Igloo: Soundly Linking Compositional Refinement and Separation Logic for Distributed System Verification (Artifact)

Christoph Sprenger, Tobias Klenze, Marco Eilers, Felix Wolf, Peter Müller, Martin Clochard and David Basin

August 9, 2020

1 Getting Started

The artifact is a VirtualBox VM image that contains our proofs and case studies as well as the tools needed to check them (Isabelle 2020, VeriFast 19.12 and the most recent version of Nagini (commit 31030c1)). To run it, simply import it into an up-to-date version of VirtualBox (we tested with version 6.1.10) that has the VirtualBox extension pack installed. It uses 4GB of RAM and four logical cores by default; you may need to adjust these values if they are too high for your system.

The image contains an installation of Ubuntu 20.04. Both the user name and the password are “igloo”.

Isabelle, VeriFast and Nagini are all installed; to run them, open a terminal and execute, respectively, “/home/igloo/Isabelle2020/Isabelle2020” (for the Isabelle GUI), “isabelle” (for the Isabelle command line tool), “verifast” (for the VeriFast command line tool), or “nagini” (for the Nagini command line tool).

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

1. Isabelle: Start the Isabelle GUI by running “./Isabelle2020/Isabelle2020” from a terminal and open the file igloo-isabelle/Igloo.thy

2. VeriFast: Run

   ```
   cd igloo-java/leader-election-concurrent
   verifast -allow_assume -c src/impl/Leader.jarsrc
   ```

   in a terminal

3. Nagini: Run the following commands from a terminal:

```
cd igloo-python/leader-election
cd leader-election-python-verification
MYPYPATH=leader/stubs nagini --ignore-obligations
 ↪  leader/election.py
```

The first step will lead to Isabelle checking proofs for a while. The third step will take ca. a minute depending on your system.

2 Artifact Overview

The submitted version of our paper can be found on the desktop.

Our formalization and the case studies are located in the following three subdirectories of the "igloo" user's home directory:

- igloo-isabelle: Contains the formalization of our framework and the Isabelle/HOL part of our three case studies (in the subdirectory "case-studies").
- igloo-java: Contains the Java implementations (with proof annotations) of our three case studies.
- igloo-python: Contains the Python implementations (with proof annotations) of the first and third case study.

In the remainder of this document, all paths are relative to the home directory unless specified otherwise.

3 Claims

To summarize, our artifact supports the following claims made in the paper:

1. The Igloo framework and all theorems from the paper are formalized and checked in Isabelle. See Sec. 4.1.
2. We have implemented the Leader Election case study described in the paper. In particular,
   - We have proved correctness of an abstract model of the leader election and then refined it according to the Igloo methodology, resulting in an I/O specification. See Sec. 4.2.1.
   - We have a sequential and a concurrent Java implementation of the leader election, proved correct w.r.t. the I/O specification with VeriFast. See Sec. 4.2.2.
   - We have a sequential Python implementation, proved correct w.r.t. the I/O specification with Nagini. See Sec. 4.2.3.

- The Java implementations and the Python implementation are compatible with each other. See Sec. 4.2.4.
- The proofs in the different tools are linked via the common I/O specification; the specification used to verify the code is manually translated from the I/O specification defined in Isabelle. See Sec. 4.2.5.

3. We have implemented the Primary Backup case study described in the paper. In particular,

   - We have proved correctness of an abstract model and then refined it according to the Igloo methodology, resulting in I/O specifications for server and client. See Sec. 4.3.1.
   - We have a Java implementation of the primary backup with a sequential client and a concurrent server, proved correct w.r.t. the I/O specification with VeriFast. See Sec. 4.3.2.
   - As before, the specifications used to verify the code is a manual translation of the I/O specification defined in Isabelle. See Sec. 4.3.3.

4. We have implemented the Authentication case study described in the paper. In particular,

   - We have proved correctness of an abstract model and then refined it according to the Igloo methodology, resulting in I/O specifications for initiator and responder. See Sec. 4.4.1.
   - We have a sequential Java implementation of initiator and responder, proved correct w.r.t. their I/O specification with VeriFast. See Sec. 4.4.2.
   - We have a sequential Python implementation of initiator and responder, proved correct w.r.t. the I/O specification with Nagini. See Sec. 4.4.3.
   - As before, the specifications used to verify the code is a manual translation of the I/O specification defined in Isabelle. See Sec. 4.4.4.

We are not aware of any major claims made in the paper that are not supported by the artifact.

4 Step by Step Instructions

4.1 Isabelle Formalization of the Igloo Framework

The files document.pdf and outline.pdf in the 'generated' directory contain our formalization with and without proofs, respectively, as single PDF documents. A more convenient, and interactive way of browsing theories is to use the Isabelle GUI.

Isabelle/HOL has two main executables:

- Isabelle2020: This starts the default Isabelle GUI (based on jEdit). It is the most convenient way of browsing the theories, as it allows inspecting the proof state and using Control+Click to jump to definitions and lemmas.
- isabelle: This is a command line tool. You may also build the logic images corresponding to the main session by typing "isabelle build -v -b -D ." in the igloo-isabelle directory. Running this tool also shows the successful verification of our proofs. See also ROOT file.

Loading the theory Igloo.thy into Isabelle2020 will load the entire formalization (excluding the case studies). This theory file also lists all lemmas of the paper, and their corresponding formalizations in Isabelle. Do Ctrl-Click on one of the "thm" statements to jump to the file containing the proof.

Page 7 of generated/outline.pdf shows the dependency graph of the different Isabelle theories that we have developed (Pure, HOL and HOL-Library belong to the Isabelle package and are not developed by us; the three most abstract models of the authentication case study and some of their dependencies are adapted from an earlier paper). We now give a brief description of the theories, grouped into four categories:

Event Systems and infrastructure:

- Event_Systems: defines event systems as labeled transition systems, defines simulation relation and shows its soundness by proving that simulation implies trace inclusion.
- ENat: defines extended natural numbers (nats with infinity)
- EMultiset: defines multisets based on extended natural numbers
- Preliminaries: defines I/O actions, typing restriction, and I/O-guarded event systems

From Monolithic Event Systems to Components:

- Composition: defines composition of two event systems, and composition of two traces sets via product, restriction and relabel operators. Soundness of event system composition is shown w.r.t. trace composition.
- Decomposition: contains lemmas to relate a monolithic system with its decomposition, given as components and an environment.

- Interleaving: defines the interleaving composition of both pairs of event systems and event system families (i.e., parametrized event system) and shows trace equivalence with a single event system that simulates all components.
- Event_Composition: defines a new datatype events that distinguishes between internal events, and "real" IO events. Only the latter synchronize with the environment. The use of this theory is optional, and only the Primary-backup case study uses it.

From Components to I/O Processes:

- IO_Processes: defines I/O process codatatype, derives various choice operators over countable sets, defines operational semantics of processes and shows its properties.
- Event_Systems_into_IO_Processes: defines the actual embedding of I/O-guarded event systems into processes and shows that the event system is trace equivalent to the traces produced by the embedded process' operational semantics.

From I/O Processes to I/O Separation Logic:

- IO_Separation_Logic: defines syntax and semantics of separation logic I/O assertions, heap transitions, and traces of assertions.
- IO_Processes_into_IO_Separation_Logic: defines embedding of I/O processes into I/O separation logic, and canonical models
- IO_Behavior: shows the triple trace inclusion of I/O processes, their canonical heap models, and their embedding into I/O separation logic.

4.2 Leader Election Case Study

The leader election development is our first case study. As described in the paper, it consists of a model of the entire system defined in Isabelle, which we refine down to an I/O specification for a single node according to our methodology. We then have two Java implementations (one sequential, one concurrent) and one sequential Python implementation that have been verified to fulfill this specification using VeriFast and Nagini, respectively.

4.2.1 Inspecting and Checking the Isabelle Formalization

Loading the theory Leader_Election_4.thy from directory igloo-isabelle/case-studies/leader-election into Isabelle/HOL 2020 will load the entire case study, since it contains the most concrete formalism present in Isabelle – the I/O specification of Step 5.

As described in the paper, the abstract models are refined down to component I/O specifications in Isabelle/HOL. We shortly describe the files in igloo-isabelle/case-studies/leader-election:

- Leader_Election_0: Abstract model tr0. Specifies the problem, but not the protocol to solve it.
- Ring_network: defines the ring network topology that we assume.
- Leader_Election_1: Protocol model tr1. Most abstract model that implements the protocol. Refines tr0.
- Leader_Election_2: Protocol model tr2. Introduces internal buffers ibuf and obuf. Refines tr1.
- Leader_Election_3: Interface model tr3. Decomposition into system tr3s and environment tr3e and decomposition of system to individual components that are expressed as I/O-guarded event systems. tr3 is bisimilar to tr2. We show that the (re-composed and global) model tr3 satisfies the requirements (lemma tr3_satisfies_property).
- Leader_Election_4: We embed a system component tr3s (parametrized by the index parameter) into a process, and the process into I/O separation logic. In lemma trace_equivalence_tr3s_IOspec we show them to be trace equivalent. Finally, we unfold this I/O specification; this is shown in lemma P_unfolding.

The formalization is described in detail in the third chapter of the outline.pdf in igloo-isabelle/generated.

4.2.2 Inspecting, Verifying and Executing the Java Implementations

The Java implementations can be found in igloo-java/leader-election-sequential and igloo-java/leader-election-concurrent.

The structure of both Java implementations of the leader election is as follows:

- Leader.java: Implementation of the leader election, along with specification: Declaration of state type, internal I/O operations, I/O contract and initial state
- UDPSocketHelper.java: Trusted implementation of wrapper around socket library
- UDPSocketHelper.javaspec: Specification for UDPSocketHelper.java, declaration of socket I/O operations, message type definition

- java.net.javaspec: Stub file for used libraries
- java.nio.javaspec: Stub file for used libraries
- verifast.javaspec: Definition of (I/O) specification primitives (places, tokens, bigstar), datatypes (sets)

To re-verify the implementations, execute the following commands from igloo-java:

```
verifast -allow_assume -c
↪  leader-election-sequential/src/impl/Leader.jarsrc
```

and

```
verifast -allow_assume -c
↪  leader-election-concurrent/src/impl/Leader.jarsrc
```

To compile and execute the sequential Java implementation, execute the following commands from igloo-java/leader-election

```
cd leader-election-sequential/src
javac impl/*.java
java impl.Main OUT_HOST ID IN_PORT OUT_PORT
```

where OUT_HOST is the IP address of the next host in the ring, ID is the identifier of the local node, IN_PORT is the port on which the current node should listen for packets, and OUT_PORT is the port on which the next node in the ring is listening.

Since this will start a single node, and the leader election will only work with a ring of nodes, you need to start at least two nodes whose in- and out-ports form a ring. As an example, try running

```
java impl.Main 127.0.0.1 34 50000 50001
```

and

```
java impl.Main 127.0.0.1 23 50001 50000
```

in separate terminals to create a ring consisting of two nodes with IDs 34 and 23.

Compiling and executing the concurrent implementation works the same way, just go to the directory leader-election-concurrent/src in the first step.

For both versions, we have also defined an example run that will automatically start several nodes in a ring; to execute it, run

```
java impl.Main
```

4.2.3 Inspecting, Verifying and Executing the Python Implementation

Two versions of the Python implementation can be found in igloo-python/leader-election: The directory leader-election-python-verification contains the version of the files with specifications. The file contents are as follows:

- election.py: Implementation of the leader election
- int_socket.py: Trusted specification and implementation of wrapper around socket library, along with declaration of socket I/O operations, message type definition
- spec.py: Definitions of internal I/O operations, state type, and I/O specification of the implementation
- types.py: Definition of message type
- stubs/socket.pyi: Stub file for used library

The executable Python implementation, in which the specifications have been removed, can be found in leader-election/leader-election-python-executable. This version does not contain the files spec.py, types.py and the stub file. All other files are present and identical except that specifications have been commented out (Nagini currently does not do this automatically, so this step has been performend manually). In particular, this means that

1. All import statements for specification-only files have been commented out
2. All lines containing preconditions, postconditions, invariants or Assert statements have been commented out
3. All definitions of IO operations or specification-only data types have been commented out
4. All function parameters, local variables, and return types (as well as the corresponding variable assignments, arguments in calls, and return statements) that have specification-only-types have been commented out; in particular, this means that ghost variables containing the abstract system state (usually called “s”) and ghost parameters of type Place that are used for I/O specifications (usually called “t” or “tp”) have been removed.

We have commented out the specification parts instead of deleting them so that line numbers in both versions are identical and it is easy to see from a diff that the executed implementation itself has not been changed. The executable version additionally contains the file run_election, which parses command line arguments and then calls the verified main method in election.py.

To re-verify the Python implementation using Nagini, run:

```
cd igloo-python/leader-election
cd leader-election-python-verification
MYPYPATH=leader/stubs nagini --ignore-obligations
↪  leader/election.py
```

Verification should take between 20 seconds and 2 minutes depending on the system.

To execute the Python implementation, execute the following commands:

```
cd igloo-python/leader-election
cd leader-election-python-executable
PYTHONPATH=. python3 leader/run_election.py OUT_HOST ID
↪  IN_PORT OUT_PORT
```

where OUT_HOST is the IP address of the next host in the ring, ID is the identifier of the local node, IN_PORT is the port on which the current node should listen for packets, and OUT_PORT is the port on which the next node in the ring is listening.

Since this will start a single node, and the leader election will only work with a ring of nodes, you need to start at least two nodes whose in- and out-ports form a ring. As an example, try running

```
PYTHONPATH=. python3 leader/run_election.py 127.0.0.1 23 50000
↪  50001
```

and

```
PYTHONPATH=. python3 leader/run_election.py 127.0.0.1 34 50001
↪  50000
```

in separate terminals to create a ring consisting of two nodes with IDs 34 and 23.

4.2.4 Interoperability of Java and Python Implementations

As claimed in the paper, all three implementations of the leader election are compatible with each other. To test this, set up a ring that consists nodes running different implementations. For example, you can run the following commands in different terminals (from the directories specified above for each version):

```
PYTHONPATH=. python3 leader/run_election.py 127.0.0.1 34 50000
↪  50001
```

for the Python version and

```
java impl.Main 127.0.0.1 35 50001 50002
```

for the sequential Java version and

```
java impl.Main 127.0.0.1 36 50002 50000
```

for the concurrent Java version.

4.2.5 Comparing the I/O specification between Isabelle/HOL / VeriFast / Nagini

The connection point between the Isabelle/HOL model of the leader election and the implementations is the I/O specification that results from embedding the decomposed model into an I/O assertion, as described in the paper. This specification was manually translated from Isabelle/HOL syntax to VeriFast and Nagini syntax, respectively. To compare the three versions, consider

- the Isabelle/HOL version, defined in igloo-isabelle/case-studies/leader-election/Leader_Election_4.thy (lemma leader_election_node_iospec)
- the VeriFast version, defined in file Leader.java in igloo-java/leader-election-sequential/src/impl (predicate P, along with previous predicate_ctor definitions; the same definition exists in the concurrent version)
- the Nagini version, defined in igloo-python/leader-election/leader-election-python-verification/leader/spec.py (IOOperation P, along with previous IOOPerations; we have split the definition into its parts for convenience in the Nagini version)

While the translation is purely syntactical, the syntactical differences are quite large in places. We provide a comparison of the syntax of different important constructs in the file SYNTAX stored on the desktop of the virtual machine; this file also shows the different syntax of some important specification constructs in VeriFast and Nagini.

4.3 Primary Backup Case Study

For our second case study (primary backup), we have an Isabelle formalization of the system as well as verified Java implementations of the client (sequential) and server (concurrent). The main steps for inspecting the proofs and verifying and running the implementation are similar; here, we therefore only give a short description of the steps that are different.

4.3.1 Inspecting and Checking the Isabelle Formalization

The Isabelle formalizaton of the primary backup case study can be inspected by opening case-studies/replication/Primary_Backup_3.thy and the files it includes as dependencies in the Isabelle GUI.

Its contents are described in detail in Chapter 4 of the outline.pdf in igloo-isabelle/generated.

4.3.2 Inspecting, Verifying and Executing the Java Implementation

To re-verify the Java implementation of the primary backup case study, run the following comand from igloo-java:

```
verifast -allow_assume -c primary-backup/src/impl/Main.jarsrc
```

This verifies both server and client of the primary backup case study.

To compile it, run

```
cd primary-backup/src
javac impl/*.java
```

Since a proper run of the implementation requires running multiple nodes and simulating failures, we have prepared a pre-configured example run that does this. To execute it, run

```
java impl.Main
```

4.3.3 Comparing the I/O specification between Isabelle/HOL / VeriFast / Nagini

As before, the I/O specification we manually translated from Isabelle/HOL syntax to VeriFast syntax serves as the connection point between both tools, as described in the paper.

To compare the two versions, consider

- the Isabelle/HOL version, defined in file Primary_Backup_3.thy in igloo-isabelle/case-studies/replication (lemma m3s_iospec_ord for the server and m3c_iospec_ord for the client)
- the VeriFast version, defined in file data.javaspec in igloo-java/primary-backup/src/impl/theory (predicate ms_iospec_ord for the server and mc_iospec_ord for the client, along with previous predicate_ctor definitions)

4.4 Authentication Case Study

Finally, for our third case study (authentication), we have an Isabelle formalization of the system as well as verified Java and Python implementations of the initiator and the responder. Note that in this case, the Python implementation is not compatible with the Java implementation (and the paper does not claim that they are) because the implementations use different crypto libraries.

The main steps for inspecting the proofs and verifying and running the implementation are again similar to the previous case studies; here, we therefore only give a short description of the steps that are different.

4.4.1 Inspecting and Checking the Isabelle Formalization

The Isabelle formalizaton of the primary backup case study can be inspected by opening case-studies/security-protocols/m6_sig.thy and the files it includes as dependencies in the Isabelle GUI.

Its contents are described in detail in Chapter 6 of the outline.pdf in igloo-isabelle/generated.

4.4.2 Inspecting, Verifying and Executing the Java Implementation

To re-verify the Java implementation of the authentication case study, run the following command from igloo-java:

```
verifast -allow_assume -c authentication/src/impl/Main.jarsrc
```

This verifies both initiator and responder.

To compile it, run

```
cd authentication/src
javac impl/*.java
```

To run the responder, execute

```
java impl.Main Resp privK
```

where privK is the private key of the responder. For the initiator, execute

```
java impl.Main Init addrB pubkB
```

where pubkB is the public key of the responder and addrB is the IP address of the responder. Note that the responder has to be started first, and the initiator needs to start within nine seconds of the responder to avoid a timeout.

To generate valid key pairs, you can call

```
java impl.Main Gen
```

which will generate and then print a private and public key, in that order, separated by a newline.

As an example, try running

```
java impl.Main Resp MIIEvQIBADANBgkqhkiG9w0BAQEFAASCBKcwggSjA⌟
 ↪   gEAAoIBAQDR+pkmGXk0f6YZI76d4hn0VMG9xEJttz9s4bF75fBE3xSRUR⌟
 ↪   1ftDgjwM0vR16AWiQbGx6bGS+R9zVl6XYgur11Yvx0gA6H4CTXlBhliIp⌟
 ↪   aH87R6E15ukn3wNMCQXKfBGkKyUvY0LJAZPkeWUheu1mheaFRO0cSSoVX⌟
 ↪   A6TFbTR92U1mY2+JHCWyfK9A82oZFivWCCnckXZjoIOBXbyCxHG/4RDgb⌟
 ↪   XjVmd14JGITH7bhLsqyEYrLqj2MNx6oXPQUgRhMmtA7nvfpoB+hQU7pQM⌟
 ↪   ILmRu1c5Cyoy41t1LfMJCZIc2NrYAfduGBe+GmPoMoyGFcL8yZwHqSUMT⌟
 ↪   HUQZgDW/dAgMBAAECggEAZBuymjx4v5XSDZhCD6m1MAyclamU9hPwdCuh⌟
 ↪   32z/wQYOGz3t4CvevATyBoXjIxRWtGmYjN9UE6YlWiIVBIOjQezgV1isF⌟
 ↪   NGjHvhnLTkMpO0XQHIdRdSv4SOYNjIXwIJsxgy0UXkmEpdRQcvV0fzePa⌟
 ↪   Lsavr05l6K9bIyA0IOTZTIMLIdD4GeM6LJtVokSyrk0dCkm6+J1FIycf7⌟
 ↪   tmej1RXBC5iBMz6ahFBm3vANH7xMXUk6H57rtqB7Usi+BPQYywyTdbIqy⌟
 ↪   Z25EcNcqjZ3bvLC5ILGwaXbALvBlicd7iPnDGa3+xpDhPsWABFpusffK6⌟
 ↪   cccsPkE3eUFxYHIKooMbQKBgQDt+FKyoGW0NJUFiWtobcL1ABg0Uy1QEk⌟
 ↪   +G/EMnvEeMcxJkEQPuATcNMQMLbUTDHY0Bi5mkoKzC3kAKtRAlUCTpbxO⌟
 ↪   P+7NIuU8OVsUP0DQBGW+mVj8QGs2+fzw3k60Kvb8Rc62jmCYIFa8jnFYJ⌟
 ↪   hMObzeFC7Ky1KJtpjHdXzL7OtwKBgQDh41u7uV/VkBYCe2sN5CCz2gOJp⌟
 ↪   PwuN5bR5ZGd65+iRG3s+AZe+KNFx8RSN64BDxvKAd67Sko7Ib/AAGhd03⌟
 ↪   s6sZUcAeFBM2AepSn3TTixB7CwyU9Z1mWCrLfISX9/VNMSmD/Y86Jbn7T⌟
 ↪   cTlW/uy9xhUcNr56ZlFS4EF8lPIviCwKBgDBA3zv7TEQrOWCDCfWF9DdL⌟
 ↪   ypyplRGcdOXRNyjSg8uV9c+2p45WTrxtCMoDYSMTVelPTltUfVOST3gcW⌟
 ↪   ObIBoQTEutqRWNkuAQXUiQvuqvSZZJcALOaS8fp8uLuHfOEoD7Arx/yFR⌟
 ↪   wkkXvuEoAhnKx2Jcw1Q5wEXOcdJYH3MWWnAoGBAIj61KAME99wFsi+ivb⌟
 ↪   LhkFKTabkk8B7GUyDiEBZqF5AXOC8rzBcWrZwI88v0Kb3wIRJigXNUSJ2⌟
 ↪   ns8R8DgljK7VDXUEdtKRExLCWaaL/3rrDOzHxwTVjI0nq/MbDuPqTm0SQ⌟
 ↪   nWPmL8zI/wMzNcN7gFLLwFPpD/BwXY1B12PT8xtAoGADQtSbZmaseGSCN⌟
 ↪   T7jduDBJb/DVJ/Ddu4MuKH35+0uU3ilK8JMJTFs9uGVVXYROmWJgR3Z2y⌟
 ↪   zvNwCOsVscjPndtisoi/yg5KdJCiQzRrWHWOiKqrpzN5TsmuT8ccsPzY6⌟
 ↪   snkEqEKrEDiEQMGo05nAEy1YJNF/FqroD4mKRgJeuXg=
```

and

```
java impl.Main Init ::1
 ↪   MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEA0fqZJhl5Dn+mG⌟
 ↪   SO+neIZ9FTBvcRCbbc/bOGxe+XwRN8UkVEdX7Q4I8DDr0degFokGxsemx⌟
 ↪   kvkfc1Zel2ILq9dWL8ToAOh+Ak15QYZYiKWh/O0ehNebpJ98DTAkFynwR⌟
 ↪   pCslL2NCyQGT5HllIXrtZoXmhUTtHEkqFVw0kxWO0fdlNZmNviRwlsnyv⌟
 ↪   QPNqGRYr1ggp3JF2Y6CDgV28gsRxv+EQ4G141ZndeCRiEx+24S7KshGKy⌟
 ↪   6o9jDceqFz0FIEYTJrQO5736aAfoUFO6UDCC5kbtXOQsqMuNbdS3zCQmS⌟
 ↪   HNja2AH3bhgXvhpj6DKMhhXC/MmcB6klDEx1EGYA1v3QIDAQAB
```

In case copying this from a PDF is problematic, we have also written down these parameters in the file authentication_example_calls.txt, which can be found on the desktop in the VM.

Alternatively, we have again prepared a pre-configured example run; to run it, execute

```
java impl.Main
```

4.4.3 Inspecting, Verifying and Executing the Python Implementation

Two versions of the Python implementation with specifications can be found in igloo-python/authentication. As in the first case study, directory authentication-python-verification contains the version with proof annotations.

Again, there is a separate version in which we have removed proof annotations and added a run script for parsing command line arguments in the directory igloo-python/authentication/authentication-python-executable.

To re-verify the Python implementation using Nagini, run:

```
cd igloo-python/authentication
cd authentication-python-verification
MYPYPATH=authentication/stubs nagini --ignore-obligations
↪  authentication/initiator.py
MYPYPATH=authentication/stubs nagini --ignore-obligations
↪  authentication/responder.py
```

Verification should take between 20 seconds and 2 minutes depending on the system for each of the files (initiator and responder).

To execute the Python implementation, execute the following commands from igloo-python/authentication/authentication-python-executable:

```
PYTHONPATH=. python3 authentication/run_responder.py B privkB
```

and subsequently in a separate terminal

```
PYTHONPATH=. python3 authentication/run_initiator.py A B pubkB
↪  addrB
```

where A and B are integer identifiers of the initiator and responder, addrB is the IP address of the responder, and privkB and pubkB are the signature and verification key of B, respectively, as generated by PyNaCL, in Hex format. Note that it is important that the responder is started first.

Example calls with valid keys are

```
PYTHONPATH=. python3 authentication/run_responder.py 2
↪  e1f169afd23d9e9fc1a9ffe929a8590eccc427eabfe9fc1bcc60eaa87159ec98
```

and

```
PYTHONPATH=. python3 authentication/run_initiator.py 1 2
↪  01f0f7ae5fbf0f40061539b45313b8a2b0d4014dcd4c6e5f4a4e9d3daf9b0c7f
↪  127.0.0.1
```

In case copying this from a PDF is problematic, we have also written down these parameters in the file authentication_example_calls.txt, which can be found on the desktop in the VM.

4.4.4 Comparing the I/O specification between Isabelle/HOL / VeriFast / Nagini

As before, the I/O specification we manually translated from Isabelle/HOL syntax to VeriFast and Nagini syntax serves as the connection point between the tools, as described in the paper.

To compare the three versions, consider

- the Isabelle/HOL version, defined in igloo-isabelle/case-studies/security-protocols/m6_sig.thy (lemma m6i_iospec_ord for the initiator and m6r_iospec_ord for the responder)
- the VeriFast version, defined in file data.javaspec in directory igloo-java/authentication/src/impl/theory (predicate m6i_iospec_ord for the initiator and m6r_iospec_ord for the responder, along with previous predicate_ctor definitions)
- the Nagini version, defined in igloo-python/authentication/authentication-python-verification/authentication/spec.py (IOOperation m6i_iospec_ord for the initiator and m6r_iospec_ord for the responder)