Litrature Review

Abdelsalam ElTamawy
School of Science and Engineering
the American University in Cairo
solomspd@aucegypt.edu

Rinal

Andrew Fahmy
School of Science and Engineering
the American University in Cairo
andrewk.kamal@aucegypt.edu

Ahmed

Ramy

Dina

I. Introduction

Although reverse engineering a binary program has a relatively mature ecosystem of tools such as Ghidra and Binary Ninja, they all still involve a large amount of manual time and effort to bring back some sense of structure and human readability to the assembly code. This is mainly due to their dependence on pattern matching and other rule-based approaches, introducing many limitations to traditional decompilers, including poor scalability and development. Since decompilation is crucial to multiple cybersecurity domains such as malware analysis and vulnerability discovery, the more this process is enhanced, the more we can offer faster results to analysts and reverse engineers.

II. Problem Description

A. Summary

One of the core conceptual problems with decompilation is that although one can deterministically compile code into the assembly, the reverse is untrue; there are countless forms that code can take that produce the same assembly. To this end, the deterministic nature of rule-based decompilers fails to pick which of the countless routes to take. Machine learning offers a heuristical approach to settling these uncertainties, allowing us to get as close as possible to how a human has written it. Furthermore, x86 architecture has been given plenty of focus in recent papers, so this leads our attention to see what would happen if we use these techniques on other architectures, namely android and its Java virtual machine. Moreover, besides the typical compilation problems that are present in any decompiler(e.g., syntactic distortion, semantic incorrectness, and significant difficulty with code readability), android apps decompilers show a considerable failure rate in recovering programs as well as a significant bias towards a specific goal/usage of the decompiler. In addition, android decompilers perform differently across different applications, and the output of the decompiled code is heavily dependent on the compiler used to compile the source code. We believe that the more expressive nature of java byte code can open up better prediction and more straightforward extrapolation of the underlying structure due to java's more strict object-oriented paradigm. Also, this is to provide tools to combat the current rise in security concerns on the mobile platform.

B. Proposed Solution

This project aims to propose a new platform for credible Android apps, ML-based decompilation, to raise the accuracy of code recovery. Our proposed project adopts and improves on major successful models that dramatically enhance the performance of desktop apps decompilation. Hence, beyond simply migrating well-established solutions from `C` binaries to bytecode, we also propose a 3 phase solution to addressing the decompilation problem in Android applications, each phase leveraging a different approach to extracting information from the cryptic binary.

III. Literature Review

Machine learning-based decompilation has been explored in several research projects before. Most of the previous work that addressed this problem focused solely on recovering high-level `C` based on `x86` architecture. Thus, besides reviewing the literature dedicated to ML-based systems and techniques for decompiling `C` binaries, we also present a complete review of all the popular platforms and applications used for decompiling Android applications.

A. Binary enrichment

1) Debin: Debin, a prediction system, aims to deal with the stripped binary that contains low-level information, which is often unknown due to optimization purposes, or even worse, to hide malicious and vulnerable code. Debin tries to automate this process with machine learning techniques instead of relying on rule-based techniques to recover this stripped information such as variables and types. In other words, Debin was trained on thousands of non-striped binaries and then used to predict properties of meaningful elements in unseen stripped binaries. Debin's support for binary files extends to three architectures: `x86`, `x64`, and `ARM` with high levels of precision. Debin mainly uses two probabilistic methods to implement the recovery of the variables, the Extremely randomized Tree and the linear graphical model for the prediction of the extracted program's elements, such as their names and types.

Debin follows five main steps to produce the improved and enhanced binary file. Firstly, the stripped binary file is taken as input and lifted from assembly code into an intermediate Binary Analysis Platform (BAP-IR). The reason for this transition is to have a high-level form of the semantics available, generalize the syntax among the three architectures Debin targets, and have a better understanding of the logic instructions. BAB-IR preserves the semantics of the basic instructions, explicitly shows the operations on machine states, and recognizes the function boundaries via its ByteWeight component to obtain the code elements. The next step would be to extract two types of data from the intermediate: the known and unknown elements. The unknown elements are the ones that have been lost in the compilation process due to optimization stripping. In contrast, the known elements are the already present properties in the binary code, and no need to infer any information. Examples of known elements would be Dynamically linked library (DDL) functions, flag, instruction, unary operator, constant, and location nodes. Also, temporarily allocated registers and memory offsets are treated as known nodes and do not need any name or type prediction.

After acquiring the known and unknown elements, a dependency graph is built and formed with nodes and their relationship as edges. In the next crucial step, Debin starts to infer the unknown elements through the probabilistic method, Maximum a Posterior (MAP). The relationship between elements is represented in the form (a,b, rel). These three words represent node `A` connected to node `B` with a relationship between them rel. Those relationships can vary between functions, variables, types, and factor relationships. Finally, after the predictions are applied to the unknown elements, an updated binary file is released as output and ready with improved debug information.

Debin dataset consisted of `9000` non-stripped binary executables initially written in pure C language, `3000` for each of their targeted architectures. They gave 2700 binary files as training data and left the remaining 300 binary files for testing and prediction purposes. Also, as mentioned by the authors of [1], Debin uses a binary classifier to evaluate better the updated binary file's accuracy based on the following formula, Accuracy = $\frac{|TP|+|TN|}{|P|+|N|}$. After going through all 9000 binary files and evaluating them, variable recovery reached 90.6%; type recovery reached 73.8%, name prediction accuracy up to 63.2%, and structured prediction up to 63% precision.

DEBIN's significant fundamental limitations exist in predicting the contents of struct and union types. Also, the trained-based output of different compilers works well in predicting symbols omitted in compilation, yet it does not perform well with the intentional use of obfuscation or human-based written assembly. Also, another limitation lies in the fact that DEBIN uses BAP for the binary analysis stage. The problem with BAP is that it is impossible to prove the correctness of its code lifting mechanism as the semantics of ISAs like x86 are not formally defined. Therefore, it uses random tests to find the differences between their code lifting results and the behavior of an actual processor. Moreover, BAP lifting process expects aligned sequences of instructions. This is done by manually supplying the locations of code or through the use of recursive analysis tools. The problem of the latter approach is to obtain 100% accuracy because of the presence of indirect branches or functions without explicit calls. Thus, it is hard to have a precise image of the control flow. These problems are to be solved to a reasonable extent using **Probabilistic Disassembly**

2) Probabilistic Disassembly: Analyzing and reversing software has many applications; however, it is challenging because the source code is usually not present. The first problem is how to disassemble the software accurately. This process is highly complex because of the diversity in the compilation and optimization techniques. The paper discusses two popular disassemblers: linear sweep disassemblers and traversal disassemblers. Linear sweep follows the byte order, whereas traversal disassembling follows the control flow of function calls and jumps. The problem with a linear sweep is that it introduces many false positives and even false negatives, especially when data and code interleaves. On the other hand, traversal disassemblers suffer indirect control flow like in the switch-case statement.

The paper also discusses **Superset Disassembly**. It is the state-of-the-art technique in rewriting/instrumentation. The idea behind this approach is to consider every starts an instruction, called superset instruction. Consecutive superset instructions can share standard bytes. Superset disassemblers have no false-negative but must have a bloated code body because of the many superset instructions that are false positives.

The paper then discusses the approach they proposed about probabilistic disassembling. This approach inherits the strengths of superset disassembling that it produces no false negatives. This is because true positives exhibit many hints indicating that they are accurate instructions. However, hints are not specific; thus, false-positive instructions also have a low chance of exhibiting the same features.

In x86, part of a valid instruction may be another valid instruction, and also, two good instructions could have overlapping bodies, and these are called **occluded instructions**. A problem with occluded instructions is that they can be cascaded. So, if we start from the wrong place, many following instructions are occluded. However, this cascading is highly unlikely. The excellent point is that if one of the sequences is the true positive, the occluded sequences quickly converge with the true positive. The authors also noticed that the suffix of instruction is likely to be another instruction. This is based on the **occlusion rule** which states that occluded sequences tend to agree on a common suffix of instructions quickly.

The paper then discusses what the authors call **probabilistic hints.** Simply this is a way of predicting that the analyzed bytes are valid instructions, not data. The first hint is **control flow convergence.** This is done by analyzing the bytes and finding more than jump to a proper instruction; this usually indicates that those bytes are more likely to be instructions, not data. The second hint is the **control flow crossing**. This happens when more than one jump instruction crosses each other. For example, when there are three instructions *inst1, inst2, inst3* and *inst2, inst3* are adjacent to each other, and *inst1* jumps to *inst3* and *inst2* jumps to instruction before *inst1*. Since it is doubtful that data bytes can form two control flow instructions with one jumping right after the other, they are most likely to be instructions. The third hint is **register define-use relation**.

A pair of instructions *inst1, inst2* have a register define-use relation when *inst1* defines a values of a register and *inst2* uses it. For example, when a comparison instruction sets a flag and then uses it by a conditional jump. Those hints indicate that the analyzed bytes are not data, but they do not assure true positives. This is because they may be occluded instructions part of some ground truth instructions, as they share similar features such as register operands. Fortunately, the occlusion rule tells us that it will automatically be corrected if there is even an occlusion.

B. Android Decompilers

1) Java Decompiler: Java IDEs such as IntelliJ and Eclipse include built-in decompilers to help developers analyze the third-party classes for which the source code is unavailable. Decompilation is the process of transforming bytecode instructions into source code [2]. An ideal Jdecompiler can transform all inputs into source code, and this decompiled code can be recompiled with a Java compiler and behaves the same as the original program. However, previous studies comparing Java decompilers found that this ideal Java decompiler does not exist because of the irreversible data loss during compilation. A decompiler's capacity to produce faithful retranscription of the original code is evaluated by: Syntactic correctness: when the produced decompiled code is recompiled with a java compiler without producing any error, Syntactic distortion: the minimum number of atomic edits required to transform the abstract syntax tree (AST) of the original code into the decompiled version, Semantic equivalence to modulo inputs: when the decompiled program passes the set of tests from the original test suite, and Deceptive decompilation: when the decompiler output is syntactically correct but not semantically equivalent to original inputs [2]. In this paper, a comprehensive assessment of syntactic correctness of the decompiled code, semantic equivalence with the source, and syntactic similarity to the source, was performed by evaluating eight recent decompilers on 2041 Java classes. The decompilers under study were CFR, Dava, Fernflower, JADX, JD-Core, Jode, Krakatau, and Procyon. Each decompiler was developed for different usage and was meant to achieve different goals. For example, CFR is used for Java 1 to 14 for code compiled with javac, Procyon from Java 5 and beyond, and javac, Fernflower is embedded in IntelliJ IDE Krakatau up to Java 7 and does not support Java 8, JD-Core is the engine of JD-GUI. It supports Java 1.1.8 to Java 12.0, JADX targets dex files, and Dava produces decompiled sources in Java and does not decompile bytecode produced by any specific compiler nor from any specific language [2]. Each of the eight decompilers was evaluated according to the four characteristics to produce faithful retranscription of the original code mentioned above. For syntactic correctness, no single decompiler could produce syntactically correct sources for more than 85.7% of class files in the dataset they used. This implies that the decompilation of Java bytecode cannot be blindly applied and requires manual effort. For semantic equivalence and equivalence to sources modulo inputs, five decompilers generate equivalent code for more than 50% classes [2]. Then, they isolated a subset of 157 java classes that no decompiler can handle correctly and merged the results with several incorrect decompiled sources. After that, they merged the results of the incorrect decompilers to produce a version that can be recompiled through a process known as Meta-decompilation. Meta-decompilation is a new approach for decompilation that leverages the natural diversity of existing decompilers by merging the results of different compilers, and it can provide decompiled sources for classes that no decompiler in isolation can handle. They named their meta-decompiler Arlecchino [2]. This paper's main takeaway is that even the highest-ranking decompiler in their study produces correct output for 84% of classes of the dataset used and 78% equivalent modulo input. Even their new tool Arlechino can produce semantic equivalence modulo inputs sources for 37.6% of classes that were not successfully decompiled by the other eight decompilers [2].

2) Kerberoid: A Practical Android App Decompilation System with Multiple Decompilers: One of the most notable attempts dedicated towards Android decompilers development is "Kerberoid" [3]. Kerberoid relies on the meta-decompilation approach to integrate different outputs generated from different decompilers so that it achieves higher accuracy and coverage of the decompiled code [3]. The integration happens by classifying the decompilers' output and automatically selecting the best partial result from each. Kerberoid does this process in 4 main steps. The first step (Collector) runs the target executable (the input APK file) on multiple decompilers, namely CFR, JD Project, and Jadx, and structure these outputs so that they are passed to the next stage (Parser). Different code blocks such as functions, variables, and classes are classified during the second stage (Parser) to be passed to the third stage (Integrator). Overlapping between different code blocks generated from different decompilers is handled during the third phase of the process by comparing the code blocks from different decompilers against each other. Finally, the output from the integrator is passed on to the fourth and final stage of Kerberoid. The final stage (Selector) is based on a machine learning classifier trained to select the best code block among multiple potential code blocks produced from multiple decompilers for the same function. Using this technique, Kerberoid outperformed existing Android apps decompilers when compared with them. It managed to recover 75% of the overall functions as successfully recovered 75% of the applications in the test set [3].

3) DexFus: An Android Obfuscation Technique Based on Dalvik Bytecode Translation: This paper describes how obfuscation increases the difficulty of reverse analysis of android applications without changing the semantics of the original code. It also describes how current android obfuscation techniques primarily concentrate on Dalvik byte code obfuscation as the byte code, in this case, contains much semantic information, and obfuscation will not hinder decompilation that much [4]. As mentioned, existing android obfuscation tools focus on Dalvik bytecode obfuscation, including encrypting the original DEX file and migrating x86 architecture techniques. The decrypted DEX file will be loaded into memory at runtime and obtained via a memory dumping attack. Also, the Dalvik bytecode contains too much analytical data, making it easier to analyze than assembly code [4]. This paper then introduces three standard android obfuscation systems: Proguard, Android Shelling, Compile-Time Code Virtualization, and Migrate x86. Proguard is integrated into Android Studio and provides minimal protection against reverse engineering by obfuscating names of classes, fields, and methods. Android Shelling works at the DEX obfuscation level. It replaces the origin Dex file with a stub Dex file, then encrypts the origin and hides it in the assets directory. Compile-time code virtualization uses automatic tools to transfer code virtualization from DEX level to native level at compile time. The project contains two components, the pre-compilation component for improving the performance and the compile-time virtualization component that automatically translates Dalvik bytecode to LLVMIR. This translates Dalvik bytecode into VM instructions instead of raw native code and splits the method into small ones, sacrificing few efficiencies. Finally, Migrate x86 techniques like control flow flattening and instructions substitution to Dalvik byte code. Their work increases the cost of reverse engineering. However, Dalvik bytecode, which can be disassembled back into a smali file, is easier to read than assembly language. Attackers may not spend too much time understanding the obfuscated Dalvik bytecode [4]. Then, the paper's authors introduce their solution, a proposed system that translates essential methods in the origin Dalvik bytecode to C code and obfuscates it to perform an expressive obfuscation. This tool is called Dexfus. First, dexfus uses Apktool to decompile Dex files in Android applications, then to a C code translator. Then, it hits the JNI call strings and UTF8 strings in the methods and encrypts and replaces them with function calls that decrypt strings at execution time. DexFus replaces Hot DVM methods with C methods if it can be translated to reduce JNI call consumption. It will then insert the code of loading compiled native libraries automatically and use apktool to repack the modified Dex files and copied native libraries together to an obfuscated APK file. After translating from Dalvik bytecode to C, the code still contains plain-text strings such as JNI methods which are then encrypted by the tool. After encryption, the original plain text string will not appear in the dynamic link library and will be replaced by a function call in GetResource (StringId) [4]. Finally, the paper is concluded by describing how android applications face severe threats like cracking and repackaging and how Dexfus applies obfuscation on translated C code, providing a higher level of protection than obfuscating the original Dalvik bytecode [4].

C. Neural Decompilers

1) Using recurrent neural networks for decompilation: One of the very early attempts to generate good results using ML in decompilations systems was introduced by Katz et al. for recovering C source codes from binary snippets using encoder-decoder recurrent neural networks model, which acts as a decompiler [5]. The paper's authors trained the model to recognize different patterns and properties that exist in human-written source codes to generate a similar result when decompiling different binary files. Their direct approach depends on an adaptation of a natural language translation RNN model to translate between different programming language representations using an RNN-based encoder-decoder translation system [5]. In specific, a sequence-to-sequence-based model is used in [5] to process the input sequence using an encoder RNN, then a hidden state, contained in the model, is used to summarize the entire input sequence. The summarized output of that hidden model is then fed to a decoder RNN which eventually creates the output. A significant advantage of the proposed platform is its easy extensibility to new programming languages with enough data fed to train the model on that new language. Authors of [5], however, focused only on recovering C source codes compiled at optimization level zero, which poses a validity threat that their technique might not be usable when decompiling real-world projects since most of these projects do not require user zero level optimization. In addition, one of the main limitations of their approach is that it only works for small snippets of binary machine code and fails to recover longer ones making this approach inapplicable to real-world code projects due to their lengthy code profiles [5]. Besides these limitations, the Seq2Seq model used in [5] was found to be not suitable for decompilation problems mainly due to the difference in formats between natural languages and PL, which was not accounted for when adapting the model to decompilation problems.

2) Coda: An End-to-End Neural Program Decompiler: To address the challenges and limitations highlighted in [6], in 2019, Fu et al. introduced "Coda" framework to be the first end-to-end neural-based framework for code decompilation [6]. To address decompilation, the authors divided the decompilation process into two main phases (code sketch generation and iterative error correction) and tackled each phase separately. For code sketch generation (first phase of the solution), the authors of [6] use an instruction type-aware encoder and an abstract syntax tree (AST) decoder supported by an attention feeding mechanism to convert between the input binary and high-level programming language. The code sketch generation phase works as follows: Different types of instructions, including memory, arithmetic, and branch instructions, serve as the inputs to the encoder, which makes use of e N-ary Tree-LSTM models to encode these different instructions types. A dedicated LSTM model is designated to handle each statement of the input program according to its instruction type. To address the adaptation problems encountered when using natural languages models on programming languages, Coda's abstract syntax tree (AST) is generated according to a tree decoder as it offers multiple advantages and some solutions to the limitations highlighted in [5]. Using a tree decoder and terminal node representation automatically preserves code statement boundary and facilitates syntax restriction verification. In addition, the connections between the tree nodes better highlight the dependency constraints in the program being analyzed. It also makes it easier to mitigate the error propagation problem during the code generation stage.

In contrast, to [5], the RNNs model presented in [6] dedicates a separate RNN for each statement type which results in preserving the modular property of the programs (preserving statement boundaries) as well as capturing the control and data dependencies of the program (the connections between the different states of the RNN.

The second stage (Iterative Error Correction) of the proposed method focuses on solving the prediction errors that are unintentionally generated during the code sketch phase. To tackle this problem, authors of [6] develop an iterative error predictor that works on the output of the autoencoder-decoder of the first stage to increase the accuracy of the translation. The output of this predictor indicates the location and error type information in the code sketch [6]. The error correction machine takes this output and relies on confidence scores generated by the EP to prioritize correction strategies that have the potential to increase the accuracy of the generated high-level code [6].

Although the method proposed in [6] outperforms earlier attempts to deal with decompilation as a translation problem, the method also suffers from considerable limitations that affect its performance and applicability to real-life decompilation problems. These limitations include the decrease in its performance when working on low-level input code but from complicated ISA such as x86-64 [6]. In addition, the Coda framework does not provide reliable recovery and does not work correctly when complex data structures, control graphs, static libraries are used in the input program [6]. Moreover, the framework generates credible recovery results only with short programs with the average code length of 45, and 60 [6].

3) Toward Neural Decompilation: Toward Neural Decompilation tackles the limitation of using recurrent neural networks for decompilation, which is the translation between natural and programming language. Discovering vulnerabilities and malware analysis begins by comprehending the low-level code comprising the program. Although most Reverse Engineers and Malware Analysts go through this process manually by reverse-engineering the program is a slow, time-consuming, and tedious task. The problem here lies in manually going through every line to try and understand what a program does and how it is done. Hence, decompilation can significantly improve this manual process by automatically translating the executable binary code to a better visual and readable higher-level code. Not only is valuable decompilation for invulnerability and security analysis, but also in the easiness of portability to other architectures or operating systems since we are dealing with the source code itself.

Many existing decompilers vary in their decompilation process. For instance, some rely on pattern matching to clarify the relations between the low-level and high-level structures. However, the failure rate of this method is high when used on sophisticated code and advanced statements such as the goto. Semantically equivalence may be achieved compared to the original binary file, yet it is unreadable and non-efficient. Another decompiler method worth mentioning is those based on neural machine translation (NMT) due to their significant improvement and results in regards to binary to source code translation. As mentioned by the authors of [7], these decompilers still have their problems and constraints. For example, a decompiler that used RNN for decompilation had difficulties relating between the translated programming languages and our natural language, thus leading to poor results. Hence, the code they generate often is not recompiled or equivalent to the source code.

An automatic neural decompiler is a two-phased approach that targets some of the previously mentioned problems above. The first stage tries to generate a template code snippet with an equivalent structure, including a computation to the input file. The second stage would be filling the template code snippet with the program's values to finalize the decompilation process. Instead of applying the methods of the existing NMT decompilers that work directly on binary files without strong natural language knowledge, an Automatic neural decompiler applies the first augmentation with programming-languages knowledge (domain-knowledge). The NMT translation will be more readable and more straightforward code than the original NMT decompilers thorough domain knowledge. Also, the Automatic neural decompiler incorporates techniques used in Natural Language Processing (NLP) to better structure the translated programming language. The second phase of the Automatic neural decompiler receives the template code snippet as input to find the correct values to represent actual code from the template code snippet released from the NMT translation [7]. The second phase verifies that the generated values are correct and relevant. If not, they are replaced using the delexicalization practices learned through NLP. Hence, the process starts from the assembly code to the NMT model, the snippet code result, and the final NLP checking step.

Automatic neural decompiler primary purpose is to decompile off-the-shelf compilers that use optimization techniques in the compilation process. They do not aim to handle handwritten assembly, nor do they try to outperform existing decompilers. The major limitation they face is thetas the length of an input increases; there is a higher chance that the decompiled code would fail [7]. As the fields of NMT evolve to handle long inputs better, so would the resulted output. Finally, the decompilation testing was implemented on LLVM IR and x86 assembly to C.

4) N-Bref: A high-fidelity Decompiler Exploiting Programming structures: This is the current state of the art neural to neural system for decompiling binaries with an `x86` target back to high level `C`[8]. Essentially, the paper proposes two models that support each other. One model creates an abstract syntax tree (AST) from the disassembly and another that extracts a graph neural network from the assembly, to create a graph of how the low level registers interact with each other. This way we have a way to inspect how the small scale variables interactions as well as the larger scope and complexity of the application as a whole.

The low level code, in this case the `x86` assembly, is analyzed instruction by instruction to decompose it into its elemental components, the opcode and the registers it is manipulating identifying whether they act as sources or destinations. We can then create a graph of these relations how the instructions flow, From this we also try to extrapolate small scale data flows, such as adding directional edges to convey the equivalence of certain nodes and registers to make it easier for the model to recognize the flow of data. All this is to have an idea of what the paper likes to call ”Control Flow”.

The heart of the system is the structural transformer. First it takes the abstract syntax tree and encodes it along with the graph of the assembly instructions to predict what node could be missing from the AST. Then this new AST is used for another iteration of predicting the next missing node. This leads to the AST being developed breadth first. This process concludes when the predicted node leads to termination.

To address the lack of determinism in the structures expressed in abstract syntax tress, all unary operations are converted to conventional mathematical expressions and all while loops are made into for loops. This however does hurt the final accuracy since the original code is likely to have been written with a variety of styles from different users. Various hyper parameters are used to fine tune the model, namely they influence how complex the code to be analyzed is expected to be and compensates accordingly. The dataset this papers used were mostly unoptimized. Their approach was not mature enough to extract the data types of optimized and stripped data structures.

This system can be thought of as an encoder decoder transformer network where both the assembly and the abstract syntax tree are encoded to decode into a more elaborate AST.

The actual architecture of the system is a bit similar to natural language processors. Both extracted structures are encoded fed into self attention layers and decoded into the new AST node.

However the inner workings of the model also involves more advanced techniques such as memory augmentation and graph augmentation. Graph augmentation is implemented by making the matrices graphs between attention layers to allow more complicated inferences to be performed.

5) Neutron: an attention-based neural decompiler: Neutron is a neural decompilation framework that uses previous Neural Machine Translation (NMT) models to overcome the bottleneck of decompilation technologies that rely on experts to write rules resulting in low scalability, development difficulties and long cycles. Neutron's framework is phased into three stages: Code Preprocessing, Neural Translation and Function Reconstruction.

The main goal in the first phase (code pre-processing) is to standardize the PL to assist the model in learning the conversion rules between the tokens of the high-level language to that of the lower-level representation of the language. In order to minimize the complexity of the model, the binary code is disassembled; the assembly language is then used as the target low-level PL. This helps reduce the model difficulty learning conversion rules because it contains richer semantic and structural information. The standardization module focuses on processing the identifiers, numbers, etc. in the PL code, which facilitates model training and translation to better learn the conversion rules between low-level and high-level PLs.

The second stage develops a neural decompilation model that performs the actual recovery of an equivalent high-level C from the low-level target. Afterwards, Neutron trains the model by the name of AsmTran to translate low-level PL into a high-level PL while keeping their functionally similar. This model is based on LSTM-Seq2Seq-attention (Luong et al. 2015) architecture. The AsmTran model is mainly divided into two sub-models. The first sub-model is a text classification model aiming to fine-grain code segmentation for low-level PL based on basic operations. The second sub-model is an NMT model which takes each basic operation of the target low-level PL as input and outputs its corresponding high-level PL. It is worth mentioning that the authors of [] introduce the iterative error correction (EC) mechanism in both sub-models in the training phrase. The prediction errors in the two sub-models' output are fed back to the sub-model itself to improve the AsmTran's performance through the manual definition of judgment and EC rules.

In Neutron's final phase, Function Reconstruction works with the aim of reconstructing the missing dependencies from the previous stage. This includes data flow recovery, control flow recovery, as well as parameters and return value recovery, and it works by the manual definition of rules as it aids in the reconstruction process. Neutron is implemented on the base of the attention-based NMt mechanism in the tensor2tensor framework. The results show that Neutron achieves an average accuracy of 96.96% on three real-word projects and three different tasks.

Neutron's contribution is mainly its introduction to a new technique that has high applicability and redability as measured by benchmarks in varying levels of complexity, and it offers a new understanding of the feasibility of applying NMT model that works on natural language to Pls. Similar to natural language translation, the decompilation of low-level PL to high-level PL can also be seen as a translation problem between two natural languages. The results on three real-world projects and three different tasks show that Neutron's accuracy can reach 96.96% on average. Compared with using the LSTM-Seq2Seq-attention model (Luong et al. 2015), our approach achieves 36.11% higher accuracy on average, which reflects that the attention- based NMT method fails to learn the conversion rules between PL pairs effectively. Besides, the proposed approach acheived 74.71% improvement, on average, compared to using other LSTM models. Even though, the results shown by Neutron are promising, it still exhibits a number of limitations. Firstly, it does not effectively restore the semantics of target low-level PL. It also performs poorly in optimized code primarily due to the adoption of the slicing mechanism. The final limitation is Neutron's inability to recover user-defined data types such as classes and structures which hinders the readability of the decompiled high-level PL.

D. Evolutionary Decompiler

1) Evolving an Exact Decompiler: This paper [9] takes a rather unique approach where it considers the compiler itself to be black box that produces binary files to arbitrary text inputs and searches for a corresponding pattern. This approach guarantees that the resulting code generated will be syntactically and functionally correct. They call this technique Binary Equivalent Decompilation (BED).

BED starts with random code samples from its extensive database of code then and compiles it. with Byte-similarity is the comparison metric, it compares this compiled target with the binary it is trying to decompile. According to that, it mutates the code snippets and consults the database to create a version that has the potential to be more similar to the original binary. We keep iterating through this process until the result of BED is byte for byte identical to target binary. This can be considered as an evolutionary algorithm.

Although the results of the paper greatly improved on readability and correctness measures compared to other existing decompilers, a number of limitations hindered the applicability of this framework to real-world applications. The three biggest problems with this technique is how certain compilers can inheritley create different binary from the same code and optimization level, meaning even if BED happened to achieve identical code, it would still reject the binary. Another major problem is its lack of scaling for complex and large code bases; it would take an exponential amount of time for it to decompile as the binary grows. Finally the last fundamental problem is if the code being predicted is not a snippet from the dataset then it would be very difficult to properly decompile it and it is too unlikely for mutation to happen upon the right changes. This culminates in this solution working well on very small examples but having problems generalizing especially at scale.

IV. Discussion

Each of the surveyed papers inspired us in some way till we reached a final version for the system architecture for our project.

A. Binary Enrichment

For the first stage in our pipeline integration. As discussed in the previous sections, there is a lot of information lost or omitted during the compilation process as we reach the binary file. As a result, we adopt DEBIN's methodology to predict the symbols omitted in the compilation stage with a high accuracy. Moreover, ARM is prone to the problems previously mentioned in the x86 disassembling. Therefore, to avoid such problems, we are modifying the disassembling stage adopted by DEBIN with the approach proposed in probabilistic disassembly paper [10].

B. Neural Decompilation

This survey served to give us insight into how to create the all important high level code we are targeting. Inital attempts at neural decompilation attempted to use recurrent neural networks

References

[1] "Debin — proceedings of the 2018 ACM SIGSAC conference on computer and communications security." (), [Online]. Available: https://dl.acm.org/doi/10.1145/3243734.3243866 (visited on 09/24/2021).

[2] N. Harrand, C. Soto-Valero, M. Monperrus, and B. Baudry, "Java decompiler diversity and its application to meta-decompilation," *Journal of Systems and Software*, vol. 168, p. 110 645, Oct. 1, 2020, ISSN: 0164-1212. DOI: 10 . 1016 / j . jss . 2020 . 110645. [Online]. Available: https : / / www . sciencedirect . com / science / article/pii/S0164121220301151 (visited on 09/24/2021).

[3] H. Jang, B. Jin, S. Hyun, and H. Kim, "Kerberoid: A practical android app decompilation system with multiple decompilers," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19, New York, NY, USA: Association for Computing Machinery, Nov. 6, 2019, pp. 2557–2559, ISBN: 978-1-4503-6747-9. DOI: 10 . 1145/3319535.3363255. [Online]. Available: https://doi.org/10.1145/3319535.3363255 (visited on 09/24/2021).

[4] H. Naitian, M. Xingkong, F. Lin, L. Bo, and L. Tong, "DexFus: An android obfuscation technique based on dalvik bytecode translation," 2020. DOI: 10.1007/978-981-15-9739-8_32.

[5] D. S. Katz, J. Ruchti, and E. Schulte, "Using recurrent neural networks for decompilation," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Mar. 2018, pp. 346–356. DOI: 10.1109/SANER.2018.8330222.

[6] C. Fu, H. Chen, H. Liu, X. Chen, Y. Tian, F. Koushanfar, and J. Zhao, "A neural-based program decompiler," *arXiv:1906.12029 [cs]*, Jun. 27, 2019. arXiv: 1906 . 12029. [Online]. Available: http://arxiv.org/abs/1906. 12029 (visited on 09/28/2021).

[7] O. Katz, Y. Olshaker, Y. Goldberg, and E. Yahav, "Towards neural decompilation," *arXiv:1905.08325 [cs]*, May 20, 2019. arXiv: 1905.08325. [Online]. Available: http://arxiv.org/abs/1905.08325 (visited on 09/28/2021).

[8] C. Fu, K. Yang, X. Chen, Y. Tian, and J. Zhao, "N-bref : A high-fidelity decompiler exploiting programming structures," Sep. 28, 2020. [Online]. Available: https: //openreview.net/forum?id=6GkL6qM3LV (visited on 09/24/2021).

[9] E. Schulte, J. Ruchti, M. Noonan, D. Ciarletta, and A. Loginov, "Evolving exact decompilation," Jan. 1, 2018. DOI: 10.14722/bar.2018.23008.

[10] K. Miller, Y. Kwon, Y. Sun, Z. Zhang, X. Zhang, and Z. Lin, "Probabilistic disassembly," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, ISSN: 1558-1225, May 2019, pp. 1187–1198. DOI: 10.1109/ICSE.2019.00121.