Isaac Yuheng Tian
Iytian

CSE 503 Assignment 1

1.

In languages that require manual memory management (i.e. C, C++) it is possible to have pointers to memory addresses that are valid addresses on the stack but are invalid in runtime semantics because they were allocated in a frame that was then popped, and then the stack extended past that address. A trivial example of this would be returning the address of a variable that was declared on the stack in a function call and then dereferencing the variable later. This is especially easy to do with arrays for people who are used to programming in Java or Python where lists are returned by value. In C, if an array is returned via its pointer, the next time it is referenced in a different frame it will either be invalid, or it will fail silently because another stack frame was pushed on top of the calling frame, making the address legal but the value meaningless. I had this problem in undergraduate computer graphics course during debugging: my partner did not understand memory semantics as well and it took many hours to find that he was returning pointers to values in an array declared in the stack frame of called functions. These values then get overridden by later stack frames but no errors are thrown during runtime. The easiest way to fix this is by teaching people to never return pointers to stack addresses from functions. It's not easy to catch in a compiler or during execution because often these errors result in functions referencing a variable on their own stack address, and this is obviously a legal and necessary operation most of the time that is hard to distinguish from the silent failures. We could just make compilers throw warnings or exceptions whenever pointers are returned that are not heap addresses.

2.

Documentation has always been an issue with any kind of code base, both code that I've written, written with other people, or just code I refer to that I find on other people's githubs. (Usually research code or toolkits). Usually sufficient documentation is considered to be just enough to understand functions or entire files as black boxes: summary of use, inputs, outputs. As soon as you have to understand how anything happens inside the source blocks, whether for

research purposes or for debugging, it becomes very difficult. Source code explanations line by line are usually rare and if they happen in the raw text it has the added tradeoff of making the file longer and less readable. But their absence means a human being has to interpret the source code directly without the aid of a natural language. Automatic code annotation is very difficult because some semblance of natural language is necessary to convey the meaning of source without requiring a human to perform their own static analysis. A very basic annotation system can be constructed (or probably already exists in some IDE) that will give function summaries. All inputs are listed, all declarations are explicitly highlighted, and a basic summary of how these affect the return value (if there is one) is generated in simple syntax. (i.e. A is ADDED to B and DIVIDED by an INT that is fixed to the value 3). For void type returns, a summary of how inputs are modified in place can be provided. Even though this kind of annotation doesn't explain in fully natural language what the underlying purpose or motivation of the code was, paired with the summary provided before function declaration this should aid in reasoning about code in a more natural way.

3.

Manual "static" analysis is particularly annoying to do with dynamically typed languages like Python. There is no way to know what the variables are supposed to be by simply glancing at the source code and you are entirely dependent on comments or by completely tracing a call stack to its origin and notating yourself the possible values of the variable. This makes maintenance very annoying as you have to now document all the intended variable types if you intend to understand this code in 2 years. Whenever I read research code online that is written in Python, it always becomes a long backwards trace of every single variable of interest to find the first time it was used, possibly many many function calls before, and check for reassignments along the way as well. I comment a lot of types into my code as a result, but this makes the file long and not very concise. This can be fixed by either a) making it standard practice to thoroughly document intended typing in languages with no static typing or b) making some kind of static analysis tool that will do the call tracing for you and list either an annotated type that the author notates in the source on declaration or all possible types as constrained by all modifying function calls along the way.