Advanced Topics in Software Engineering: Memory Safety, QA Tools

Giovanni Magoga
MatrNr: 21855119
Supervisor: M.Sc. Alexander Trautsch

October 24, 2020

Contents

1 Memory Safety in Programming Languages

Following is the summary on the topic presented on 23.01.2019 by Robin Hundt.

1.1 Introduction

Memory safety issues had already been formalized as early as 1988 after the widespread damage caused by the infamous Morris Worm, notable enough to subsequently gain itself an exclusive spot in popular culture. Nevertheless, it's with modern languages that memory related bugs have gained undisputed relevance, primarily due to their ever wider spectra of users and use cases. The high abstraction level of such languages, contributes to the already troublesome existence at run-time and difficulty of replication of memory bugs, ultimately characterizing their undetectability to a major extent.

1.2 What is memory safety?

Accordingly to its formal definition: "Memory safety is a property that ensures that all memory accesses adhere to the semantics defined by the source programming language.", which, in practice, implies the two following sub-classes of memory safety.

1.2.1 Spatial safety

Spatial safety ensures that all memory references are within their own pointer's valid objects whenever one wishes to de-reference them. Those bounds are defined when the object is allocated, inherited by successive pointers and must contain the result of any future pointer aritmetic operation. De-referencing a pointer whose associated memory address is i.e. outside the boundaries of its corresponding object will then result in a spatial memory safety error and undefined behavior.

1.2.2 Temporal safety

Temporal memory safety ensures that memory references are valid whenever one wishes to de-reference a pointer to their address, and thus that the pointed-to object is the same as when the pointer was created. De-referencing a pointer whose associated memory address has been i.e. freed will then result in a temporal memory safety error and undefined behavior.

1.3 Troubleshooting techniques

Theoretical approaches to memory safety have been researched at both run-time and compile-time. In the first case, most of the work is left to a garbage collector handled by the language implementation, consuming potentially valuable resources. In the second case, the end developer is expected to be responsible for antecedently implementing those practices according to the specific scenario.

1.3.1 Run time

In its most general description, any memory reference should be coupled with meta-data providing and unique identifier with syntactic and semantic valence, namely value, base and size of the pointer. Before any pointer is de-referenced, its value is checked to be inside the [base, base + size] dimensions, terminating the program if the check should fail.

1.3.2 Compile time

Robert Millner demonstrates how "well-typed programs cannot 'go wrong'", meaning they won't exhibit any kind of undefined behavior. (i.e. cannot crash, write to random memory locations, or read past the end of a buffer). This requirement is although hard to enforce in practice, as a syntactically well typed program could still compile while not meeting semantic typing requirements according to deeper sections of the specification, likely overlooked by the average developer.

1.4 Implementation in Rust

Rust is an open source, system-oriented programming language which focuses on memory safety without significantly hampering performance. Aimed at suppressing the numerous issues encountered in over-relied upon, type-safe languages like C++, it was first released in 2010 and its development been endorsed by the Mozilla foundation ever since.

1.4.1 Ownership

A central class of issues addressed by any language striving for memory safety are those found under the umbrella term of 'memory leaks', such as heap exhaustion, double free, invalid free, etc. Traditionally, most of these problems are addressed through a garbage collector executed at run-time, with notable examples being Java and Objective-C. This cumbersome requirement is eliminated through a simple set of rules verified at compile-time, reported down here accordingly to the official documentation:

- Each value in Rust has a variable that's called its owner.
- There can only be one owner at a time.
- When the owner goes out of scope, the value will be dropped.

1.4.2 Borrowing

Another central issue tackled by Rust are access errors, notably present in multithreaded programs and hard to debug. In this case, safety is achieved with a reverse look at the definition of 'data race': "[...] when two or more pointers access the same memory location at the same time, where at least one of them is writing, and the operations are not synchronized", resulting in a simple set of rules in this instance as well:

- There can be one or more references to a resource
- There can be one and only one mutable reference to said resource

2 The role of QA tools in software engineering practice

Following is the summary on the topic presented on 30.01.2019 by Gulzaib Amjad.

2.1 Introduction

Thanks to the static nature of common issues found in source code, analytical tools can be distributed and executed on the developer's own machine with neglegible overhead.

The most widely used analyzers go under the name of linters, whose purpose is enforcing syntactic adherence to the specification of programming languages, best practices and/or company guidelines. Sanitizers are also frequently employed for preventing malicious behavior in user inputs, making them a popular choice expecially in security sensitive applications. More sophisticated techniques rely on formal methods, which require the soundness of certain mathematical properties in the program according to its specification.

A major advantage of static analysis tools over their dynamic counterparts are the relatively low hardware requirements which enable real time execution.

In general, static analysis tools constitute a best-effort approach to guarantee a solid baseline to conduct further code reviews, if necessary, thus reducing the overall costs of quality assurance.

2.2 Data flow analysis

Data flow analysis is one example among the various techniques typically performed by formal method tools; its objective is tracking local definitions, undefinitions and references to variables along different paths of the data flow. The PMD tool, for instance, classifies anomalies with three labels as reported below, accordingly to its documentation:

- UR - Anomaly: There is a reference to a variable that was not defined before. This is a bug and leads to an error.
- DU - Anomaly: A recently defined variable is undefined. These anomalies may appear in normal source text.
- DD - Anomaly: A recently defined variable is redefined. This is ominous but don't have to be a bug.

2.3 Limitations

The relatively low adoption rate of static analysis tools can be imputed mainly to project dependant configuration overhead and high variance between IDEs/text editors. The development of such tools constitutes another significant hurdle to their reliability. Given the high variety of development frameworks and languages, many static analyzers are found in form of plugins which are often created and maintained by a single user to satisfy personal needs, yet they can require high degrees of logical complexity. In this scope, precautions should be set in place to account for:

- File relevance: Large and non-source files should be handled separately to avoid performance degradation in IDEs.
- Weak/strong typing: Tools should be reliable in both cases without crossing the boundaries of static analysis.
- False positive rates: Bad configurations would result in unnecessary nuisances to the developer.
- Loops: Long iterations would result in additional performance degradation.

On the user side, they require additional knowledge of yet another tool-chain component which further slows down the development cycle, and need to satisfy a set of minimum hardware requirements in order to operate reliably, which contrasts with its high variability found in real-life. Some of the illustrated issues could be overcome by standardizing each of the various families of static analyzers so as to ease integration with well established IDEs.