Verification of Object Invariants in the Presence of Unverified Code

Master's Thesis Project Description

Frederic Necker
Supervised by Dr Marco Eilers and Prof. Dr. Peter Müller

Programming Methodology Group

October 2023

1 Introduction

In the last years, many efforts have been made in the field of deductive program verification whose objective is mathematical proofs of correctness of a given program, with correctness defined as adherence to a given specification. In short, program verification aims to prove that a program does what it is supposed to do. In more precise terms, programs are annotated with preconditions, postconditions, and invariants. A program is correct if all methods, when called in states where their preconditions hold, do not abort and establish their postconditions before they return, and invariants are maintained throughout the program.

Amongst the problems that kind of analysis faces, a fairly important one is heap access and aliasing. The presence of aliasing means that a heap region or a field might be changed without ever being mentioned in the analysed code, because another alias of it is being modified. Commonly, for tools such as Viper [1] a permission system [3] is used to permit local reasoning on the heap and solve that problem. To put it simply, a given heap location's writing permissions can only be held by a single thread at a time. Thus if you hold writing permissions for two different references, you are assured that they are not aliases of the same object. One can also deduce from holding permission to an object that said object will not be modified by another thread or method being called.

But suppose a malicious agent, which we will name the adversary, is able to run some code of his own on the same system our code is running. This could happen if a library the code uses becomes compromised for example. Then this system has a fairly major problem; the adversary will ignore all restrictions that are not explicitly imposed on him, including the permission system. Suddenly we are not verifying who *has permission* to modify the data anymore, we are instead verifying who *can* modify the data. And if the adversary is able to modify a field, he can do so forever and can thus prevent us from using it correctly ever again. Thankfully, object-oriented languages like Java have given us tools such as private fields and methods to help with the situation. If they are used correctly then we can still obtain some guarantees about our code by reasoning in a smart way.

Thus, the goal of the project is to create a more robust verification system that takes into account both the new challenges and tools to give us guarantees on the functionality of code in this new environment.

2 Challenges

The setting we will be tackling will be the following: the user and the adversary both own a pointer to the same object, which we assumed the adversary obtained after the constructor was ran.

This could happen if the following code was run for example:

```java
MyClass myobject = new MyClass();
//the library badlibrary was corrupted by a malicious agent!
badlibrary.use(myobject)
//bad library opens a new thread before returning
//now both the badlibrary and ourselves have access a pointer to myobject
myobject.method();
//usage of my class which we would want to verify despite the leak.
```

The malicious code is running on the same system but on a different thread. The adversary does not have total control over the heap. But he can run arbitrary code using any objects he could get a pointer to and, quite critically, can call any public functions of objects he has a pointer to without obeying their preconditions.

To demonstrate what kind of property we would want to prove and the difficulties in doing so, here is a fairly trivial example:

```java
class Carrier{
    private int content;
    public Carrier(int con) {
        this.content = con;
    }

    public int getContent() {
        return content;
    }
    public void setContent(int newcon){
        this.content = newcon;
    }
}

class CarrierCarrier{
    private Carrier contentCarrier;
    //requires(True)
    //ensures this.contenCarrier.content == 1
    public CarrierCarrier(int con) {
        if(con == 1){
            this.contentCarrier = Carrier(con);
        } else {
            this.contentCarrier = Carrier(1);
        }
    }
    //requires True
    //ensures \result == 1
    public int getOne(){
        return contentCarrier.getContent();
    }
}
```

This example here is correct in relation to its specification. Since we assume the constructor ran its course, it established that the `Carrier contentCarrier` object contained a 1 in its content field. Since the adversary cannot call `setContent` as `contentCarrier` is private, it cannot stop `getOne` from returning 1. Should `contentCarrier` be assigned a public access modifier, then the example would no longer be correct , as the specification would become broken by the adversary calling `setContent` before our call of `getOne`, resulting it in our call to return a value different than 1.

Let us return to our first example (with `contentCarrier` set to private). Say we added the following function to `CarrierCarrier`:

```java
    public Carrier leak(){
        return contentCarier;
    }
```

Clearly the specification is incorrect again. The adversary can now call `leak` to get access to the `contentCarrier` again and break the specification.
This shows that we not only need to think of the visibility of fields and methods, but also of the reachability of objects. Even though we did not change the methods we called, the specification became incorrect as the adversary had an indirect route to an object supposed to be private. This also shows that verification will at have to consider a whole object and not only individual methods. Side effects of methods become incredibly important if the adversary can call them while a legitimate method call is in progress. If a method temporarily breaks an invariant but then reestablishes it by the end of the method call, then all the adversary has to do is to time his method call in a way that ensures that the invariant is broken when the legitimate user depends on it.

3 Approach and goals

To work on those problems, trying to guess all actions our adversary could take is not that useful. Instead our approach will be to develop new specification constructs that are resilient to our setting, and then create an automatic tool to verify program in our new setting using the new constructs.

To be more precise; our **core goals** will be:

- To define (or find in already existing work) a small subset of Java to use as the target language. It should have at least classes with fields and methods, constructors, normal integer arithmetic, loops, assignment, function calls, simple concurrency, and access modifiers.

- To specify in an exact manner the capacity of our adversary, as well as the exact setting of engagement. The adversary should at least be able to use every feature of our language, and thus create multiple threads, he should be able to call functions without respecting their specification but he should have to respect access modifiers, and thus only have access to data he obtain from either methods calls or from public fields.

- To create in that language a set of interesting code examples that we can use to test our tool and then carefully evaluate those examples by hand to give us a ground truth on which respect their specification and which are not.

- To define some specification constructs one can use to help prove that a code fragment respects specifications even when faced by the adversary we specified.

- To define an encoding of our Java subset to the Viper language using the new constructs, such that if the Viper program is verified as correct by the Viper verifier then the original Java program is correct, too.

- To implement the encoding in a tool based on Viper. Given a code fragment, and its specification, the tool should apply the corresponding Viper encoding and specification, that will then be verified by the base Viper program to finally output whether or not it could prove that the fragment does indeed satisfy the specification.

- To evaluate our technique with our examples, seeing if the tool we developed is working, and to seek potential upgrade avenues if the tool fails to attain expectations.

After that, should time allow it, our **extension goals** will be:

- To formalise our technique as a program logic based on separation logic [2].

- To extend the setting, by adding to the Java subset we use as our tested language, and thus extending the power of our adversary. An interesting feature to add in that regard would be subclassing, as overriding methods would certainly be an action an attacker could use in a realistic setting.

- To extend our specification constructs to incorporate information flow, such that one could verify not only that an adversary can't modify a field, but also can't read it or read a copy of its content.

- To extend our tool to include termination proofs.

References

[1] P. Müller, M. Schwerhoff, and A. J. Summers. Viper: A verification infrastructure for permission-based reasoning. In B. Jobstmann and K. R. M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 9583 of *LNCS*, pages 41–62. Springer-Verlag, 2016.

[2] J.C. Reynolds. Separation logic: a logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, 2002.

[3] Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames. *ACM Trans. Program. Lang. Syst.*, 34(1), may 2012.