Proposal

September 26, 2021

1 Overview

Although reverse engineering a binary program has a rather 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 which introduces many limitations to traditional decompilers including poor scalability and development. Since the purpose of decompilation is crucial to multiple cybersecurity domains such as malware analysis and vulnerability discovery, the more this processed is enhanced, the more we can offer faster results to analysts and reverse engineers.

One of the core conceptual problems with decompilation is that although you can deterministically compile code into assembly, the reverse is untrue; there are countless forms that code can take that produces 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 would have written it.

Furthermore, x86 architecture has been given plenty of the focus in recent papers so this leads our attention to seeing what would happen if we use these techiniques on other architectures, namely android and its Java virtual machine. We believe that the more expressive nature of java bytecode has the potential to open up for us better prediction along with easier extrapolation of under lying structure due to java's more strict object oriented paradigm. Also this is to provide tools to comabt the current rise in security concerns on the mobile platform.

Beyond simply migrating well established solutins to bytecode, we also propose a 3 phase solution to addressing the decompilation process, each phase leveraging a different approach to extracting information from the cryptic binary. The first part of the pipeline deals with the disassembly phase of the decompialtion process. Disassembly is a crucial step during the decompilation process since the rest of the phases are heavily dependent on the output of the disassembler. However, the complexity of optimizations and other compilation techniques induce

many uncertainties that the disassembler needs to handle during the process. The output of the currenlty used disassemblers have false postives and false negatives which negatively reflects on the decompiled code. Thus, in order to improve the accuracy of the disassembler output, probabilities are used to model the uncertainities that are induced during the compilation phase. The assigned probabilities basically reflect the likelihood of the corresponding instruction address to have false positive or false negative instruction. Probabilistic inference is used afterwards for disassembling the target binary file according to the probabilities assigned to each address in the code section. The second phase relies on ideas proposed by debin to find data within the disassembly itself without actually starting the transition back to a high level language. n Here, we attempt to predict what symbols were omitted during the optimization process; think of it as taking a optimized binary and extrapolating from it a debug binary complete with debugging symbols. This would allow us more insight where creating the abstract syntax tree in the last phase to produce the final high level langauge prediction.

Ofcourse, we will not only fully integrate these phases into a single cohesive tool stack and migrate them to another architecture but also to improve on each one individually.

Machine learning techniques offer an opportunity to dramatically automate this process. Thus, machine learning based decompilation has been explored in a number of research projects before. However all of these projects target the recovery of C code running on x86 architecture. Although decompiling android applications has a lot of problems still that hinder the accurate recovery of the source code, none of the projects that got implemented tackle this issue. Besides the typical compilation problems that are present in any decompiler(e.g. syntatic distortion, semantic incorrectness, and major difficulty with code readability), android apps decompilers show a considerable failure rate in recovering programs as well as a major 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 original source code. Therefore, this project introduces a new platform for credible android apps decompilation that adopts major successful models that dramatically enhanced the performance of desktop

2 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 not available. Decompilation is the process of transforming bytecode instructions into source code (Citation). An ideal Jdecompiler is one that can transform all inputs into source code and this decompiled code can be both recompiled with a Java com-

piler, and behaves the same as the original program. However, previous studies that compared Java decompilers (Citation) found that this ideal Java decompiler does not exist because of the irreversible data loss that happens during compilation. (Citation all definitions) A decompiler's capacity to produce faithful retranscription of the original code is evaluated by: Syntactic correctness: when the produced decompiled code is recompilabe 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. In the paper (Java Decompiler Paper Citation/Name), a comprehensive assessment of: syntactic correctness of the decompiled code, semantic equivalence with the original source, and syntactic similarity to the original source, was performed by evaluating eight recent decompilers on 2041 Java classes. The set of 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 and 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. Each of the eight decompilers was evaluated according to the four characteristics to produce faithful retranscription of the original code that were mentioned above. For syntactic correctness, no single decompiler was able to produce syntactically correct sources for more than 85.7% of class files in the dataset they used. This implies that 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. 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 is able to provide decompiled sources for classes that no decompiler in isolation can handle. They named their meta-decompiler Arlecchino. 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.

3 N-Bref

This is the current state of the art neural to neural system for decompiling binaries with an `x86` target back to high level `C`. Essentially, the paper proposes 2 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 aswell 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 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 companstes accordingly. The dataset this papers used were mostly unoptimized. Their approach was not mature enough to extract the datatypes 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 acrchitecture of the system is a bit similar to natrual langugae processors. Both extracted structures are encoded fed into self attention layers and decoded into the new AST node.

However the inerworkings of the model also involves more advanced techinques such as memory augmentation and graph augmentation. Graph augmentation is implemented by making the matrecies graphs between attention layers to allow more complicated intferences to be performed.

4 Evolving an Exact Decompiler

5 DEBIN

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

Basically, DEBIN follows five main steps to produce the improved and enhanced binary file. Firstly, the stripped binary file is taken as input and then lifted from assembly code into an intermediate Binary Analysis Platform (BAP-IR). The reason for this transition is have a high level form of the sementacis available, generalize the syntax among the three achritectures DEBIN targets, and a better understanding of the logic instructions. BAB-IR preserves the raw instructions semantics, showes explicitly the operations on machine states, and also recognizes the function boundaries via its ByteWeight component to obtain the code elements. The next step would be to extract two type of data from the intermediate, which are the known and unknown elements. The unkown elements are the ones that have been lost in the compilation process due to optimization's stripping. In contrast, the known elements are the already present properties in the binary code and no need to infering any its information.Examples of known elments 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 dependacy graph is built and formed with nodes and their relationship with each other as edges. Moving on to the next crucial step, DEBIN starts to infer the unknown elements through the pobablistic method, Maximum a Posterior (MAP).The relationship between elements is represented in the form (a,b,rel). These three words represents node `A` that is 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 on the unkown elements, an updated binary file is released as output and ready with its improved debug information.

DEBIN dataset consisted of 9000 non-stripped binary excutables that were originally written in pure C language, 3000 for each of their targeted architectures. They gave 2700 of the binary files as training data and left the remaining 300 binary files for testing and prediction purposes. Also, DEBIN uses a binary classifier to better evaluate 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 major key 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 initial use of obfusication or human-based written assembly.

6 Toward Neural Decompilation

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, it is a slow, time consuming, and tedious task. The problem here lies in the act of manually going through each and every line to try and understand what a program does and how it is done. Hence, decompilation can greatly improve this manual process by automatically translating the excutalbe binary code to a better visual and readable higher-level code. Not only is Decompilation useful in vulnerability 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, there are those who 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 in comparison to the orginal binary file, yet it's unreadable and non-efficient. Another decompiler method that is worth mentioning is those that are based on neural machine translation (NMT) due to their significant improvement and results in regards to binary to source code translation. Still these decompilers have their own problems and constraints. For example, a decompiler that used RNN for decompialtion had difficulties relating between the transalted programming languages and our natural language, thus leading to poor results. Hence, the code they generate often is not recompilabe or is not equivalent to the original source code.

Automatic neural deocmopiler is a two phased approach that aims at targeting some of the previously mentioned problems above. The first stage tries to generate a template code snippet that has an equivalent structure including computation to the input file. The second stage would be filling the template code snippet with the programs 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 langauge knowledge, Automatic nueral decomopiler applies first augmentation with programming-languages knowledge (domain-knowledge). Through domain-knowledge, the NMT translation will by more readable and simpler code in contrast to the original NMT decompilers. Also, Automatic neural deocmopiler incoperates techniques used in Natural Language Processing (NLP) to better structure the translated programming language.The second phase of the Automatic nueral deocmopiler recieves the template code snippet as input to find the right values for represent actual code from the template code snippet released from the NMT translation. The second phase verifies that the generated values are correct and relevant, if not then they are replaced using the delexicalization practices learned through NLP. Hence, the process starts from the assembly code, to the NMT mode, to the snippet code result, to the final NLP checking step.

Automatic neural deocmopiler main purpose is decompile off-the-shelf compilers that use optimization techiniques in the compilation process. They do not aim to handle hand written assembly, nor do they try to outperform existing decompilers. Mjor limitation they face is thet as the length of an input increases, there is a higher chance that the decompiled code would fail. As the fields of NMT evolves to better handle long inputs, so would the resulted output. Finally, the decompilation testing was implemented on LLVM IR and x86 assembly to C.

7 Contact Details

contact us through solomspd@aucegypt.edu