18-732, Spring 2022 *Bryan Parno*

Assignment 3: Verification Practice

Version No.: 1.0 *Due: April 22, 2022, 9:00PM EDT*

This assignment consists of two (2) parts. In this assignment you will:

- **part 1:** practice using the Dafny language and verification toolset to explore verification of properties of programs.
- **part 2:** use Dafny to reason about the semantics, type-safety, and security of MiniC.

You will complete this homework as part of a team. See Section 4 for more details.

1 Dafny Basics (30 points)

You will need to use Dafny for the rest of the assignment.

Documentation If you haven't already, we suggest that you start by completing the official Dafny guide `https://rise4fun.com/Dafny/tutorial/guide`, as well as the following tutorials (linked at the bottom of the guide):

1. Termination: `https://dafny-lang.github.io/dafny/OnlineTutorial/Termination`
2. Sequences: `https://dafny-lang.github.io/dafny/OnlineTutorial/Sequences`
3. Value Types: `https://dafny-lang.github.io/dafny/OnlineTutorial/ValueTypes`
4. Lemmas: `https://dafny-lang.github.io/dafny/OnlineTutorial/Lemmas`

Additional extensive documentation is available online: `https://dafny-lang.github.io/dafny/`

Note that Dafny continues to evolve and the documentation does not always keep with it, so don't be surprised if there are a few discrepancies along the way.

1.1 Primality test (10 points)

Several cryptographic protocols have prime number generation as one of their steps. Several algorithms exist to efficiently generate very large prime numbers. The main component of these algorithms is the primality test procedure: an algorithm that, given a positive number, decides whether it is prime. In this problem you will implement a simple and inefficient version of the primality test.

Your task

The source code for this problem can be found in the `prime.dfy` file of the handout. You are given a specification of the `prime` predicate, which returns true if and only if the integer given as an argument is prime. You need to write the method `IsPrime`, which should return true if and only if the argument is prime.

Note: We recommend that you do not try to implement an "efficient" version of the primality test, as it would quite hard to convince Dafny that it satisfies the post-condition.

1.2 Bridge Controller (30 points)

U.S. intelligence agencies have learned that the government of Blackhattia intends to destabilize the United States by employing hackers to find and exploit vulnerabilities in the traffic engineering algorithms that manage traffic lights, dynamic road signs, and other transportation infrastructure, leading to chaos and crashes. Your group has been hired as consultants by the Department of Transportation to stop this from happening. Your assignment is to develop and verify an algorithm for correctly managing traffic lights at either end of an infrastructure-critical one-lane bridge, according to the following specifications.

Bridge controller specification

- At each end of the bridge (end A and end B), there is a traffic light that can be either red or green. The two lights can never be green at the same time.
- At each end of the bridge, there is also a sensor that can detect the number of cars currently waiting to cross. These can be modeled with counters W_a and W_b for the two ends.
- At any time, a car traveling from A to B is either waiting at A or has passed across the bridge. During a given “clock tick," the following actions happen atomically: If necessary, the lights change, and one car from one end may cross, decrementing the appropriate counter W_a or W_b. Time spent traveling across the bridge is not modeled, and there is no need to keep track of cars once they cross the bridge.
- Once a car is waiting at one end to use the bridge, it continues waiting until it crosses.
- Both lights start out red, with no cars at either end.
- If there is at least one car waiting at A and no cars at B, the light at A turns green (and vice versa for end B).
- If both lights are red and cars arrive simultaneously at both ends, the light at A turns green first.
- If the light at A is green and there are cars waiting at B, no more than 5 cars may travel A to B from that time point. When either no more cars are waiting at A **or** 5 cars have crossed from A to B, the light at A turns red and the light at B turns green. If the light at A is green and no cars are waiting at B, then the light at A may stay green until a car arrives at B, from which time the 5-car limit is imposed. (And vice versa for end B). In short, this means that if any cars are waiting at one end of the bridge, and that end has a red light, that light will turn green no more than 5 clock ticks later.

Your task

The source code for this problem can be found in the `bridge.dfy` file of the handout. You are given an implementation of this bridge controller specification in Dafny. You need to specify the appropriate post- and pre-conditions for the various methods (20 points). In addition, you will also need to fill in the implementation of the predicate `Valid` (10 points). The `Valid` predicate must be strong enough to uphold the specification, and weak enough to describe what is a valid state before and after each method individually, not at just the outermost `Tick` method.

2 MiniC in Dafny (87 points)

In this section of the assignment, you will get your hands dirty working with the semantics, type-safety, and security of MiniC, the small language discussed in class. We will also tie all of these concepts back to the topic of taint analysis that we learned about in the first part of the semester.

2.1 MiniC: Semantics (20 points)

In this task, you will fill in some gaps in the semantics for MiniC and complete the proof of type safety, showing that well typed MiniC programs will not fail.

In `def.dfy`, you will find the definitions for MiniC Abstract Syntax Trees (ASTs), the semantics for MiniC, and the typing rules for MiniC, all of which elaborate on the definitions discussed in class. However, some important pieces are missing! Hence, when you first open the file (or run Dafny on it from the command line), you will see lots of errors. As you work through the file, you may want to comment out the code below the point where you are working, so that you are not flooded with errors related to the portion of the code you have not yet filled in/fixed up.

1. **Semantic Definitions.** In the definition of `EvalCommand`, fill in the correct semantics for the cases indicated with "TODO" (`IfThenElse, PrintS, PrintE, GetSecretInt`). The semantics must match those defined mathematically (as inference rules) in lecture.

   You can sanity check your work by confirming that the assertions in `CommandExamples` all succeed, given your definitions. Do not change the assertions or add extra annotations there, as they will be discarded by AutoLab.
2. **Type Checking.** In the definition of `ExprHasType`, fill in the correct check for the cases indicated with "TODO" (`BinaryOp`). The checks must match those defined mathematically (as inference rules) in lecture.

   In the definition of `CommandWellTyped`, fill in the correct checks for the cases indicated with "TODO" (`IfThenElse, PrintS, PrintE, GetSecretInt`) – these are the same cases you defined semantics for above. The checks must match those defined mathematically (as inference rules) in lecture.

   You can sanity check your work by confirming that the assertions in `TypeExamples` all succeed, given your definitions. Do not change the assertions or add extra annotations there, as they will be discarded by AutoLab.

Now that we have a complete set of semantic definitions and type checks, we need to prove type safety. In the file `proofs.dfy`, you can find a partial proof of type safety. Your job is to complete the proof by filling in the necessary steps for the semantic definitions you added above.

1. **Proof of Type Safety.** The `WellTypedExprSuccess` should pass automatically, if you have defined the correct checks in `ExprHasType` above. If it does not, you should revisit your checks.

   In the definition of `WellTypedCommandSuccess`, fill in the necessary steps so the lemma verifies. You should only need to make changes to the cases you defined above (i.e., `IfThenElse`, `PrintS`, `PrintE, GetSecretInt`), as indicated with "TODO". Depending on how you defined your semantics and type checks, some of these proof steps may go through automatically; that's perfectly okay.

   Do not change the signature of the lemmas in this file. As noted in the file's comments, only the lemma bodies (containing your improved code) will be consumed by AutoLab; so any changes you make to the signatures or other parts of the file will be ignored, likely causing confusion when you receive the results.

2.2 MiniC: Security Types (20 points)

(2 points) As a warm up, add the necessary security labels to the variable declarations in `sec-types.mc` so that the file will pass our security type checker. Once you finish constructing the type checker below, you can check whether your labels are correct. See the CODE.md file in the handout for instructions on how to run the checker. **DO NOT CHANGE THE PROGRAM**, aside from adding the security labels to the declarations.

In the rest of this task (18 points), you will fill in some gaps in our security type system.

In `security-types.dfy`, you will find the definitions for our security type system, and a partial proof that it ensures non-interference.

1. **Security Types.** In the definition of `ExprHasSecType`, fill in the correct checks for the cases indicated with "TODO" (`BinaryOp`).
   In the definition of `CommandHasSecTypeBasic`, fill in the correct checks for the cases indicated with "TODO" (`IfThenElse, PrintS, PrintE, GetSecretInt`). The semantics must match those defined mathematically (as inference rules) in lecture.
   You can sanity check your work by confirming that the assertions in `NIexamples` all succeed, given your definitions.
2. **Non-Interference Proof.** Now that we have defined our type system, we need to prove that it ensures non-interference. Start by filling in the cases necessary for `NonInterfenceTypeExpr` to verify (some cases may go through automatically).
   Now update the missing cases (indicated with "TODO") in the `HighCommandPreservesLowVars`, `HighCommandPreservesPubIO`, and `NonInterferenceTypesInternal`, so that the proofs all go through.
   As before, do not change anything other than the bodies of the lemmas; changes to lemma signatures will be discarded and cause you confusion.

2.3 MiniC: Dynamic Taint Analysis (32 points)

In this task, you will bring together the semantics of MiniC, the notion of non-interference, and a key idea from early in the course: dynamic taint tracking. Recall that dynamic taint tracking is a run-time technique that instruments a program to taint data that enters the system, and then tracks how the tainted data propagates through the program's execution.

In this task, you will complete the implementation of a basic dynamic taint engine for MiniC that will ensure that no value derived from a secret is ever printed to the screen. You will also prove that the engine does its job properly, i.e., that it ensures a non-interference property similar to that of our security type system.

In `taint-tracking.dfy`, you will find a partial implementation and a partial proof. You will need to complete both.

1. **Taint Tracking.** In the definition of `EvalExprTaint`, fill in the correct taint-propagation and value calculations for the cases indicated with "TODO" (`BinaryOp`). You should implement a basic, conservative taint policy when calculating the taint of the result; i.e., the result is tainted if either of the inputs is tainted.

In the definition of `EvalCommandTaint`, fill in the correct checks and taint-propagation steps for `IfThenElse`. Our policy is as follows: If the branch expression is tainted, then the PC should be considered tainted only for the duration of its execution.
You should also update `PrintS` and `PrintE` to prevent leaks!
You can sanity check your work by confirming that the assertions in `TaintExamples` all succeed, given your definitions.

2. **Non-Interference Proof.** Now that we have defined our taint-analysis engine, we need to prove that it ensures non-interference. Start by filling in the cases necessary for `NonInterfenceExpr` to verify (some cases may go through automatically).
Now update the missing cases (indicated with "TODO") in the `TaintedPcPreservesLowVarsPubIO`, and `NonInterferenceInternal` lemmas, so that the proofs all go through.
As before, do not change anything other than the bodies of the lemmas; changes to lemma signatures will be discarded and cause you confusion.

2.4 Static vs. Dynamic Non-Interference (15 points)

Consider the MiniC program below (which is also available in the starter files as `static-vs-dynamic.mc`).

```
decl
  a:int; b:int; c:int; z:int;
begin
  a := get_int();
  b := get_secret_int();
  c := get_secret_int();

  if a <= 0 {
    b := b + 1;
    c := 0 * b;
    z := c;
  } else {
    b := 0 * c;
    c := c + 1;
    z := b;
  }

  print_expr z;
end
```

Your task

For this task, write your answers in the `README.md` file included with the starter files.

(5 points) Without changing the code, is there any assignment of security labels to the variables (a, b, c, z) that will allow this program to securely type check?

- If so, provide the necessary labels, state which label (i.e., Low or High) the program will type check at, and explain how the relevant security typing rules will enable a successful type check. Hint: You can test your labels by actually running the security type checker on the program! You still need to explain why the type checking works, however.

- If the program cannot be type checked in our security type system, explain why not (1-2 paragraphs), again, referring to the relevant typing rules.

(10 points) If you run this program through the current version of our taint checker, it will report a leak. Improve the taint checker (**hint**: focus on `EvalExprTaint`) so that (a) this program and others like it run successfully, without reporting a leak, and (b) all of the proofs of non-interference still hold.

Summarize, in 1-2 paragraphs, what you changed and why the proofs still hold.

(As a side note, this illustrates one of the benefits of verification: You can optimize your code, knowing that the verifier will catch any mistakes you might introduce via the optimizations.)

3 Extra Credit (20 points)

Our Dafny development formally proves that our security type system enforces non-interference. We prove something similar for our taint-analysis engine. Assuming that we trust Dafny to check our proofs accurately, this is a very strong, mechanically-checked guarantee! Nonetheless, all proofs come with limitations, and ours are no different. In this exercise, your goal is to exploit these limitations.

IMPORTANT: You must employ *different* attacks in the tasks below. If you employ the same (or essentially similar) attacks in both, then you will receive credit for only one.

For each task, in the `README.md` file included in the starter files, add a write up that describes why your program passes the security checks, how it leaks secret information, and what limitation of the non-interference theorem your attack exploits.

3.1 Security Type Leak (10 points)

Your goal is to craft a MiniC program that will pass our security type checker but that will leak information about its secrets. To simplify the task, the program will read in a single secret integer, and your goal is to leak whether that integer is zero, positive, or negative.

In the handout files, you will find a set of starter files named `type-leak*`. You will put your leaky implementation in `type-leak.mc`.

You'll find that `type-leak-check.py` already contains infrastructure for invoking the security type checker on `type-leak.mc` and collecting the result of this invocation. **DO NOT CHANGE THIS CODE!** Any changes you make will be discarded by AutoLab and hence lead to results that will confuse you.

Implement your attack in `type-leak-attack.py`, which is imported by `type-leak-check.py`. See the included comments for more details.

3.2 Taint Analysis Leak (10 points)

Your goal is to craft a MiniC program that will runs in our taint-analysis engine but that will still leak information about its secrets. To simplify the task, the program will read in a single secret integer, and your goal is to leak whether that integer is zero, positive, or negative.

In the handout files, you will find a set of starter files named `taint-leak*`. You will put your leaky implementation in `taint-leak.mc`.

You'll find that `taint-leak-check.py` already contains infrastructure for invoking the taint-analysis on `taint-leak.mc` and collecting the result of this invocation. **DO NOT CHANGE THIS CODE!** Any changes you make will be discarded by AutoLab and hence lead to results that will confuse you.

Implement your attack in `taint-leak-attack.py`, which is imported by `taint-leak-check.py`. See the included comments for more details.

4 Submission and Grading

For both Parts 1 and 2, submit your solution to the class' **Autolab**. Follow the instructions on Piazza for how to register your team for Autolab.

To create the `handin.tar` file that you will upload to Autolab, from the base directory in the starter code, run:

```
make handin.tar
```

This will bundles up all of these files:

- `prime.dfy` — your modified file for the primality test problem.
- `bridge.dfy` — your modified file for the bridge controller problem.
- `def.dfy` — your modified file with MiniC semantics.
- `proofs.dfy` — your modified file with proofs of type safety.
- `security-types.dfy` — your modified file for security types and non-interference proof.
- `taint-tracking.dfy` — your modified file for taint tracking and non-interference proof.
- `README.md` — your written answers for the static vs. dynamic non-interference problem.
- OPTIONAL: `type-leak.mc` and `type_leak_attack.py` — the modified files used to implement the extra credit attack from section 3.1.
- OPTIONAL: `taint-leak.mc` and `taint_leak_attack.py` — the modified files used to implement the extra credit attack from section 3.2.

Your grade for this part will be directly correlated to the score reported by AutoLab.