Product Programs in the Wild: Retrofitting Program Verifiers to check Information Flow Security (Artifact)

Marco Eilers, Severin Meier and Peter Müller

April 29, 2021

1 Getting Started

The artifact is a VirtualBox VM image that contains our implementation and benchmarks, i.e., Nagini with the information flow extension, the product program implementation for Viper, and SecC for comparison. To run it, simply import it into an up-to-date version of VirtualBox (we tested with version 6.1) that has the VirtualBox extension pack installed. It uses 8GB of RAM and four logical cores by default; if these values are too high for your system, feel free to adjust the number of cores, but the VM requires 6-8GB of RAM to work correctly.

The image contains an installation of Ubuntu 20.04. Both the user name and the password are "artifact".

For a quick check to ensure that the setup works, you can for example try the following two steps:

1. Nagini: Run the following commands from a new terminal:

   ```
   cd nagini
   nagini --sif=true tests/sif-true/verification/examples/banerjee.py
   ```

   This should result in the following output (modulo timings):

   ```
   Verification failed
   Errors:
   The precondition of method setName might not hold.
   Assertion LowVal(n) might not hold. (banerjee.py@76.8)
   Verification took 17.82 seconds.
   ```

2. SecC: Run the following commands from a new terminal:

   ```
   cd secc
   ./SecC.sh examples/case-studies/cav2019.c
   ```

   This should result in the following output (modulo timings):

```
examples/case-studies/cav2019.c
  thread1 ...    success  (time 374ms)
  thread2 ...    success  (time 72ms)
  thread1_insecure ...    caught insecure
  thread2_insecure ...    caught insecure
```

2 Artifact Overview

In this document, we will first give an overview of the data available in the artifact; then we will describe the general setup and usage of the modified Nagini tool, show how to replicate the experiments done in the paper, give an overview of the source code of all involved programs, and finally show how to re-create the artifact VM using the supplied "setup_vm.sh" script.

The home folder of the artifact user in the VM contains the following important files and directories:

- Paper: the submitted version of our paper can be found on the desktop and in the home folder of the "artifact" user.
- The "nagini" directory contains the sources of the modified Nagini verifier, along with all Nagini benchmarks (in the "tests" subfolder). Nagini has been pre-installed and can be run by simply executing "nagini <options> path/to/file.py" from the command line.
- The "secc" directory contains the sources and benchmarks of the SecC verifier. These have also been compiled; SecC can be run using the script "/home/artifact/secc/SecC.sh".
- While the "nagini" directory already contains a pre-compiled version of the Viper tool Nagini builds on, the "viper" directory contains the sources of the Viper tool, including the standard Viper language and backend verifiers, as well as the language extensions added for verifying information flow security and the implementation of the product program construction on the Viper level.
- The "scripts" directory contains scripts to automatically re-create the experiments from the paper.
- The "logs" directory contains the outputs we got when running said scripts in the VM
- The "docs" directory contains setup and usage instructions and a tutorial for Nagini that explains the supported specification language.

3 General Tool Setup and Usage

Here, we describe the general usage of the Nagini implementation that was modified to support verification of information flow security; to see how to exactly replicate the evaluation from the paper, see the next section.

Nagini is available open source under the Mozilla Public License 2.0; its source code is hosted publicly on Github[1]. The branch used in this artifact is named "cav_artifact". This branch contains a pre-compiled version of the Viper verification infrastructure that Nagini depends on; Viper's sources are also available open source under the MPL 2.0, and available on Github[2]

Installation instructions for Nagini can be found in the aforementioned Github repository, or in this VM in the file "docs/README.rst". In this VM, Nagini has already been installed.

To run Nagini, execute "nagini <options> path/to/file.py", where the file is a Python 3 file (version 3.5 or newer are supported) that has been annotated with type hints according to PEP484, and that contains specifications in Nagini's own specification language. As an example, a supported file looks as follows:

```
from nagini_contracts.contracts import *

def abs(x: int) -> int:
    Ensures(Result() >= 0)
    Ensures(Implies(Low(x), Low(Result())))
    if x >= 0:
        return x
    return -x
```

This file first imports Nagini's contract library; it then defines a function that computes the absolute value of an integer. The function's input parameter and its return value are annotated with a type hint (they are both integers), and two calls to the "Ensures" function at the top state two postconditions of the function, namely, that the returned result will be non-negative, and that if the input value is low (non-secret), then the result of the function will be non-secret as well.

A tutorial for Nagini's specification language, including the specification constructs used for information flow verification, can be found in the Wiki of Nagini's Github repository[3] or in this VM in "docs/tutorial.md".

The most important command line options supported by Nagini are the following:

- "–sif=v" activates verification of information flow security properties by using the product program construction. v can either be `true` for

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

[2] Viper consists of several repositories which can all be found under `https://github.com/viperproject/`; we will discuss the relevant repositories in one of the following sections.

[3] `https://github.com/marcoeilers/nagini/wiki`

proving ordinary non-interference for sequential programs, `poss` for proving possibilistic non-interference for concurrent programs, or `prob` for proving probabilistic non-interference for concurrent programs, as described in the paper.

- "–benchmark=n" will verify the given file n times in a row and measure and output the verification times. We use this option to measure times for our evaluation.

- "–ignore-obligations" disables the checking of some liveness properties that Nagini normally proves by default (e.g., that all locks that are acquired by a thread are eventually released again, so that no other thread waits for a lock infinitely long). Proving these liveness properties sometimes requires additional specifications and can impact verification time; where these checks are not needed (in particular, when comparing with SecC, which does not prove any liveness properties), we use this option to focus on proving information flow security only.

- "–print-viper" will print the Viper program that Nagini generates from the input Python program. When information flow verification is activated, the printed program will be the modular product program of the encoded Python program.

- "–counterexample" will lead Nagini to print a counterexample for every verification error, i.e., for every statement or piece of specification that could not be verified. As an example, consider a version of the example above where the second postcondition is simply "Ensures(Low(Result()))": this is not correct, since the output obviously leaks secret information if the input is secret. Nagini will show that the postcondition does not hold, and show states from two executions s.t. low values are equal in both executions:

```
Verification failed
Errors:
Postcondition of abs might not hold.
Assertion Low(Result()) might not hold. (arttest.py@5.12).
First execution:
Old store:
  x -> False
Old heap: Empty.
Current store:
  x -> False,
  Result() -> False
Current heap: Empty.

Second execution:
Old store:
  x -> True
Old heap: Empty.
Current store:
  x -> True,
  Result() -> True
```

```
Current heap: Empty.
```

In this example, the input values are non-equal between both executions (since they are high), and therefore the result values are non-equal as well, which violates the postcondition. Note that, in Python 3, booleans are a subtype of integers, which is why Nagini in this case generates a counterexample where integer variables contain boolean values.

4 Reproducing Results from the Paper

We have included scripts to automatically run the experiments to reproduce tables 1, 2, and 3 in the "scripts" folder. We have also included the results we got when running these scripts in the "logs" folder.

Note that, for all benchmarks, the results of running them in a VM with four logical cores will differ from the ones we got when running the evaluation for the paper, which we did on a machine with 12 physical and 24 logical cores, not using a VM. In particular, it is expected that tests that can be verified concurrently using many cores (which Nagini/Viper does by default) will slow down substantially more when reducing the number of available cores than tests that could only be verified mostly sequentially in the first place. Additionally, Nagini/Viper by default starts a verification thread for every logical core on a machine; the overhead caused by this can mean that simple examples actually verify more quickly on a four core machine than on a 12/24 core machine.

4.1 Table 1

The results in Table 1 can be reproduced by running "scripts/table1.sh", which will run Nagini 8 times on each of the examples in Table 1 with the respective appropriate settings. Users may feel free to increase the number of repetitions; we chose 8 because some repetitions are needed for the JVM to warm up (which is why the first few runs are generally slower) and deliver good performance (we state in the paper that measured times were after JVM warmup) but we wanted to keep the time to run the examples somewhat reasonable.

The paths to the respecitve examples are printed out, so that the files (which include annotations that state which error messages are expected) can be inspected manually. As stated in the paper, the examples were adapted from Eilers, Müller and Hitz, ESOP 2018 / TOPLAS 2020; for comparison, the original versions can be found online[4] and also in the directory "other-benchmarks" in the home directory in the VM.

[4] `http://viper.ethz.ch/modularproducts/`

4.2 Table 2

For Table 2,

- "scripts/table2-nagini-sif.sh" reproduces column T_N (Nagini with information flow verification enabled)
- "scripts/table2-nagini-no-sif.sh" reproduces column T_{NP} (Nagini with information flow verification disabled)
- "scripts/table2-secc.sh" reproduces column T_S (SecC with information flow verification enabled)

As before, for Nagini, we do 8 repetitions per example, except for the two longest-running examples with information flow verification enabled, where we have set the number of repetitions to 4 to keep running times reasonable. Note that we have observed that verification times for these two examples can fluctuate quite a bit (e.g. between barely over one minute and 10 minutes).

Here, Nagini is used with the "–ignore-obligations" option throughout to avoid verifying liveness properties not checked by SecC (as explained in the previous section).

4.3 Table 3

For Table 3,

- "scripts/table3-products.sh" reproduces column T_{SIF} (Nagini with the product construction enabled)
- "scripts/table3-noproducts.sh" reproduces column T (Nagini with the product construction disabled)

Here, we do not use Nagini's benchmark option, but instead run the test suite directly; as a result, Nagini will check if the verification result is the same (i.e. conforms to the annotations inside the test files) with and without the product construction. For JVM warmup, four "warmup" files are run at the beginning of the test suite. Note that the relevant tests are the ones marked as "verification" tests; they are followed by a number of "translation" tests that only check if Nagini rejects invalid inputs, which are not included in Table 3 (since the programs are rejected and therefore not verified) and can be ignored.

5 Source Code Description

Here, we will give pointers to the relevant source code.

5.1 Viper

Viper is implemented in Scala. The "viper" directory contains the subdirectories "silver" (the Viper language), "silicon" and "carbon" (Viper's backend verifiers), as well as "silver-sif-extension" and "silicon-sif-extension". The latter two contain extensions of the Viper AST with nodes for information flow specifications and code that performs the product construction.

In particular, in "silver-sif-extension"

- "src/main/scala/SIFAstExtensions.scala" contains classes like "SIFLowExp", representing the assertion `low(e)`, and "SIFLowEventExp()", representing `lowEvent`.
- "src/main/scala/SIFAstExtensions.scala" also contains classes that implement statements like `break` and `continue` ("SIFBreakStmt" and "SIFContinueStmt") which, as mentioned in the paper, are statements that Nagini previously encoded using `goto` statements, but which now have special AST nodes, which makes it easier to construct a product program.
- "src/main/scala/SIFExtendedTransformer.scala" constructs a product program of an (extended) Viper AST.

"silicon-sif-extension" is only relevant when using counterexamples; it translates a counterexample for a Viper product program back to two separate counterexamples of the original Viper program (one for each execution).

5.2 Nagini

Nagini is implemented in Python but used Viper via a library that allows interacting with a JVM from Python. The pre-compiled Viper JAR files are contained in "src/nagini_translation/resources/backends". In general, "src/nagini_contracts" contains contract functions like "Ensures", "Result" and "Low" that are to be imported and used in code to express specifications, and "src/nagini_translation" contains the code that encodes an annotated Python program into a Viper program, invokes a Viper backend, and subsequently translates errors back to the Python level.

Since the product transformation is performed on the level of the encoded Viper program, Nagini's implementation only required few adaptations to enable information flow specifications:

- As mentioned above, "nagini_contracts" now contains additional functions like "Low" and "LowEvent"
- In "nagini_translation", "main.py" offers the "–sif" command line option and invokes the product transformation if it is used (line 138)

- Some of the translator classes that perform the actual encoding from Python to Viper (contained in "nagini_translation/translators", for example the statement translator that contains methods like "translate_stmt_Assign" and "translate_stmt_While") have been subclassed (in "nagini_translation/sif/translators"), either to encode new specification functions like "Low" (in the contract translator) or to use the new extended Viper AST nodes for `break`, `continue`, and other statements that are normally encoded using `goto`s (e.g., the "SIFStatementTranslator", which extends the ordinary "StatementTranslator", overrides function "translate_stmt_Break").

- Additional checks have been added in the ordinary translator classes where necessary. For example, as described in the paper, the statement translator adds an assertion to check that the branch condition of every `if`-statement is low if probabilistic non-interference is verified (line 1011 and following in "nagini_translation/translators/statement.py").

6 Re-Creating the Artifact

The file "setup_vm.sh", included on Zenodo, can be executed as root in a fresh Ubuntu 20.04 VM from the home directory; it will re-create the entire artifact VM (i.e., download and compile/install all tools, download all benchmarks and logs).