Artifact for the paper "Verification-Preserving Inlining in Automatic Separation Logic Verifiers"

Overview

This is the artifact for the OOPSLA 2023 paper "Verification-Preserving Inlining in Automatic Separation Logic Verifiers", which contains:

1. An Isabelle/HOL mechanization that fully supports the technical claims from the paper made in Section 4, also formalizes the structural condition from Section 5 and formally proves that the structural condition implies the semantic condition.
2. An analysis of the test suites of VeriFast, GRASShopper, RSL-Viper, Nagini, which supports the claims in table 1.
    a. We provide scripts to support the results in the rows "satisfy syntactic condition", "violate syntactic condition", "not always preserving".
    b. We do not explain in detail how to obtain the rows "always preserving", "validated by semantic condition", "validated by structural condition" (we provide the files and an informal argument). The reason is the following. Understanding why a file is always preserving (or is captured by the semantic or structural condition) requires an understanding of how the verifiers work and requires conveying an informal proof that is out-of-scope for this artifact (as the call for artifacts states: "paper proofs are not accepted for the evaluation"). The reason why such a proof is required is because one must reason about any inlining bound and any feasible client. We do not expect the reviewers to be able to confirm these claims.
3. An inlining tool for Viper, which inlines calls and unrolls loops, while also checking the structural condition. This supports the claim that we have built such a tool.
4. A test framework that runs the inlining tool on the examples in table 2 (main paper) and table 3 (appendix in the extended version). This supports the claims in Section 6.4.

We describe below how to get started, namely how to use Isabelle/HOL to ensure that all files of the mechanization are successfully verified, then how to use our inlining tool on an example, and finally how to (quickly) check that most of our evaluation performs as expected.
After that we include a detailed step-by-step instructions section.

The source code for our inlining tool is available at https://github.com/tdardinier/carbon/tree/b314c6e16d202139ad812670b681ecbbbb10cf3f, which is a fork of Viper's verification condition generator (it is already cloned in the virtual machine at ~/inlining_tool). The main part of the inlining implementation is done at https://github.com/tdardinier/carbon/blob/b314c6e16d202139ad812670b681ecbbbb10cf3f/src/main/scala/viper/carbon/modules/impls/DefaultInliningModule.scala.

Our artifact comes with a virtual machine and we have recorded all commands used to set up the virtual machine at https://github.com/tdardinier/carbon/blob/b314c6e16d202139ad812670b681ecbbbb10cf3f/etc/vm/setup_vm.sh.

Getting Started Guide

1. Virtual machine

We have installed all required dependencies on an Ubuntu 20.04 VirtualBox virtual machine. To test the artifact, we used 8192 GB of RAM (using less RAM can lead to problems when checking proofs with Isabelle) and 2 processors for the virtual machine and VirtualBox version 6.1. The username is “inlining” and the password is “test”.

2. Mechanization

To get started, we recommend making sure that all the files are successfully verified by Isabelle.

Our mechanization is located in ~/artifact/mechanization, and it contains the following 5 Isabelle files:

- SepAlgebra.thy
- Semantics.thy
- Inlining.thy
- StructuralCondition.thy
- ExtendedInlining.thy

a. Using Isabelle’s CLI

One can check that Isabelle successfully verifies all 5 files using the Isabelle command line interface (located at *~/tools/Isabelle2022/bin/isabelle*, accessible via the shortcut *isabelle*) with the command *“isabelle build -c -d. -l Inlining”* (this command tells Isabelle to build the *Inlining* session, which is defined in the *ROOT* file).

This can be achieved with the following command:

```
> cd ~/artifact/mechanization
> isabelle build -c -d. -l Inlining
```

Expected output:

The final lines of the output should look like the following:

```
…
Session Unsorted/Inlining
  ~/artifact/mechanization/ExtendedInlining.thy
```

~/artifact/mechanization/Inlining.thy
~/artifact/mechanization/Semantics.thy
~/artifact/mechanization/SepAlgebra.thy
~/artifact/mechanization/StructuralCondition.thy
Running Inlining ...
Finished Inlining (0:02:26 elapsed time, 0:08:41 cpu time, factor 3.57)
0:02:31 elapsed time, 0:08:41 cpu time, factor 3.44

This output indicates that Isabelle successfully verified the 5 files in 8 minutes and 41 seconds (it might take a bit longer). A different output might indicate a problem.

b. Using Isabelle's GUI

Note that Isabelle's GUI, which is located at *~/tools/Isabelle2022/Isabelle2022,* can also be used to ensure that Isabelle can verify all files. To verify that a file is successfully verified:

1. Open the file (File > Open…).
2. Open the Theories panel (Plugins > Isabelle > Theories panel). It should be visible on the right of the window.
3. Activate "continuous checking" by ticking the box at the top of the Theories panel.
4. Put the cursor at the end of the file.

The verification status can be seen on the right of the editor, next to the scrollbar:

- Pink indicates a part that has not been verified yet.
- Purple indicates ongoing verification.
- Clear or orange indicates successful verification. Orange indicates a warning (warnings do not make a proof invalid, but provide help to improve the proof).
- Clear or orange indicates successful verification for this part; Orange indicates a warning (warnings do not make a proof invalid, but help to make the proof better).

Red indicates an error (this should not happen).

3. Inlining tool

The Viper file ~/*artifact/example_figure_2.vpr* contains the encoding shown in Figure 2, but where the annotations are not commented out.

1. First, run modular verification :

> inlining artifact/example_figure_2.vpr

(not providing the --SI option results in modular verification)

Verification should be successful.

Output expected:

carbon finished verification successfully in 4.05s.

2. **Run inlining (without checking our conditions):**

```
> inlining --noCheckSC --SI 1 --entry r --ignoreAnnotations artifact/example_figure_2.vpr
```

Verification should fail, which shows that inlining is not preserving in this case.

Output expected:

```
carbon found 1 error in 3.91s:
  [0] Assert might fail. There might be insufficient permission to access A(b).
(example_figure_2.vpr@4.1)
```

The options mean the following:

- --noCheckSC: The tool does **not** check the structural condition
- --SI 1: This option activates inlining, with the bound 1. Note that if this option is not provided, then inlining is not applied (irrespective of any other arguments) but the input Viper file is verified modularly (as was the case in step 1.).
- --entry r: This option specifies the name of the entry method (*r* in this case).
- --ignoreAnnotations: Does inlining without partial annotations (as described in Section 4.2)

3. **Run inlining and check the structural condition:**

```
> inlining --SI 1 --entry r --ignoreAnnotations artifact/example_figure_2.vpr
```

On top of the error from inlining the method, our tool should report two errors here: “Statement might not be monoOut” (i.e., output monotonicity does not hold), and “Statement might not be framing”, which proves that our tool captures this false positive.

Output expected:

```
carbon found 3 errors in 3.57s:
  [0] FRAMING 1: Statement might not be monoOut (order not preserved) ?
(example_figure_2.vpr@14.1)
  [1] FRAMING 1: Statement might not be framing ? (example_figure_2.vpr@14.1)
  [2] Assert might fail. There might be insufficient permission to access A(b).
(example_figure_2.vpr@4.1)
```

4. **Playing with the tool**

Other options of the tool include:

- --disableSyntacticSC: Disable the syntactic condition.
- --noCheckSC: Does not check the structural condition when inlining
- --print file.bpl: Write the Boogie output file to *file.bpl*
- --printSC: Print the Viper code for which we check mono or framing

To find other examples of Viper programs, and understand the Viper syntax, we recommend the Viper tutorial: http://viper.ethz.ch/tutorial/.

4. Quick checks for the evaluation

We propose the following quick checks, to ensure that most of our evaluation works correctly.

a) VeriFast

Run the following commands (should take less than two minutes):

```
> cd ~/artifact/verifast
> sh 1_clone_and_analyze_verifast.sh
> sh 2_run_witnesses_non_preserving.sh | diff -s expected_output.txt -
```

Output expected (first command):

```
…
Automatic analysis of the VeriFast test suite
Number of analyzed files: 1002
Lines of code: 160.4 (mean), 67.0 (median)
Number of files violating the syntactic condition:  696
Number of files considered for manual analysis:  271
```

Output expected (second command):

```
Files expected_output.txt and - are identical
```

b) GRASShopper

Run the following commands (should take less than one minute) and then compare the output of the second script with ~/artifact/grasshopper/expected_output.txt

```
> cd ~/artifact/grasshopper
> sh 1_clone_and_analyze_grasshopper.sh
> sh 2_run_witnesses_non_preserving.sh
```

The first script should output the following numbers at the end:

```
Automatic analysis of the GRASShopper test suite
Number of analyzed files: 314
Lines of code: 123.8 (mean), 57.0 (median)
Number of files violating the syntactic condition: 111
Number of files violating the syntactic condition because of exact bounds on resources: 11
Number of files violating the syntactic condition because of imprecise assertions: 107
```

c) RSL-Viper

Run the following command:

```
> cd ~/artifact/rsl_viper
```

> sh 1_clone_and_analyze_rslviper.sh

The expected output at the end is:

…
14 out of 14 files violate the syntactic condition

d) Nagini

Run the following command:

> cd ~/artifact/nagini
> python3 global_analysis_nagini.py --naginirepo nagini_expected

The expected output is (should not take more than a few seconds):

Python files analyzed:232
Viper files analyzed (should match number of Python files):232
tests that violate the syntactic condition:232
trivial tests:114
non-trivial tests (manual analysis required):118
non-trivial tests considered for manual analysis:79
non-trivial tests discarded:39
tests with permission branching:64
tests with dynamic field branching:62
Captured lines of code for 232 files
All lines of code: mean 73.02155172413794 median 47.5

Run the following command (should not take more than a few seconds):
> source ~/tools/nagini_env/bin/activate
> nagini
> deactivate

The expected output is:
/home/inlining/tools/nagini_env/lib/python3.7/site-packages/mypy/lex.py:770: FutureWarning:
Possible nested set at position 1
  open_bracket_exp = re.compile('[[({]')
usage: nagini [-h] [--viper-jar-path VIPER_JAR_PATH] [--boogie BOOGIE]
          [--z3 Z3] [--mypy-path MYPY_PATH] [--print-silver]
          [--write-silver-to-file WRITE_SILVER_TO_FILE] [-v]
          [--verifier VERIFIER] [--sif] [--show-viper-errors] [--arp]
          [--log LOG] [--benchmark BENCHMARK] [--ide-mode]
          [--select SELECT] [--ignore-global] [--server]
          python_file
nagini: error: the following arguments are required: python_file

e) Test framework

Start the Viper HTTP Server in one terminal by running:
> cd ~/test_framework/viperserver/
> sbt run

After at most a few minutes and around 25 lines of “[info]...” output, the following message should appear:

[info] ViperServer online at http://localhost:PORT

where PORT is some integer. This means the server can be reached on the port PORT (the port is not always the same).

Keep this terminal open and do not stop the process, since it is required for the next command

Open another terminal and run (where PORT is given by the number outputted above and R is the number of times each configuration should be tested):

> cd ~/test_framework/viperserver/carbon/src/test/inlining_test_framework/
> python3 run_inlining_tests.py --port PORT --reps 1 --boogieExe $BOOGIE_EXE --z3Exe $Z3_EXE --dir ../resources/inlining/

This command should finish in around 6 minutes. Check that the output matches “~/artifact/expected_output_test_framework.txt”.

Step by Step Instructions

1. Isabelle/HOL mechanization

Structure of the mechanization

The artifact contains the following 5 Isabelle files:

- *SepAlgebra.thy*: This theory formalizes the separation algebra.
- *Semantics.thy*: This theory formalizes the parametric verification language described in Section 4.1, with its semantics and verification definition (sem and ver), as well as mono and framing.
- *Inlining.thy*: This theory formalizes inlining (Definition 4.1), the semantic condition (Definition 4.4), and proves the verification-preserving inlining theorem without taking (partial) annotations into account (Theorem 4.5).
- *StructuralCondition.thy*: This theory formalizes the structural properties (structural mono and structural framing), the structural condition, and proves that the structural condition implies the semantic condition (Section 5).
- *ExtendedInlining.thy*: This theory proves the verification-preserving inlining theorem when (partial) annotations are taken into account (Theorem 4.7). Moreover, this theory includes the corresponding corollaries of both Theorem 4.5 and 4.7, which

prove that an error in the inlined program implies that there is no annotation that makes the original program verify modularly (the corollary for 4.7 considers only annotations that are stronger than the already existing partial annotation).

Correspondence with the paper

We suggest using Isabelle's GUI to navigate the mechanization (see how in the *Getting Started Guide*), in order to check that it is consistent with the claims in the paper. To jump to the definition of a term, click on it while holding the Control key.

Note the following differences between the formal descriptions in the paper and the Isabelle/HOL mechanization:

1. **Annotations**: Annotations in the Isabelle/HOL mechanization are directly part of the program, whereas the annotation and the program are separate in the paper. That is, the methods always have an annotation and loops always have an invariant.
2. **Renaming**: The paper ignores renaming issues, but the mechanization covers this aspect. In particular, the locale *semantics_algebra* (lines 84-152 in the file *Semantics.thy*, see (**)) has several parameters relevant to renaming. Moreover, the functions *inline* (to inline) and *SC* (corresponding to the semantic condition) have a supplementary parameter that records which variables have been used until this point, to avoid variable capturing when inlining further method calls.
3. **Semantics**: In the paper, the functions *sem* and *ver* take as input a single state. In the mechanization, they take as input a set of states. Moreover, the semantics in the mechanization is defined via a function *semantics* (see table below), from which we derive *sem* and *ver*. Accordingly, the definitions of *mono* and *framing* in Isabelle/HOL are also expressed in terms of sets of states instead of single states.

The table below connects the claims in the paper with the Isabelle/HOL mechanization. To show the line numbers, click on "View > Toggle Line Numbers".

Paper		**Isabelle/HOL mechanization**	
Section	**Element(s)**	**File**	**Element(s)**
4.1	State model	SepAlgebra.thy	locale (*) *commutative_monoid* (lines 9-17) locale (*) *sep_algebra* (lines 158-175)
	Language and semantics	Semantics.thy	datatype *stmt* (lines 7-19) function *semantics* (lines 1409-1445)
	Functions *ver* and *sem*	Semantics.thy	locale (**) *semantics_algebra* (lines 84-152) definitions *ver* and *sem* (lines 1484-1488) (**)
4.2	Definition 4.1	Inlining.thy	function *inline* (lines 46-60)
	Definition 4.2	Semantics.thy	definitions *ssafeMono*, *smonoOut*, and *smono* (lines 1497-1504) (**)
	Definition 4.3	Semantics.thy	function *framing* (lines 3939-3941) (**)

	Definition 4.4	Inlining.thy	functions *SC* and *inlinable_SC* (lines 106-124) (****)
	Theorem 4.5	Inlining.thy	theorem *preservation* (lines 4310-4323)
	Corollary of Theorem 4.5	ExtendedInlining.thy	theorem *preservation_corollary* (lines 4484-4512)
4.3	Definition 4.6	ExtendedInlining.thy	functions *annotate_stmt*, *annotate_method*, and *annotate_program* (lines 22-37) (***)
	Theorem 4.7	ExtendedInlining.thy	theorem *extended_preservation* (lines 4026-4110)
	Corollary of Theorem 4.7	ExtendedInlining.thy	theorem *extended_preservation_corollary* (lines 4112-4118)
5	Definition 5.1	StructuralCondition.thy	definition *structural_mono* (lines 8-13)
	Lemma 5.2 (in the updated Section 5 shown in "paper_updated_section_5.pdf") [structural mono implies mono]	StructuralCondition.thy	lemma *structural_mono_implies_mono* (lines 15-39)
	Theorem 5.3 (in the updated Section 5 shown in "paper_updated_section_5.pdf") [structural condition implies semantic condition]	StructuralCondition.thy	theorem *structural_SC_implies_SC* (lines 110-175)
App. A.1	Definition A.1	SepAlgebra.thy	locale (*) commutative_monoid (lines 9-17) locale (*) sep_algebra (lines 158-175)
App. A.2	Language	Semantics.thy	datatype *stmt* (lines 7-19)
	Definition A.2	SepAlgebra.thy	type_synonym *'a assertion* (line 5)
	Definition A.3	SepAlgebra.thy	definition *add_set (⊗)* (lines 374-375) definition *bigger_set (≫)* (lines 443-444)
	Definitions A.4 and A.6	Semantics.thy	locale (**) *semantics_algebra* (lines 84-152) function *semantics* (lines 1409-1449) definitions *ver* and *sem* (lines 1484-1488) (**)
	Definition A.5	Semantics.thy	definition *h* (lines 474-475) definition *h_comp* (lines 673-674)
App. B	Definition B.1	Inlining.thy	function *inline* (lines 46-60)
	Definition B.2	Inlining.thy	functions *SC* and *inlinable_SC* (lines 106-124) (****)

	Definition B.3	ExtendedInlining.thy	function *annotate_stmt* (lines 22-30)
App. F	Definition F.1 (in the updated appendix of "paper_updated_section_5.pdf")	StructuralCondition.thy	definition *structural_framing* (lines 42-45)
	Lemma F.2 (in the updated appendix of "paper_updated_section_5.pdf")	StructuralCondition.thy	lemma *structural_framing_implies_framing* (lines 47-70)

(*) A "locale" is a way in Isabelle/HOL to define a parametric theory, by fixing some parameters and assuming some axioms. In our case, the locale *sep_algebra*, which inherits from the locale *commutative_monoid*, corresponds to the separation algebra described in Section 4.1 and Appendix A (definition A.1).

(**) The locale *semantics_algebra* captures the parameters and requirements needed to instantiate our parametric verification language. In particular, it covers all aspects relevant to renaming variables, and the heuristics for *exhale* statements.

(***) Since (in our semantics) *assert P* verifies in a state **larger than** some state in which the assertion *P* holds, we only use "*P*" and not *"P * true*" in the mechanization.

(****) The terminology "call-free" in the paper corresponds to the negation of *inlinable* in the mechanization: *inlinable(s)* holds iff the statement *s* contains a loop or a method call.

2. Evaluation Part 1 (table 1)

In the following section, we explain how we obtained the results for table 1 discussed in Section 6.1 and Section 6.2. We explain our approach for each of the four verifiers (VeriFast, GRASShopper, RSL-Viper, Nagini) in a separate subsection. Each of the subsections is structured as follows:

1. First, we explain how to obtain the corresponding verifier test suite and how to perform an automatic analysis on the test suite. The automatic analysis on the test suite is used to (1) find files that violate the syntactic condition (i.e., rows "satisfy syntactic condition" and "violate syntactic condition" in table 1), (2) automatically discard files for the manual analysis, (3) obtain numbers on different patterns appearing in the test suite.
2. Second, we explain our manual analysis discussed in 6.2 and 6.3.
    a. For the "not always preserving files" we provide "non-preserving witnesses" that show that the file is not always preserving. That is, we add clients (i.e., an initial statement) to the analyzed files such that: the file verifies modularly, but does not verify if the client is inlined. We have written a script for each verifier that runs the modular verification followed by the verification of the inlined program.

b. If a file is always preserving (and typically always captured by our semantic and structural conditions), we write the informal reason (coming from our manual analysis of these files) for why this is the case. Understanding why the informal reasons imply that a file is always preserving requires an understanding of how the verifiers work and requires conveying an informal proof that is out-of-scope for this artifact (as the call for artifacts states: “paper proofs are not accepted for the evaluation”). We do not expect the reviewers to be able to confirm these claims.

VeriFast

1. Clone the VeriFast test suite, and run the automatic analysis

The script *1_clone_and_analyze_verifast.sh* in the folder *~/artifact/verifast* first clones the VeriFast repository (with the relevant commit), and then runs the Python file *verifast.py* (see the comments in the file for more details), which performs the automatic analysis:

```
> cd ~/artifact/verifast
> sh 1_clone_and_analyze_verifast.sh
```

Output expected (should take less than a minute):

```
Cloning into 'verifast'...
remote: Enumerating objects: 22153, done.
remote: Counting objects: 100% (161/161), done.
remote: Compressing objects: 100% (80/80), done.
remote: Total 22153 (delta 83), reused 134 (delta 74), pack-reused 21992
Receiving objects: 100% (22153/22153), 7.74 MiB | 11.93 MiB/s, done.
Resolving deltas: 100% (15824/15824), done.

Automatic analysis of the VeriFast test suite
Number of analyzed files: 1002
Lines of code: 160.4 (mean), 67.0 (median)
Number of files violating the syntactic condition:  696
Number of files considered for manual analysis:  271
```

The analysis script also writes two files:

- *selection.txt*: List of the 271 files that were considered for the manual analysis. The expected output for this file is in *expected_selection.txt*.
- *discarded.txt*: List of the 425 files that were **not** considered for the manual analysis. The expected output for this file is in *expected_discarded.txt*.

2. Manual analysis

The table below shows the 20 files that were randomly sampled from the list of files considered for the manual analysis (*selection.txt*). For each file, we write whether it is always preserving or not.

Always-preserving files: If a file is always preserving (and typically always captured by our semantic and structural conditions), we write the informal reason (coming from our manual analysis of these files). Since these reasons can be considered as informal proofs, and require some knowledge of how VeriFast works to be understood, we do not expect the reviewers to be able to confirm these claims.

Non-preserving witnesses: For each file that is not always preserving (in red), we have written a client that proves this claim. These clients are in the folder *~/artifact/verifast/non_preserving_witnesses*. More precisely, for each file *FILE.c* not always preserving, we have written two modified files: *FILE_client.c*, which only contains the client (non-inlined), and *FILE_client_inlined.c*, in which we have also manually inlined the method calls and unrolled loop iterations manually. Running the tool on *FILE_client.c* should verify (since verification is modular), whereas running the tool on *FILE_client_inlined.c* should fail, which thus proves non-preserving inlining.

Note that, in some tedious cases, we inline or unroll only partially (i.e., we keep method calls or loops that should have been replaced by *assume false* because the inlining bound has been reached), since this is sufficient to prove non-preserving inlining. Every partially inlined client could be turned into an inlined client where we *assume false* once the bound is reached, and we would get the same results.

To automatically run all the witnesses (first showing successful modular verification and then showing verification failure with the inlined client), run the script *2_run_witnesses_non_preserving.sh* from the *~/artifact/verifast* folder:

```
> cd ~/artifact/verifast
> sh 2_run_witnesses_non_preserving.sh
```

It should take less than a minute to run. The expected output of this script is given in the file *~/artifact/verifast/expected_output.txt*.

All paths in the table below are relative to the folder *~/artifact/verifast/verifast/examples*.

Path	Always Preserving	Reason
crypto_ccs/bin/stdlib/crypto/auxiliary_definitions.c	No	see client in non_preserving_witnesses folder
dyncode.c	Yes	• potentially imprecise assertions are precise • “open” statement directly following the relevant “close” statement, so it does not have a choice
floating_point/sqrt_with_rounding/sqrt_with_rounding.c	No	see client in non_preserving_witnesses folder

shared_boxes/datastructures/dklmap.c	Yes	“exwitness(?)” statements always following the relevant “close” or “assert” statements, so they do not have a choice
java/Account.java	Yes	potentially imprecise assertions are precise
fm2012/problem1-split.c	No	see client in non_preserving_witnesses folder
shared_boxes/stack_hp/listex2.c	No	see client in non_preserving_witnesses folder
vstte2012/problem5/lset.c	Yes	“exwitness(?)” statements always following the relevant “close” or “assert” statements, so they do not have a choice
java/Iterator/SingletonIterator.java	Yes	potentially imprecise assertions are precise
shared_boxes/shared_boxes.c	No	see client in non_preserving_witnesses folder
vstte2012/problem4/treerec_list.c	Yes	potentially imprecise assertions are precise
java/Iterator.java	Yes	potentially imprecise assertions (“valid(?)”) are precise
quicksort.c	Yes	potentially imprecise assertions (“ints(,,?)”) are precise
splitcounter/splitcounter.c	No	see client in non_preserving_witnesses folder
monitors/barbershop.c	No	see client in non_preserving_witnesses folder
shared_boxes/incrbox.c	Yes	imprecise “leak” always following the allocation of memory, so they do not have a choice
mcas/ghost_lists.c	No	see client in non_preserving_witnesses folder
linking/simple_pred/give_pred_body_main.c	Yes	“open” statements directly following the relevant “close” statement, so they do not have a choice
shared_boxes/datastructures/dlset.c	Yes	• potentially imprecise assertions are precise • “open” statement directly following the relevant “close” statement, so it does not have a choice
java/chat/Member.java	Yes	“open” statements directly following the relevant “assert” statements, so they do not have a choice

1. **Clone the GRASShopper test suite, and run the automatic analysis**

The script *1_clone_and_analyze_grasshopper.sh* in the folder ~/*artifact/grasshopper* first clones the GRASShopper repository (with the relevant commit), and then runs the Python file *grasshopper.py* (see the comments in the file for more details), which performs the automatic analysis:

```
> cd ~/artifact/grasshopper
> sh 1_clone_and_analyze_grasshopper.sh
```

Output expected (should take a few seconds):

```
Cloning into 'grasshopper'...
remote: Enumerating objects: 22254, done.
remote: Counting objects: 100% (1379/1379), done.
remote: Compressing objects: 100% (418/418), done.
remote: Total 22254 (delta 946), reused 1378 (delta 946), pack-reused 20875
Receiving objects: 100% (22254/22254), 7.79 MiB | 8.74 MiB/s, done.
Resolving deltas: 100% (14679/14679), done.

Automatic analysis of the GRASShopper test suite
Number of analyzed files: 314
Lines of code: 123.8 (mean), 57.0 (median)
Number of files violating the syntactic condition: 111
Number of files violating the syntactic condition because of exact bounds on resources: 11
Number of files violating the syntactic condition because of imprecise assertions: 107
```

2. **Manual analysis**

The table below shows the 20 files that were randomly sampled for the manual analysis. For each file, we write whether it is always preserving or not.

Always-preserving files: If a file is always preserving (and typically always captured by our semantic and structural conditions), we write the informal reason (coming from our manual analysis of these files). Since these reasons can be considered as informal proofs, and require some knowledge of how GRASShopper works to be understood, we do not expect the reviewers to be able to confirm these claims.

Non-preserving witnesses: For each file that is not always preserving (in red), we have written a client that proves this claim. These clients are in the folder ~/*artifact/grasshopper/non_preserving_witnesses*. More precisely, for each file *FILE.spl* not always preserving, we have written two modified files: *FILE_client.spl*, which only contains the client (non-inlined), and *FILE_client_inlined.spl*, in which we have also manually inlined the method calls and unrolled loop iterations manually. Running the tool on *FILE_client.spl* should verify (since verification is modular), whereas running the tool on *FILE_client_inlined.spl* should fail, which thus proves non-preserving inlining.

To automatically run all the witnesses (first showing successful modular verification and then showing verification failure with the inlined client), run the script *2_run_witnesses_non_preserving.sh* from the *artifact/grasshopper* folder:

```
> cd ~/artifact/grasshopper
> sh 2_run_witnesses_non_preserving.sh
```

It should take less than a minute to run. The expected output of this script is given in the file *~/artifact/grasshopper/expected_output.txt*.

All paths in the table below are relative to the folder *~/artifact/grasshopper/grasshopper/tests/spl*.

Path	Always Preserving	Reason
array/destroy.spl	Yes	potentially imprecise assertions are precise
array/quicksort.spl	Yes	potentially imprecise assertions are precise
array/selection_sort.spl	Yes	the argument *content* is existentially quantified (implicit ghost), but it is not used
counter/counter_simple.spl	No	see client in non_preserving_witnesses folder
dl/dl_dancing.spl	Yes	potentially imprecise assertions are precise
flows/b-link-core.spl	Yes	• potentially imprecise assertions are precise • disjunctions are pure
flows/b-link-half-old.spl	Yes	potentially imprecise assertions are precise
flows/b-link-half.spl	Yes	potentially imprecise assertions are precise
flows/hashtbl-link.spl	Yes	• potentially imprecise assertions are precise • disjunctions are pure
include/treeset.spl	Yes	potentially imprecise assertions are precise
list_set/delete.spl	Yes	potentially imprecise assertions are precise
list_set/difference.spl	Yes	potentially imprecise assertions are precise
list_set/intersect.spl	Yes	potentially imprecise assertions are precise
list_set/traverse.spl	Yes	potentially imprecise assertions are precise
recursive_defs/list.spl	No	see client in non_preserving_witnesses folder
recursive_defs/treeset.spl	Yes	potentially imprecise assertions are precise
simple_arrays/bubble.spl	Yes	the parameters *ub* and *lb* of the procedure *bubble_sort* are existentially quantified; it thus seems that the precondition *array_bnd(a, lb, ub)* is imprecise, but it

		nonetheless does not seem possible to get non-preserving inlining out of this, because of the way the automation for this existential works
tree/binary_search_tree.spl	Yes	potentially imprecise assertions are precise
tree/union_find.spl	Yes	potentially imprecise assertions are precise
uninterpreted_datat_set/hashset.spl	Yes	potentially imprecise assertions are precise

RSL-Viper

0. General remarks about RSL-Viper

RSL-Viper is a tool to automatically verify weak-memory programs using the RSL logic by encoding programs into Viper. The originally published RSL-Viper tool is not distributed as a separate standalone tool. Instead, programs are directly expressed in Viper, where operations are expressed via Viper macros. The macro definitions provide the Viper encoding.
Each RSL-Viper file in the test suite is a Viper file where the first part of the file is given by the program expressed via Viper macros and the rest of the file includes the macro definitions along with other background theory needed for the encoding. For example, “fetchUpdate(count, t, t+1, true, false)” expresses an atomic fetch-and-add operation. In Viper, the syntax “define fetchUpdate(x, tmp, newVal, readSync, writeSync) { … }” defines the macro.

1. Clone the RSL-Viper test suite, and run the automatic analysis

The script *1_clone_and_analyze_rslviper.sh* in the folder *~/artifact/rsl_viper/* extracts the RSL-Viper tests from the Viper examples repository (with the relevant commit), and then runs the Python file *analyze_rsl_viper.py* (see the comments in the file for more details), which performs the automatic analysis:

```
> cd ~/artifact/rsl_viper
> sh 1_clone_and_analyze_rslviper.sh
```

The expected output is:

14 out of 14 files violate the syntactic condition

For the lines of code reported in the caption of table 1, we used all lines from the first part of the program to the last part of the program (excluding the comments at the beginning and the macro definitions and background theory).

Proof rule selection strategies and exact bounds on resources

In the paper, we report that proof rule selection strategies that depend on the owned resources appear in 5 files. We established this by manual inspection and the 5 files are given by:

- RelAcqMsgPass.sil
- RelAcqDblMsgPassSplit.sil
- FencesDblMsgPass.sil
- FencesDblMsgPassAcqRewrite.sil
- FencesDblMsgPassSplit.sil

One can observe that all 5 files have such proof rule selection strategies by noticing that each of these files use the “waitOnAcquireRead” or “waitOnRelaxedRead” macros, which are defined in terms of the macro atomicReadInhaleWithVar(x, v, sync, isRMW, tmpSet) where isRMW is set to false. As can be seen in the macro definition for atomicReadInhaleWithVar, if isRMW is false, then there is a branch on the permission of the AcqConjunct resource (see macro hasAcqConjunct) that corresponds to the proof rule selection strategy discussed in figure 2 of the paper.

Considered for manual analysis
We took all files into account for the manual analysis except for FollyRWSpinlock_err.sil and FollyRWSpinlock_err_mod.sil, because they were too complex to analyze.

2. Manual analysis

The table at the end of this subsection shows the 12 files that were taken into account for the manual analysis. For each file, we write whether it is always preserving or not.

Always-preserving files: If a file is always preserving (and typically always captured by our semantic and structural conditions), we write the informal reason (coming from our manual analysis of these files). Since these reasons can be considered as informal proofs, and require some knowledge of how RSL-Viper works to be understood, we do not expect the reviewers to be able to confirm these claims.

Non-preserving witnesses: For each file that is not always preserving (in red), we have written a client that proves this claim. These clients are in the folder *~/artifact/rsl_viper/non_preserving_witnesses*. More precisely, for each file *FILE.sil* not always preserving, we have written one modified file: *FILE_client.sil*, which contains the client (non-inlined) [the original file is also included in the same folder]. Verifying *FILE_client.sil* should verify modularly, whereas inlining the client (with some bound) in *FILE_client.sil* should fail, which thus proves non-preserving inlining.
Note that in some cases we commented out irrelevant methods of the program in *FILE_client.sil.*

Since we are dealing with Viper files here, we can use our Viper inlining tool to show that inlining the clients leads to verification failing.

To automatically run all the witnesses (first showing successful modular verification and then showing verification failure with the inlined client), run the script *2_run_witnesses_non_preserving.sh* from the *~/artifact/rsl_viper* folder:

> cd ~/artifact/rsl_viper
> sh 2_run_witnesses_non_preserving.sh

The command takes around 4 minutes. The expected output of this script is given in the file *~/artifact/rsl_viper/expected_output.txt*.

All paths in the table below are relative to the folder *~/artifact/rsl_viper/examples* (which is created once the examples are cloned with the script in step 1).

Path	Always Preserving	Reason
FencesDblMsgPass.sil	No	see client in non_preserving_witnesses folder
FencesDblMsgPassAcqRewrite.sil	No	see client in non_preserving_witnesses folder
FencesDblMsgPassSplit.sil	No	see client in non_preserving_witnesses folder
FollyRWSpinlockStronger_mod.sil	Yes	The problematic statements that could lead to nonpreserving inlining are the CAS and fetch_update macros, which both take arguments “true” and “true” in this case. The resulting expanded Viper pattern cannot lead to nonpreserving inlining, because owning more permission can only lead to more permission being obtained (and the obtained permission does not interact with any non-monotonic features).
FollyRWSpinlockStronger.sil	Yes	Same reason as FollyRWSpinlockStronger_mod.sil
RelAcqDblMsgPassSplit.sil	No	see client in non_preserving_witnesses folder
RelAcqMsgPass.sil	No	see client in non_preserving_witnesses folder
RelAcqRustARCStronger.sil	No	see client in non_preserving_witnesses folder
RSLLockNoSpin-not-in-appendix.sil	Yes	This file is always preserving and captured by the semantic condition, but it is **not** captured by the structural condition, because of an incompleteness of determinization. More precisely, a method in this file creates (at some point) the set of all references to which it has some non-zero permission to some field (using permission introspection features from Viper), which is different depending on the resources owned. The method is always preserving (and captured by our semantic condition), because of the way this set is subsequently used. However, this set is created as follows (simplified): *var my_set: Set[Ref]* *assume (forall r: Ref :: r in my_set <==> perm(r.f) > 0)* The issue is that determinization is a bit too naive, and thus it will force *my_set* to start with the same value in both executions it compares. In this case, it cannot prove that the assume will not

		remove the trace (i.e., it cannot prove *exist* on line 8 in Figure 8), and thus it fails. Note that the atomic read operation is not problematic in this file (as opposed to, e.g., RelAcqDblMsgPassSplit), because in this file full ownership of AcqConjunct is never held, and thus the problematic branch is never being taken.
RSLSpinlock.sil	Yes	This file is always preserving and captured by the semantic condition, but it is not captured by the structural condition, for the same reason as RSLLockNoSpin-not-in-appendix.sil.
RustARCOriginal_err.sil	No	see client in non_preserving_witnesses folder
RustARCStronger.sil	No	see client in non_preserving_witnesses folder

Nagini

0. General remarks about Nagini

Nagini is an automatic verifier for Python programs that works by translating a Python file to a Viper file. In our evaluation, we analyzed inlining in terms of the generated Viper file.

If you want to try out Nagini yourself in the VM, you can do so by activating the Python virtual environment in which Nagini was installed:

> source ~/tools/nagini_env/bin/activate
> nagini --write-silver-to-file INPUTFILE_viper_encoding.vpr INPUT_FILE.py

You can deactivate the virtual environment with the command “deactivate”.

1A. Clone repository and generate Viper encodings [NOTE: this command takes 1 hour in the virtual machine]

For our analysis, we require Nagini’s Viper encoding for every Python file in Nagini’s test suite. To do so, we must run Nagini on every Python file in Nagini’s test suite which takes quite long. In the VM, it takes around an hour. We have already prerun the associated command and stored the resulting Nagini repository with the corresponding Viper files in the folder “~/artifact/nagini/nagini_expected” (the tests are given by all Python files in “~/artifact/nagini/nagini_expected/tests”; for each Python file “FILE.py” there is the corresponding Nagini-generated Viper encoding “FILE.vpr”).

We now describe the associated command (that we have prerun)

Comand (takes around 1 hour on the VM):

> cd ~/artifact/nagini
> source ~/tools/nagini_env/bin/activate
> sh clone_nagini.sh
> deactivate

Parts of the expected output of the command is given in "~/artifact/nagini/expected_output_after_clone_nagini.txt". The command stores the cloned Nagini repository with the Python and Viper files in "~/artifact/nagini/nagini"; the test suite is in "~/artifact/nagini/nagini/tests".

More precisely, the command does the following

1. Activate the Python virtual environment in which Nagini was installed
2. Clone the Nagini repository
3. Remove all Python files that are not verification tests (e.g., parser and consistency tests) and that are not part of Nagini's main test suite (i.e., remove tests that require Nagini extensions)
4. Generate Nagini's Viper encodings for every Python file in Nagini's main test suite

clone_nagini.sh has comments explaining the different steps

1B. Automatic analysis of test suite

Run the following command to obtain the automatic analysis of the Nagini test suite (obtained either by running the above command or directly using "~/artifact/nagini/nagini_expected"):

```
> cd ~/artifact/nagini
> python3 global_analysis_nagini.py --naginirepo NAGINI_REPO_DIR
```

where NAGINI_REPO_DIR is "nagini" if you ran the command above or "nagini_expected" if you want to reuse the precomputed Nagini test suite with the corresponding Viper files

The expected output is (should not take more than a few seconds):

```
# Python files analyzed:232
# Viper files analyzed (should match number of Python files):232
# tests that violate the syntactic condition:232
# trivial tests:114
# non-trivial tests (manual analysis required):118
# non-trivial tests considered for manual analysis:79
# non-trivial tests discarded:39
# tests with permission branching:64
# tests with dynamic field branching:62
Captured lines of code for 232 files
All lines of code: mean 73.02155172413794 median 47.5
```

In the following, we give a brief description of this output. The filenames in the following bullet points are relative to ~/artifact/nagini.

- Trivial tests: A test is "trivial" if we could establish syntactically that inlining is trivially always preserving (even though the syntactic condition may be violated) using knowledge about how Nagini produces Viper-generated files. The script stores those files in "trivial.txt" and expected content is in "expected_trivial.txt". Nontrivial tests are all other tests (i.e., manual analysis is required for nontrivial tests). (note that in

expected_trivial.txt the relative paths may be different [w.r.t. nagini or nagini_expected], and the order may be different but the tests are the same)

- The non-trivial files considered for manual analysis are stored in selection.txt. The expected content is in “expected_selection.txt” (note that in expected_selection.txt the relative paths may be different [w.r.t. nagini or nagini_expected], and the order may be different but the tests are the same).
- From the nontrivial tests, 39 files tests were automatically discarded for the manual analysis, since they contain features that were too hard to analyze (see ~/artifact/nagini/global_analysis.py for more details).The corresponding file paths are stored in discarded.txt. The expected content is in “expected_discarded.txt” (note that in expected_discarded.txt the relative paths may be different [w.r.t. nagini or nagini_expected], and the order may be different but the tests are the same).
- Tests with permission branching: These are tests that branch on the available resources in the Viper encodings using resource introspection. The corresponding file paths are stored in branch.txt.
- Tests with dynamic field branching: These are tests that branch on the available resources using Viper’s resource introspection as a result of a recurring Nagini pattern that can be shown to satisfy the structural condition only thanks to the bounded relaxation. The corresponding paths are stored in dynamic_field_branch.txt.

2. Manual analysis

The table below shows the 20 files that were randomly sampled for the manual analysis. For each file, we write whether it is always preserving or not.

Always-preserving files: If a file is always preserving (and typically always captured by our semantic and structural conditions), we write the informal reason (coming from our manual analysis of these files). Since these reasons can be considered as informal proofs, and require some knowledge of how Nagini works to be understood, we do not expect the reviewers to be able to confirm these claims.

Non-preserving witnesses: For each file that is not always preserving (in red), we have written a client (i.e., a witness) that proves this claim. These clients are in the folder *~/artifact/nagini/non_preserving_witnesses*. More precisely, for each file *FILE.vpr* not always preserving, we have written one modified file: *FILE_client.vpr*, which contains the client (non-inlined). Verifying *FILE_client.vpr* should verify modularly, whereas inlining the client (with some bound) in *FILE_client.vpr* should fail, which thus proves non-preserving inlining. Note that in some cases we commented out irrelevant methods of the program in *FILE_client.vpr.*

Since we are dealing with Viper files here, we can use our Viper inlining tool to show that inlining the clients leads to verification failing.

To automatically run all the witnesses (first showing successful modular verification and then showing verification failure with the inlined client), run the script *2_run_witnesses_non_preserving.sh* from the *~/artifact/nagini* folder:

> cd ~/artifact/nagini
> sh run_witnesses_non_preserving.sh

It takes around 2 minutes to run. The expected output of this script is given in the file *~/artifact/nagini/expected_output_nonpreserving.txt*.

All paths in the table below are relative to the folder *~/artifact/nagini/nagini/tests* (or *~/artifact/nagini/nagini_expected/tests* if you use the already-prepared Nagini test suite)

Path	Always Preserving	Reason
functional/verification/examples/keon_knapsack.vpr	Yes	dynamic field branch not an issue
functional/verification/examples/parkinson_recell.vpr	Yes	dynamic field branch not an issue
functional/verification/examples/test_student_enroll_preds.vpr	Yes	dynamic field branch not an issue
functional/verification/issues/00112.vpr	Yes	dynamic field branch not an issue
functional/verification/test_definedness.vpr	Yes	dynamic field branch not an issue
functional/verification/test_dynamic_field_creation.vpr	Yes	dynamic field branch not an issue
functional/verification/test_exception.vpr	Yes	dynamic field branch not an issue
functional/verification/test_generic_classes.vpr	Yes	dynamic field branch not an issue
functional/verification/test_isinstance.vpr	Yes	dynamic field branch not an issue
functional/verification/test_raised_exception.vpr	Yes	dynamic field branch not an issue
functional/verification/test_thread_fork.vpr	Yes	• dynamic field branch not an issue • inhale forperm but only for bound 0 (no obligations)
functional/verification/test_union_contracts.vpr	Yes	dynamic field branch not an issue
functional/verification/test_wildcard_permissions.vpr	Yes	wildcard (existential fraction) only appears in non-problematic positions
obligations/verification/test_while_must_terminate.py	Yes	• always preserving, because inlined program always verifies • semantic condition does not capture program (see reason below)
obligations/verification/chalice2silver/christian/obl_loop.vpr	No	see client in non_preserving_witnesses folder

obligations/verification/chalice2silver/lifetime.vpr	Yes	• only need to consider existing clients because would need obligations otherwise
obligations/verification/chalice2silver/loopsAndRelease.vpr	No	see client in non_preserving_witnesses folder
obligations/verification/test_behavioral_subtyping.vpr	No	acquired locks are always released
obligations/verification/test_method_leak_check.vpr	No	see client in non_preserving_witnesses folder
obligations/verification/test_waitlevels.vpr	Yes	see client in non_preserving_witnesses folder

Comments on inlining in Nagini

- Note that we analyze inlining in the Nagini-generated Viper files directly and not on the Python files, because our inlining tool works directly on the Viper files.
- Our tool currently does not reliably support gotos and labels. In almost all Nagini-generated Viper files gotos and labels exist, but they can be trivially removed without changing the meaning of the program. Before running our inlining tool, we removed gotos and labels manually (and made sure that the meaning of the program does not change).
- Nagini translates bodyless Python library methods into Viper methods with the body “inhale false” (which is the same as assume false). Before inlining, we made sure that bodyless Python library methods are also bodyless in the Viper encoding.
- Nagini often uses Viper inhale-exhale assertions, which are written as [A,B] in specifications. Their meaning is that when checking the assertion (i.e., checking the precondition) [A,B] is treated as B and when assuming the assertion (i.e., obtaining the postcondition) [A,B] is treated as A. For inlining at the Viper level, it does not make sense to take such inhale-exhale assertions into account when asserting partial annotations, because our inlining theorems that relate inlining to modular verification require that, e.g., the precondition which a caller checks is the same as the precondition which the corresponding callee assumes. As a result, we ignore all inhale-exhale assertions except in the entry method (i.e., the initial statement), because no one calls the entry method (our inlining tool automatically ignores inhale-exhale assertions in non-entry methods, but when considering modular verification to relate to what inlining does for Nagini-generated Viper programs we must comment them out).
- As we write in the paper (footnote 7), we use restrictions on possible Nagini clients when doing the analysis in table 2. In particular, we do not consider new clients (i.e., clients that do not already exist in the files) that themselves obtain “obligation resources” (e.g., via the precondition), because Nagini contains leak checks (via resource introspection) that would lead to inlining almost always being nonpreserving, which would not reflect the typical scenario.

Always preserving file that is not captured by semantic condition

The Nagini test
~/artifact/nagini/nagini_expected/tests/obligations/verification/test_while_must_terminate.py

is always preserving, because the inlined program always verifies (if one takes the “Comments on nonpreserving inlining in Nagini” into account). However, the semantic condition does not capture the example. The reason is because of the statement

```
exhale perm(MustTerminate(_cthread_158)) > none ==>
acc(MustTerminate(_cthread_158), perm(MustTerminate(_cthread_158)) -
_loop_original_must_terminate)
```

in the Viper encoding. Because the inlined program contains ownership to MustTerminate(t), the statement is not mono (the statement verifies trivially with zero ownership of MustTerminate(t), but may not verify with more ownership of MustTerminate(t)). Note that the statement is mono if the inlined program has no ownership to MustTerminate(t) due to the bounded relaxation (which is important, because this scenario occurs often in Nagini).

3. Evaluation Part 2: Table 2

In this part, we show how we obtained the results for table 2 in the paper and for table 3 in the appendix of the extended version, discussed in Section 6.4.

Table 2: Files and test configurations

The following table provides the mapping between file names used in table 2 and the actual paths of the original files in the corresponding test suites (the paths are relative to the root folder of the respective verifier repository):

Table 2 filename	Test suite path
iap_bst (Nagini)	tests/functional/verification/examples/iap_bst.py
parkinson_recell (Nagini)	tests/functional/verification/examples/parkinson_recell.py
loops_and_release (Nagini)	tests/obligations/verification/chalice2silver/loopsAndRelease.py
rust_arc (RSL)	RelAcqRustARCStronger.sil
lock_no_spin (RSL)	RSLLockNoSpin-not-in-appendix.sil
msg_pass_split_1 (RSL)	RelAcqDblMsgPassSplit.sil
msg_pass_split_2 (RSL)	RelAcqDblMsgPassSplit.sil
account (VeriFast)	examples/java/Account.java
lcp (VeriFast)	examples/fm2012/problem1-split.c
iterator (VeriFast)	examples/java/Iterator/SingletonIterator.java
stack (VeriFast)	examples/java/Stack.java
bstree (GRASShopper)	tests/spl/recursive_defs/bstree.spl

nodes (GRASShopper)	tests/spl/symb_exec/basic.spl

The three examples colored yellow (iterator, stack, nodes) were added after the artifact evaluation.

The examples from table 3 were taken from the Viper examples repository (https://github.com/viperproject/examples, commit hash 5ecb3527baf20d7bc25edeb) [note that this repository also contains the RSL examples, but those are not included in table 3].

For table 2 and table 3, we modified the original examples in order to add clients that were inlined and in order to seed errors. Below, we explain test configurations that contain the paths to the modified examples.

Test configurations

For each example in table 2 and table 3, we ran our Viper inlining tool once with a single test configuration or multiple times with different test configurations. A test configuration consists of three parameters: (1) the entry method to inline (i.e., the initial statement), (2) the inlining bound, and (3) seeded errors. In table 3, if the column “Inl. P.” has a green checkmark, then this means that inlining is preserving w.r.t. all considered test configurations. Analogously, if the columns “SC” (resp. “Str. C” have green checkmarks), then the semantic (resp. structural) condition holds for every test configuration. We established these checkmarks via manual analysis and then ran our tool to check whether the correct results were reported.

We have documented all test configurations in the following JSON files:

- VeriFast Table 2 test configurations:
  - ~/test_framework/viperserver/carbon/src/test/resources/inlining/verifast/test_config_verifast.json
- GRASShopper Table 2 test configurations:
  - ~/test_framework/viperserver/carbon/src/test/resources/inlining/grasshopper/test_config_grasshopper.json
- RSL Viper Table 2 test configurations:
  - ~/test_framework/viperserver/carbon/src/test/resources/inlining/rsl/test_config_rsl.json
- Nagini Table 2 test configurations:
  - ~/test_framework/viperserver/carbon/src/test/resources/inlining/nagini/test_config_nagini.json
- Table 3 test configurations:
  - ~/test_framework/viperserver/carbon/src/test/resources/inlining/syntactic/time/test_config_syntactic.json

Each test configuration is specified as a separate object and each test configuration is structured the same way. As an example taken from the Nagini test configurations:


```
{
  "file":"iap_bst.vpr",
  "bound":3,
  "entry":"client_error",
```

```
  "sc_holds":true,
  "errors":[{"tag": "assert.failed:assertion.false"}]
}
```

- “file” specifies the file path relative to where the JSON file is located
- “bound” specifies the inlining bound to be used
- “entry” specifies the entry method to inline (i.e., the initial statement)
- “sc_holds” specifies whether the structural condition holds for the given configuration
- “errors” specifies which kind of errors we expect (if “sc_holds” is false, the errors are irrelevant).
    - In the above example, we expect one error that has the tag “assert.failed:assertion.false” (Viper uses different tags for different kinds of errors)

Note that for the same example in Table 3, we sometimes use different files for different configurations, where different errors are seeded. For example, for iap_bst in Table 3, we use the files

- ~/test_framework/viperserver/carbon/src/test/resources/inlining/nagini/iap_bst.vpr
- ~/test_framework/viperserver/carbon/src/test/resources/inlining/nagini/iap_bst_error.vpr

Seeded errors in modified files

The test configurations specify whether we expect an error or not (in cases, where the structural condition is not expected to hold, the errors are irrelevant). In the files, where we expect errors we mostly seeded the errors ourselves. We often also added a short comment with the term “error”, used “client_error” for an erroneous client or used a Viper “ERROR” macro, where depending the value of “ERROR” an error is expected or not). In some cases, errors already existed in the original file (such as parkinson_recell.py where the original file contains “ExpectedError” annotations).

Note that for Nagini files, we seeded errors (or used existing errors) in the Python files. The test configuration for Nagini files points to the Viper encoding (since we apply our inlining tool on the Viper encoding), but for each Nagini Viper encoding there is a Python file with the same name in the same folder. For example:

~/test_framework/viperserver/carbon/src/test/resources/inlining/nagini/iap_bst.py is the Python file used to generate ~/test_framework/viperserver/carbon/src/test/resources/inlining/nagini/iap_bst.vpr (which is the Viper file mentioned in the test configuration.

Running the inlining tool on all test configurations to confirm table 2 and table 3

We developed a test framework that runs our inlining tool for each configuration separately and checks whether the inlining tool reports the correct results (that is, the inlining tool correctly reports whether the structural condition holds and if the structural conditions indeed holds, then the inlining tool reports the correct errors). The test framework is given by the Python file ~/test_framework/viperserver/carbon/src/test/inlining_test_framework/run_inlining_tests.py

The test framework relies on two already-existing components:

- ViperServer (https://github.com/viperproject/viperserver)
    - This is a HTTP server that manages Viper verification requests. By using ViperServer, one avoids the Java Virtual Machine startup times
    - We instantiate ViperServer with our inlining tool
- ViperClient (https://github.com/viperproject/viper_client)
    - A tool that communicates with ViperServer. We use ViperClient to obtain verification results of our inlining tool as JSON objects. This allows us to easily analyze verification results and verification times.

Run the following commands to run our test framework on all the test configurations

Start the Viper HTTP Server in one terminal by running:
> cd ~/test_framework/viperserver/
> sbt run

After at most a few minutes and around 25 lines of "[info]..." output, the following message should appear:

[info] ViperServer online at http://localhost:PORT

where PORT is some integer. This means the server can be reached on the port PORT (the port is not always the same).

Keep this terminal open and do not stop the process, since it is required to run the test framework.

Open another terminal and run (where PORT is given by the number outputted above and R is the number of times each configuration should be tested):

> cd ~/test_framework/viperserver/carbon/src/test/inlining_test_framework/
> python3 run_inlining_tests.py --port PORT --reps R --boogieExe $BOOGIE_EXE --z3Exe $Z3_EXE --dir ../resources/inlining/

For the results in table 2, we chose "R=5" (for the results in table 2, we used Windows and did not get the results inside a VM). In the VM "R=5" takes around 33 minutes. You could also choose "R=1", which takes around 6 minutes in VM

The expected output of the above "run_inlining_tests.py" command is given in "~/artifact/expected_output_test_framework.txt" (if the output is as expected, then this means the inlining tool always reports the correct result). Moreover, this command produces a CSV file that contains timing information (in milliseconds) for every configuration in

"~/test_framework/viperserver/carbon/src/test/resources/inlining/YYYYMMDD_HHMMSS_test_summary.csv"

where YYYYMMDD_HHMMSS provides the timestamp when the command was run (so, if you run the command multiple times, you will see multiple test_summary.csv files). Note that the times express the median and mean of all the repetitions performed (if R=1, then the median and mean are always the same).

To compute the timing information for table 2 and table 3, you can run the following command, which combines all test configurations that are associated with the same example and then takes the mean of all the times:

```
> python3 ~/artifact/print_table.py -s PATH_TO_CSV_FILE
```

The printed output shows the timing results for table 2 and table 3. The association between example names in the tables and the file names in the printed output is mostly straightforward except for:

- RelAcqDblMsgPassSplit_preserving.sil is msg_split_1
- RelAcqDblMsgPassSplit_nonpreserving.sil is msg_split_2
- RelAcqRustARCStronger_inline.sil is rust_arc
- problem1-split.vpr is lcp

The timing information computed in the VM was often larger than what we reported in table 3 (which was computed on a Windows machine and outside of a VM) but surprisingly also faster for some examples. We have stored the csv file that we obtained by running the test framework inside the VM in ~/artifact/timing_results_vm.csv (this does not include the 3 files that we added after the artifact evaluation).

There was one strong outlier in the VM: RSLLockNoSpin-not-in-appendix.sil took 108 seconds on average, while it only took 42 seconds on our Windows machine. We generally noticed large fluctuations between machines for this example in particular, which points to some instability in the axiomatization used by RSL-Viper for that example.

Other information in tables 2 and 3

We now give more information on how we obtained the numbers of other columns not yet discussed in tables 2 and 3.

Lines of code
For tables 2 and 3, we counted only non-empty lines of code that were part of the program (i.e., no comments) and we also ignored annotations. For Nagini, we considered the Python files (not the Viper files). For RSL-Viper, we ignored all the macro definitions and background theory.

Number of annotations
For Nagini, we considered the Python files (not the Viper files).

Spurious errors
To compute the spurious errors for an example in table 2 and table 3, we did the following for each configuration for that example (which already contains annotations that make the program verify modularly):

1. Remove all annotations except for the pre- and postcondition of the client
2. Verify the resulting program modularly and manually check which reported errors are spurious

For a single example, the reported spurious errors number is then given by the average of the number of spurious errors across all test configurations for that example.

For Nagini, we did the above described process for the Viper encodings.

4. Installing the tools outside the virtual machine

We recorded all the commands that we used to set up the dependencies of the virtual machine in the file located at "/home/inlining/inlining_tool/etc/vm/setup_vm.sh" in the virtual machine. It also available outside the virtual machine at: https://github.com/tdardinier/carbon/blob/b314c6e16d202139ad812670b681ecbbbb10cf3f/etc/vm/setup_vm.sh

Below we give some details on the main tools that we install. The full details on which commands were precisely executed are given in the setup_vm.sh file linked above.

Isabelle/HOL 2022

The proof assistant Isabelle can be easily downloaded and installed from https://isabelle.in.tum.de/installation.html. It can be installed on Linux, Windows 10/11, MacOS, and it is also available as a Docker image. In this document, we assume that Isabelle has been installed at the path *~/tools/Isabelle2022*, and that *~/tools/Isabelle2022/bin* is in the path.

Note that we have only tested our mechanization with the version 2022. Some proofs might fail with earlier versions.

VeriFast

We use in this artifact the version 21.04 of VeriFast (https://github.com/verifast/verifast/releases/tag/21.04).

On Ubuntu, it can be installed as follows:

```
> wget https://github.com/verifast/verifast/releases/download/21.04/verifast-21.04-linux.tar.gz
> tar xzf verifast-21.04-linux.tar.gz
```

After these commands, VeriFast should be available at *verifast-21.04/bin/verifast* (and the GUI at *verifast-21.04/bin/vfide*). Our artifact assumes that *verifast-21.04/bin* is in the path.

GRASShopper

Before installing GRASShopper, please install Z3 (see https://github.com/Z3Prover/z3).

We use, in this artifact, the commit *108473b0a678f0d93fffec6da2ad6bcdce5bddb9* from GRASShopper (https://github.com/wies/grasshopper). To clone this repository on Ubuntu:

```
> git clone https://github.com/wies/grasshopper
> cd grasshopper
> git checkout --quiet 108473b0a678f0d93fffec6da2ad6bcdce5bddb9
```

Next, we need to build GRASShopper. For this, we need to install Opam. On Ubuntu:

```
> bash -c "sh <(curl -fsSL
https://raw.githubusercontent.com/ocaml/opam/master/shell/install.sh)"
```

We can then build GRASShopper (from the folder *grasshopper* we cloned previously):

```
> opam switch 4.07.0
> opam install -y ocamlfind
> opam install -y ocamlbuild
> eval $(opam config env)
> ./build.sh
```

GRASShopper should now be available at *grasshopper/grasshopper.native*. Our artifact assumes that the folder *grasshopper* is in the path.

Nagini

We use Nagini version 0.8.5. It can be installed via pip. We use Python 3.7 to install Nagini (3.8 does not work for Nagini 0.8.5). Our setup_vm.sh file shows how to install Nagini using Python 3.7 on Ubuntu.