Litrature Review

September 22, 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 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 it's Java virtual machine. We believe that the more expressive nature of java byte code 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 byte code, 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 phase involves The second phase relies ideas proposed by debin to find data within the disassembly itself without actually starting the transition back to a high level language. 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

In this

2 N-Bref

This is the current state of the art for decompiling binaries with an `x86` target.