Verification of Information Flow Security for Python Programs

Master Thesis Project Description

Severin Meier
Supervisor: Marco Eilers, Prof. Peter Müller

March 2018

1 Introduction

The VerifiedSCION project aims to prove correctness of the SCION router implementation, which is written in Python. The Nagini verifier[1] was developed for this purpose. It uses deductive verification, which means that the programmer has to supply a specification in the Python code, i.e. pre- and postconditions and loop invariants, and Nagini then tries to prove that the program fulfills the given specification. Internally it makes use of Viper [3], a verification infrastructure with its own intermediate verification language. Nagini transforms the Python program and its specification into a Viper program, for which Viper proves correctness or reports a verification error. The purpose of this project is to add information flow security verification to the Nagini verifier.

Information flow security is a program property which expresses that secret information, such as passwords or encryption keys, is not leaked in the programs public outputs. We can re-formulate this as a *non-interference* property: If the program is run multiple times with the same non-critical inputs, but with different secret values, the public output should always be the same. As it relates multiple executions of a program, *non-interference* is called a *hyperproperty*.

Unfortunately verification tools, such as Viper, typically only reason about single executions. In order to use them to verify a hyperproperty such as information flow security, we need to give the verifier a modified program, which expresses the relation between two executions. It is important that after this transformation, verification can still be done modularly. Modularity allows us to verify each method of the program in isolation, relying only on specifications of other methods, not their body. This is crucial for performance, because without it each verification has to inspect all the code which is reachable from the verified method. Therefore to keep the verification modular, the program is transformed to a *modular product program* [2], which combines the behavior

[1] `https://github.com/marcoeilers/nagini`

of multiple executions into one. This allows us to express the desired properties as a unary specification on a single program execution, which can be verified using existing tools.

Modular product programs are defined generally for k-safety properties, which relate k program executions. Most interesting hyperproperties, including information flow security, are a special case where $k = 2$, which is why we will limit this project to this case for simplicity.

Currently modular product programs are only defined for a small language, which is not expressive enough for encoding the range of programs we aim to verify. Therefore the encoding needs to be extended, to add support for common language features, as well as some specification constructs. These extensions should then allow to define a suitable subset of Python, which allows for interesting programs to be encoded.

Then the transformation from Python program into a modular product program in Viper has to be implemented. For this, we first have to find a way to declare the inputs and outputs as *high* or *low*, meaning they are secret or public respectively. To achieve this Nagini's specification language will have to be extended. The next step is to determine at which point during the encoding from Python to Viper the transformation should be applied. The challenging part here is to do this in a way that reuses existing infrastructure as much as possible. We want to keep the existing encoding from Python to Viper and the partly existing encoding from Viper to modular product encoding separate as much as possible.

The implementation should then be evaluated on example programs from the literature. These programs need to be translated to Python, together with the specifications, and then verified to analyze the performance of the implementation.

2 Core Goals

2.1 Extending the modular product program encoding

The first goal is to extend the modular product program encoding to make it more expressive. These extensions are to be at an abstract level, meaning the encoding is not specific to Python or any other programming language. In particular the required language features are return statements, exception handling (try/catch blocks) and control flow in loops (break and continue). We also need to add support for Viper's specification constructs, namely pure functions and predicates.

An important property we want to keep for all these encodings, is that the user does not have to supply any specifications which are used only for the product transformation. The transformation should rely only on the specifications which are already needed for the functional verification and the declarations for secret data.

2.2 Design Viper encoding

The exact subset of Python for which the modular product program encoding should be implemented needs to be defined. It should include exception handling, dynamically

bound calls, predicates and functions, mutable heap, and support for some built-in types, e.g. lists. We will not support concurrency and obligations [1].

We need to determine the point during the encoding from Python to Viper where the transformation is to be applied. The transformation then should be designed in a way that maximizes reuse of existing infrastructure. In particular the existing encoding from Python to Viper and the partly existing encoding from Viper to modular product program in Viper should be kept separate as much as possible.

2.3 Information Specifications

There needs to be a way to declare information in the Python program as *high* or *low*. This has to be added to Nagini's specification language. For code which does not interact with any secret data, there should be reasonable defaults.

2.4 Implementation

In this step the Nagini verifier is to be extended with the implementation of the modular product program encoding. Depending on the design from Section 2.2, this may include extending Nagini itself, but might also require implementing some transformation on the Viper level directly.

2.5 Evaluation

The evaluation part consists of finding interesting example programs in literature, translating them to Python together with suitable specifications, and verifying them. We are interested in completeness and performance of the implementation.

3 Extension Goals

Here is a list of possible extensions to the project:

- Optimizing the encoding of modular product programs for better verification performance.
- Adding support for obligations.
- Developing a scheme for modelling the runtime of Python programs in Viper, in order to prove the absence of possible timing side channels, which could leak secret information.
- Finding a way to prove that termination depends only on public data, in order to guarantee the absence of termination side channels.

4 References

[1] P. Boström and P. Müller. Modular Verification of Finite Blocking in Non-terminating Programs. In John Tang Boyland, editor, *29th European Conference on Object-Oriented Programming (ECOOP 2015)*, volume 37 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 639–663, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[2] M. Eilers, P. Müller, and S. Hitz. Modular Product Programs. *European Symposium on Programming (ESOP)*, 2018.

[3] 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*, pages 41–62, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.