, instead assigning names and types based on implementation-specific naming conventions (e.g., 'uVar1').

With the success of machine learning models in the domains of natural language processing [4] and programming language analysis [5], researchers are proposing machine learning models to recover missing semantic information in binaries. These models leverage the insight that semantic information present in source code is context-dependent: variables that appear and are used in similar contexts tend to be assigned similar names and types [6]. One such approach is the DIRTY (**D**ecomp**I**led variable **R**e**TY**per) model proposed by Chen et al. [7], [8], which adapts the Transformer architecture to predict variable names and types in decompiler outputs. Due to its focus on variable types, DIRTY leverages both the decompiled source code tokens as well as the object layout for all variables found during decompilation to improve model performance. This architecture, combined with the novel use of variable layouts, resulted in a model capable of identifying correct variable types 75.8% of the time.

One main concern with the DIRTY model is its ability to generalize to different decompiler outputs, as a swath of tools are commonly used in reverse engineering [9]–[12]. The authors tested their implementation solely using the commercial Hex-Rays decompiler, whose expensive licensing restrictions are outside the budget of many researchers and hobbyist reverse engineers. For this reproducibility study, we address this concern by training the DIRTY architecture with a dataset decompiled using the open-source Ghidra decompiler. We observe that the performance of this neural architecture is comparable for both Hex-Rays decompiler outputs and Ghidra decompiler outputs.

The use of Ghidra is addressed in the DIRTY paper, where the authors concluded that the inability of Ghidra to reliably obtain correct ground truth semantic information from debugging symbols in the data processing stage harmed the performance of the model. While we do not address specific decompilation algorithms in this paper, we adapt the dataset used in DIRTY by including only those binaries decompiled

2643-7171/23/$31.00 ©2023 IEEE
DOI 10.1109/ICPC58990.2023.00042

by Ghidra that contain corresponding DWARF symbolic information that can serve as adequate ground truth. We show that this dataset permits the DIRTY architecture to generalize to decompiler output produced with Ghidra for variable naming and typing tasks. We hope that this reproducibility study will aid future researchers seeking to incorporate Ghidra and other reverse engineering tools into datasets and evaluations.

In summary, we demonstrate that DIRTY's architecture for the task of variable type prediction generalizes to our newly curated Ghidra dataset.

II. Related Work

Binaries have been an important artifact for applying techniques to aid reverse engineering tasks. While machine learning models exist that analyze binary files directly, these models often target specific tasks such as optimization level recovery that leverage insights on broad binary characteristics such as instruction frequency [13]. Models like DIRTY attempt to provide reverse engineers with additional semantics to augment source code produced by decompilers, ultimately improving comprehensibility.

Like other machine learning techniques tackling binary analyses such as binary similarity [14], vulnerability detection [15], and malware analysis [16], DIRTY trains on decompiled functions. Informally, constructing the dataset for DIRTY involves decompiling a set of binaries for which DWARF symbols are available, resulting in tuples of (assembly code, variable names), which in turn forms ground truth that can be used for training. This is possible because the decompiler interface is scriptable, allowing programmatic access to the internal abstract syntax tree, object layouts, and underlying type system to transform the decompiled code produced by the base set of decompilation passes. These features exist in all decompiler implementations because they are fundamental to the construction of higher level languages. Therefore, researchers can leverage these features to build general machine learning models that eliminate implementation-specific details.

The original DIRTY architecture was trained solely using the Hex-Rays decompiler. Despite the rich integration supported by Hex-Rays, other tools are frequently used in reverse engineering and binary comprehension tasks. The Hex-Rays decompiler is a closed-source software product entailing licenses specific to one backend CPU target (e.g., x86, ARM on Windows, Linux), which limits augmentation and widespread adoption. This can pose substantial barriers for those seeking to use Hex-Rays in larger, more complex machine learning techniques that depend on large volumes of decompiled code, precluding independent researchers and hobbyists from advancing the field.

Many decompiler frameworks have been released to address these concerns. In this paper, we choose to augment DIRTY by retraining the neural architecture on a dataset derived from Ghidra[1], a public version of the decompiler used by the National Security Agency released in 2019. Due in part to its recent release, features such as native scripting capabilities and DWARF parsing lag behind their Hex-Rays counterparts. However, adapting binary analysis techniques to work with Ghidra is important to ensure generalizable scientific findings that increase accessibility to important tools that advance the state-of-the-art in binary analysis and decompilation. Several other reverse engineering frameworks exist such as angr [12], a framework that goes beyond decompilation and disassembly by introducing features such as symbolic execution, Radare2[2], a framework cenetered around disassembly that offers interfacing with external decompilers, and Binary Ninja [9], a framework focusing on UI usability and automation while still offering the same scripting capabilities as other frameworks. We ultimately select Ghidra due to its similar capabilities to Hex-Rays and community interest in seeing Ghidra-based decompilation projects.

III. Experimental Design

A. Data Processing and Training

```
void ff_cavsdsp_init_x86(CAVSDSPContext* c,
                         AVCodecContext* avctx)
{
    uint uVar1;
    int cpu_flags;
    uVar1 = av_get_cpu_flags();
    if ((uVar1 & <Num>) != <Num>) {
        cavdsp_init_mmx(c, avctx);
    }

    if ((uVar1 & <Num>) != <Num>) {
        cavsdsp_init_3dnow(c, avctx);
    }

    if ((uVar1 & <Num>) != <Num>) {
        cavsdsp_init_mmxext(c, avctx);
    }

    return;
}
```

Fig. 1: Example of aliasing in Ghidra. While Ghidra finds the cpu_flags variable in the DWARF section, Ghidra assigns flag information to an unnamed uVar1 variable. These two variables are difficult to programmatically combine into one variable since they have different memory locations.

In this section, we introduce the experimental design for our replication study. First, we introduce the DIRT dataset, which serves as the basis for the dataset we curated to train our Ghidra-specific DIRTY model. We then introduce the processing, training, and testing of the model while highlighting differences between the original implementation when necessary. For future reference, we refer to $\text{DIRTY}_{\text{Hex-Rays}}$ as the original model implementation in the DIRTY paper and $\text{DIRTY}_{\text{Ghidra}}$ as our model trained on Ghidra decompiled functions.

[1]Ghidra: https://github.com/NationalSecurityAgency/ghidra

[2]Radare2: https://github.com/radareorg/radare2

B. DIRT and Dataset Creation

The **D**ataset for **I**diomatic **R**e**T**yping is a dataset created from randomly sampling C and C++ projects scraped from GitHub using GHCC [3], which attempts to compile each project with debugging symbols by recursively identifying Makefiles in the project's sub-directories. Each binary file is decompiled using Hex-Rays to obtain ground truth information about variable names and layouts. The binary files are then stripped of their debugging symbols and decompiled once again to serve as indicative outputs obtained from real stripped binaries in the wild. The final **DIRT** dataset consists of a collection of decompiled code tokens with variables clearly marked as well as object size (e.g., struct size) information obtained from the decompiler for all variables in the function.

We are interested in training the DIRTY model using output from the Ghidra decompiler. While we could compile our own set of binaries to create our dataset, assembling enough binaries to create a dataset comparable in size with the DIRT dataset is a large undertaking. More importantly, we fear that changes in compilation environments such as compiler versions might undermine the ability to directly compare model results. Although only the processed versions of the DIRT testing and training sets are publicly available, Chen et al. were kind enough to provide the underlying collection of binaries used to select the DIRT binaries. Using this collection of binaries, we created our dataset by combining the original binaries used in the DIRT dataset along with an additional set of 90,000 randomly sampled binaries to ensure we had a dataset size of a similar order of magnitude after preprocessing. We decompile these binaries using Ghidra and extract the C-code functions and object layouts using Ghidra's underlying decompiler interface.

As seen in Fig. 1, we observed that the Ghidra decompiler suffers from a form of variable aliasing when decompiling binaries with debugging information. While Ghidra recovers the `cpu_flags` DWARF variable and includes it in the decompiled function source, Ghidra fails to assign values to this variable, instead assigning `cpu_flags` information to a decompiler-created `uVar1` variable, which does not contain any debugging information. Although reverse engineers reading this function might be able to see that `uVar1` aliases the `cpu_flags` variable and thus combine the two variables to create a cleaner, more accurate decompiled function, these two variables do not reside in the same memory location in Ghidra, which prevents us from confidently merging these two variables during preprocessing. It becomes more impractical to implement variable merging when multiple variables are being aliased: the correct mapping between variables is now context-dependent. This seems to be the issue that Chen et al. faced when trying to reproduce their own work using Ghidra.

Although resolving variable aliasing is an important undertaking requiring substantial reverse engineering expertise, we identify two insights that allow us to work around this problem. First, the concept of aliasing is not specific to the Ghidra decompiler: typical decompilers face this issue since binaries lack rich semantics that are lost during compilation. Indeed, we see that, for some functions, Ghidra correctly uses variables found in DWARF debugging sections, and Hex-Rays suffers from a similar problem in variables suddenly appearing when decompiling stripped binaries compared to binaries with debugging information. Secondly, we note that this inability to correctly assign variables with DWARF types actually conveys semantic information, since it signifies that the variable is either a temporary or was combined with other variables in the source code during analysis.

Thus, we do not remove such "bad" DWARF variables from the training set because they are variables encountered by Ghidra during analysis, and we do not want to artificially bias the results of our replicated model by introducing a bad data exclusion step not present in the original published results. Therefore, we annotate these variables with a special *disappear* label to signify that DWARF information was lost during decompilation. Like *primitive*, *pointer*, and *structure* data types, this *disappear* type can be predicted by the model during retyping tasks.

To the best of our knowledge, Ghidra does not provide an interface to determine whether or not a variable contains DWARF information. Instead, before decompilation, we manually parse the DWARF information ourselves to extract the DWARF names of all variables declared in the binary. Thus, during decompilation, if a variable is found that does not match a name discovered during our DWARF parsing stage, we replace its type with our *disappear* datatype.

We implement various other techniques to filter the functions decompiled by Ghidra, summarized as follows.

1) We enforce a decompilation time limit of 3 minutes to prevent the decompiler from stalling on large executables.
2) We exclude functions that Ghidra fails to decompile cleanly (e.g., external functions) to prevent the model from training on incomplete function signatures and bodies.
3) We exclude functions that do not reference parameters or contain local variables in the function body as they are not useful for the variable naming task.

The breakdown between the DIRT original dataset and our curated version of the dataset is shown in Table I and Table. II. We see that variable aliasing issues are more prevalent in Ghidra, where 62.15% of Ghidra variables lack debugging information compared to the 20.13% in Hex-Rays.

C. Evaluation

To evaluate the predictions made by the Transformer model, we employ the same metrics as in the DIRTY paper. A variable type prediction is correct if and only if the type name, along with all other sub-fields if dealing with structure types, are equivalent to the developer-assigned type (as defined in the original source code) using direct string comparison. While this evaluation metric is strict as incorrect names can still provide important semantic information, string comparison is

[3]GHCC: https://github.com/huzecong/ghcc

TABLE I: Comparison between the datasets used to train $\text{DIRTY}_{\text{Hex-Rays}}$ and $\text{DIRTY}_{\text{Ghidra}}$. Due to the high number of *disappear* variables and low number of *struct* variables, we address these types separately from others during evaluation.

		Variables			
Model	# Binaries	# Variables	% Structs	% Disappear	# Unique Functions
$\text{DIRTY}_{\text{Hex-Rays}}$	76,472	3,689,018	5.05	20.13	727,617
$\text{DIRTY}_{\text{Ghidra}}$	73,232	1,676,346	2.45	62.15	399,130

TABLE II: Comparison between the datasets used to evaluate $\text{DIRTY}_{\text{Hex-Rays}}$ and $\text{DIRTY}_{\text{Ghidra}}$. A lower In-Train percentage for the Ghidra dataset signifies less shared library functions in the Ghidra testing and training sets.

		Variables			Functions			
Model	# Binaries	# Variables	% Structs	% Disappear	# Functions	% All Disappear	% No disappear	% In-Train
$\text{DIRTY}_{\text{Hex-Rays}}$	9,461	1,031,844	1.83	8.76	203,876	0.87	74.84	65.5
$\text{DIRTY}_{\text{Ghidra}}$	9,153	307,083	1.22	42.78	86,870	26.1	34.71	45.04

TABLE III: Comparison between the variable typing accuracies of the two models, separated by structs, variables labeled *disappear*, and recovered variables. While the Not In-Train accuracies for $\text{Hex-Rays}_{\text{Ghidra}}$ might be slightly higher, a lower % In-Train causes the Overall accuracies to be similar.

		Overall				In-Train				Not In-Train		
Model	Overall	Structs	Disappear	No Disappear	Overall	Structs	Disappear	No Disappear	Overall	Structs	Disappear	No Disappear
$\text{DIRTY}_{\text{Hex-Rays}}$	73.9	65.8	84.7	72.8	88.1	76.6	93.4	87.5	54.3	51.7	71.5	52.8
$\text{DIRTY}_{\text{Ghidra}}$	73.5	61.7	84.7	65.2	91.2	90.7	93.1	89.5	64.1	57.1	79.6	53.2

both simple to implement and deterministic. By keeping the evaluation metrics the same, we ensure a fair comparison in the final results since any changes in model accuracy can be attributed to differences in the decompiler outputs of Ghidra and Hex-Rays instead of nuances in the evaluation scheme.

IV. Results

In this section, we aim to answer the following research question: Is the Transformer architecture employed in DIRTY as effective at predicting variable names and types for a dataset decompiled with Ghidra instead of Hex-Rays?

Recall that we retrained the neural architecture in DIRTY using our Ghidra-decompiled dataset. We obtained a pre-trained DIRTY model, then evaluated our trained Ghidra model compared to their pre-trained Hex-Rays model on the task for variable retyping. The results are shown in Table III. Due to the low amount of structure variables and high levels of *disappear* variables in the Ghidra dataset, we isolate these variable types and obtain their accuracies separately.

We see that the overall performance of the two models have similar overall accuracy, with $\text{DIRTY}_{\text{Hex-Rays}}$ retyping variables correct 73.9% of the time and $\text{DIRTY}_{\text{Ghidra}}$ retyping variables correct 73.5% of the time. While $\text{DIRTY}_{\text{Hex-Rays}}$ and $\text{DIRTY}_{\text{Ghidra}}$ have similar accuracies for retyping variables in functions encountered during training, $\text{DIRTY}_{\text{Ghidra}}$ performs slightly better than $\text{DIRTY}_{\text{Hex-Rays}}$ when retyping variables in functions not seen during training.

For variables without corresponding DWARF variable information, $\text{DIRTY}_{\text{Ghidra}}$ outperforms $\text{DIRTY}_{\text{Hex-Rays}}$ by 5%. While the increase in prediction rates can be attributed to the increased proportion of these variables in the Ghidra dataset, the high rate of accurate predictions suggests that for both the Hex-Rays and Ghidra decompilers, there are certain underlying patterns that unify cases in which DWARF semantic information is unrecoverable.

For variables with corresponding DWARF variable information, $\text{DIRTY}_{\text{Ghidra}}$ performs similarly to $\text{DIRTY}_{\text{Hex-Rays}}$, correctly predicting the DWARF types 53.2% and 52.8% of the time respectively. This result shows that the Ghidra model's ability to predict types was not adversely affected by the greater number of *disappear* variables in the Ghidra dataset.

Overall, these results match our intuition and provide additional evidence that the model described in DIRTY can also apply to decompiled source code obtained from Ghidra.

V. Conclusion

In this paper, we extend the work of Chen et al. by demonstrating that for their DIRTY architecture, a model trained on a Ghidra dataset yields similar retyping accuracies as the original model trained on a Hex-Rays dataset. The model trained on Ghidra decompiler output is capable of correctly predicting DWARF debugging types 73.5% of the time and displays similar accuracies for function encountered during training and novel functions. Our results add confidence to the claim that DIRTY's Transformer architecture extends to different decompilers besides Hex-Rays. We hope that our initial results spark more confidence and adoption of the Ghidra decompiler in related research projects.

VI. Acnkowledgments

We acknowledge Jeremy Lacomis and Claire Le Goues from the DIRTY paper for graciously providing the original dataset of binaries and scripts used to train their published model.

REFERENCES

[1] S. Schleimer, D. S. Wilkerson, and A. Aiken, "Winnowing: Local Algorithms for Document Fingerprinting," in *Proceedings of the 31st ACM SIGMOD International Conference on Management of Data*. ACM, 2003, pp. 76–85.

[2] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *IEEE Trans. Software Eng.*, vol. 38, no. 1, pp. 54–72, 2012.

[3] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller, "Evaluating and improving fault localization," in *ICSE 2017, Proceedings of the 39th International Conference on Software Engineering*, Buenos Aires, Argentina, May 2017, pp. 609–620.

[4] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[5] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[6] E. Avidan and D. G. Feitelson, "Effects of variable names on comprehension: An empirical study," in *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, 2017, pp. 55–65.

[7] Q. Chen, J. Lacomis, E. J. Schwartz, C. Le Goues, G. Neubig, and B. Vasilescu, "Augmenting decompiler output with learned variable names and types," in *31st USENIX Security Symposium*, Boston, MA, Aug. 2022. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/chen-qibin

[8] J. Lacomis, P. Yin, E. J. Schwartz, M. Allamanis, C. Le Goues, G. Neubig, and B. Vasilescu, "Dire: A neural approach to decompiled identifier naming," in *34th IEEE/ACM International Conference on Automated Software Engineering*, San Diego, CA, 2019, pp. 628–639.

[9] "Binary ninja decompiler," https://binary.ninja/, accessed: 2023-02-15.

[10] "Remill binary to llvm translator," https://github.com/lifting-bits/remill, accessed: 2023-02-15.

[11] "Ghidra decompiler," https://ghidra-sre.org/, accessed: 2023-02-15.

[12] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "Sok: (state of) the art of war: Offensive techniques in binary analysis," in *2016 IEEE Symposium on Security and Privacy (SP)*, 2016, pp. 138–157.

[13] L. Chen, Z. He, H. Wu, F. Xu, Y. Qian, and B. Mao, "Dicomp: Lightweight data-driven inference of binary compiler provenance with high accuracy," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2022, pp. 112–122.

[14] S. Ahn, S. Ahn, H. Koo, and Y. Paek, "Practical binary code similarity detection with bert-based transferable similarity learning," in *Proceedings of the 38th Annual Computer Security Applications Conference (ACSAC)*, Dec. 2022.

[15] J. T. Giffin, S. Jha, and B. P. Miller, "Efficient context-sensitive intrusion detection." in *NDSS*, 2004.

[16] M. Ahmadi, K. Leach, R. Dougherty, S. Forrest, and W. Weimer, "Mimosa: Reducing malware analysis overhead with coverings," 2021. [Online]. Available: https://arxiv.org/abs/2101.07328