Proposal

September 23, 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 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`. It is a neural to neural system that means to Essentially, the paper proposes 2 models that support each other. One model creates an abstract syntax tree from the disassembly and another that extracts a graph neural netword 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.

3 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.

4 Contact Details

contact us through solomspd@aucegypt.edu