Formal Methods in Security and Privacy (SS20) Prof. Matteo Maffei

Project 3 – minnieThor

Due | *24.07.2020* TU Wien

Setting up Your System for the Project

For this project we will provide you with a virtual Linux machine where the required dependencies (in particular a version of `z3`) are already installed. We also prepared for you the parsing infrastructure for *tinyEVM* smart contracts that makes contract information available via a *HoRSt* interface.

For accessing the virtual machine please first download VirtualBox for your system:

`https://www.virtualbox.org`

Now follow these steps:

1. Download the virtual machine (VM) image from TUWEL.
2. Import the VM image in VirtualBox.
3. Start the imported VM in VirtualBox.
4. Log in to the VM (`user: horst`, no password).

In principle you are now able to execute *minnieThor* via the command line:

```
minniethor --help
```

Since it might be more convenient for you to work in your native system we configured the VM such that you can access it via SSH. To do so, please proceed as follows.

1. Within the VM access the VMs IP address by executing:

   ```
   ip addr show enp0s8
   ```

   The IP address of the VM is listed in the line starting with `inet`.

2. On your host machine you can connect to the VM using the following command:

   ```
   ssh horst@<VM IP address>
   ```

3. To exchange files between your host machine and the VM you can use the `scp` command.

   ```
   scp <file> horst@<VM IP address>:~
   ```

   would copy the file < *file* > from your host machine to the VM's home directory.

For Linux users, the described steps should work smoothly. For Mac and Windows users however the following needs to be taken into account

- Installing VirtualBox on Mac might cause some troubles. If you should run into issues, please follow the steps described in this article:

  `https://medium.com/@DMeechan/fixing-the-installation-failed-virtualbox-error-on-mac-high-sierra-7c421362b5b5`

- The network adapter needed for accessing the VM might not be created or set automatically. This will cause troubles when starting the VM. To solve this issue

  1. Open *File* → *Host Network Manager*.
  2. Click *Create* to create a new network adapter (which will be called *vboxnet0*).
  3. Close the Host Network manager and open the settings of our VM.
  4. Select the tab *Network* and there again *Adapter 2*.
  5. Tick *Enable Network Adapter* and make sure the *Attached to:* item is set to *Host-only Adapter*.
  6. Select under *Name:* the just created *vboxnet0* adapter that should now be available in the drop-down menu.

On Windows, the network adapter should be created automatically, but might not be set. It should be sufficient to follow steps 3) to 6).

For further information, please have a look at the tutorial videos for HoRSt where we walk you through the set-up of the project including the technicalities for the usage with Macs.

General Remarks

- Before you start with the project read the submission instructions carefully. Since there will often be hints at the end of the exercises, please read the *whole* exercise before starting it.
- Be neat and hand in readable (possibly annotated) *HoRSt* and *tinyEVM* bytecode files since we might need to manually check them. Comments in *HoRSt* files or *tinyEVM* bytecode files can either be written as line comments (`// this is a line comment`) or as multi-line comments (`/* this is a multi-line comment */`).
- Please use the file names specified in the exercise descriptions and the submission instructions.
- The exercises of this project are divided into *tasks* and *questions*:
  - Tasks refer to the practical part of the project and shall be provided as `.txt` files (either *HoRSt* specifications or *tinyEVM* bytecode contracts). The file `abstract-semantics.txt` already contains the selector function interface and some preliminary definitions.
  - Questions refer to the theoretical part of the project and should be answered in the file `report.pdf`. You will find questions in most of the following exercises. Please make sure to answer all these questions and insert explanations where required in the `report.pdf` file. We emphasize that it is inherently necessary to answer the questions and fill explanations in order to receive full points.

Project Overview

The goal of this project is to build *minnieThor*, a sound static analyzer for the *tinyEVM* bytecode language that was presented in the lecture. Of course we do not ask you to write such an analyzer from scratch, but you will make use of our generic analysis framework *HoRSt* that allows for giving high-level logical specifications of the abstract semantics in terms of Horn clauses. We already prepared infrastructure for parsing *tinyEVM* bytecode contracts such that contract information will be available by the following selector function interface:

```
1 sel pcs: unit -> [int];
2 sel pcsForOpcode: int -> [int];
3 sel argumentForPc: int -> [int];
```

The selector function `pcs` returns the set of all positions of the program. The selector function `pcsForOpcode` returns for the integer encoding of an opcode the set of program counters at which the specified opcode occurs in the program. Finally, the selector function `argumentsForPc` returns for a program position the singleton set containing the argument to the opcode at the specified position. This function can only be invoked on such program positions that contain a a PUSH, a JUMP, or a JUMPI instruction, since these are the only opcodes carrying arguments. When being invoked on another program positions, `argumentsForPc` throws an exception.

You can use these selector functions in your *HoRSt* analysis specification to parametrize the logical rules that constitute the abstract semantics of *tinyEVM* bytecode. For getting started with the initial setup and to get a short introduction into the *HoRSt* language and how it is used to write a Horn clause based abstract semantic, please have a look at our online tutorial.

Even though *minnieThor* will be (such as *eThor*) a general analyzer for reachability properties, we will in this project mainly focus on the single-entrancy property of smart contracts. As discussed in the lecture, the single-entrancy property shall prevent a contract from being reentered during execution. We will in the following give a short summary and motivation of this property.

Checking Single-entrancy

Single-entrancy is motivated by *reentrancy bugs*, which were also presented in the lecture. The most famous representative of this class is the so called DAO bug that led to a loss of 60 million dollars in June 2016. In an attack exploiting this bug, the affected contract was drained out of money by subsequently reentering it and performing transactions to the attacker on behalf of the contract. The cause of such bugs mostly roots in the developer's misunderstanding of the semantics of Solidity's [1] call primitives. In general, calling a contract can invoke two kinds of actions: Transferring Ether to the contract's account or executing (parts of) a contract's code. In particular, Solidity's `call` construct (being translated to a CALL instruction in EVM bytecode) invokes the execution of a fraction of the callee's code – specified in the so called *fallback function*. In Solidity, a contract's fallback function is written as a function without names or argument as depicted in the `Mallory` contract in Figure 1b. Consequently, when using the `call` construct the developer may expect an atomic value transfer where potentially another contract's code is executed. For illustrating how to exploit this sort of bug, we consider the contracts in Figure 1. The function `ping` of contract `Bob` sends an amount of 2 (sub)coins to the address specified in the argument. However, this should only be possible once, which is potentially ensured by the `sent` variable that is set to one after

[1] Currently Solidity is considered to be the main language for Smart contract development.

```
contract Bob{
   uint sent = 0;
  function ping ( address c) {
    if (sent < 1){
      c.call.value(2)();
      sent = 1;
    }
  }
}
```

(a) Smart contract with reentrancy bug

```
contract Mallory{
  function(){
    Bob(msg.sender).ping(this);
  }
}
```

(b) Smart contract exploiting reentrancy bug

Figure 1: Reentrancy Attack

```
0  :  PUSH 4294967295;
1  :  SLOAD;      \\ get value of sent
2  :  PUSH 1;
3  :  LE;         \\ check 1 <= sent (= !(sent < 1))
4  :  JUMPI 13;
5  :  PUSH 64;    \\ - oa
6  :  PUSH 0      \\- id
7  :  PUSH 2;     \\- va
8  :  INPUT;      \\- to
9  :  CALL;
10 :  PUSH 1;
11 :  PUSH 4294967295;
12 :  SSTORE;     \\ sent = 1
13 :  STOP;
```

Figure 2: `bob.txt`: (Simplified) bytecode for the `Bob` contract in Figure 1a.

the successful money transfer. Instead, it turns out that invoking the `call.value` function on a contract's address invokes the contract's fallback function as well.

Given a second contract `Mallory`, it is possible to transfer more money than the intended 2 (sub)coins to the account of `Mallory`. By invoking `Bob`'s function `ping` with the address of `Mallory`'s account, 2 (sub)coins are transferred to `Mallory`'s account and additionally the fallback function of `Mallory` is invoked. As the fallback function again calls the `ping` function with `Mallory`'s address another 2 (sub) coins are transferred before the variable `sent` of contract `Bob` was set. This looping goes on until all gas of the initial call is consumed or the callstack limit is reached. In this case, only the last transfer of (sub)coins is reverted and the effects of all former calls stay in place. Consequently the intended restriction on contract `Bob`'s `ping` function (namely to only transfer 2 (sub) coins once) is circumvented.

We present the reentrancy bug using `Solidity` code of the `Bob` contract. However our analysis will be defined in terms of *tinyEVM* bytecode, therefore in Figure 2 we show the (simplified) version of the bytecode for the `Bob` contract.

The bytecode execution starts by pushing the value `4294967295` to the stack in Line 0. We assume that this is the position in the contract storage where the value of `sent` is stored (see also Figure 1a

Line 2). This value is then pushed to the stack at Line 1 by the `SLOAD` instruction. Next the value `1` is pushed to the stack and compared to the value of `sent` at Line 3. If `sent` is bigger or equal than `1` (which corresponds to the original condition `sent < 1` in Figure 1a Line 4 being false), the LE instruction will push `1` to the stack and `0` otherwise. After the comparison we perform a conditional jump: if the value on top of the stack is `0` we go to Line 5, otherwise jump to Line 13 (where we stop). There are three subsequent pushes at Line 5 - 7: The first specifies the output address value for the call (`64`), the second provides the input data (`0`), and the last one pushes the value to be transferred (`2`). The `INPUT` instruction at Line 8 pushes the address of the destination account which is read from the call input (corresponding to the fact that the recipient is given as argument `c` in `bob.txt`). Lines 5-8 make all necessary data available for the `CALL` at Line 9. After performing the `CALL` at Line 10 we push `1` to the stack and at Line 12 store `1` at the position `4294967295` (pushed at Line 11), i.e., we set the value of `sent` to `1`.

For checking single entrancy, we will check whether it is ever possible to reach a CALL in a reentering execution. If no such CALL instruction can ever be reached, the contract is guaranteed to be safe. To check this automatically, we will follow the steps below:

1. We encode an over-aproximation of the *tinyEVM* bytecode semantics in *HoRSt* (`abstract-semantics.txt`) following the abstract semantics presented in the lecture.

2. We encode single entrancy as an (un)reachability query (`queries.txt`). Intuitively, this query will require that it is not possible to reach a CALL instruction when reentering.

3. We invoke *minnieThor* on the abstract semantics (`abstract-semantics.txt`), the reachability query (`queries.txt`) and the contract `bob.txt` as follows:

   ```
   minniethor bob.txt -s abtract-semantics.txt queries.txt
   ```

   minnieThor will parse the contract and generate using `abstract-semantics.txt` a Horn clause encoding of the abstract semantics of `bob.txt`. On this Horn clause representation it will invoke `z3`'s fixed point engine to determine whether the abstract configurations encoded by `queries.txt` are reachable or not. For the case of single-entrancy `queries.txt` will translated to individual queries for all CALL instructions of the contract. Since `bob.txt` has only a single CALL instruction, only one query will be produced. *minnieThor* will report the result of `z3` on this query:

   ```
   query id:        reentrancyCall_9
   execution time: 15
   result:          SATISFIABLE
   ```

   The first line identifies the query, using the query name (here `reentrancyCall`) and the program counter (here `9`). The second line gives the execution time needed to solve the query in milliseconds. Finally, the result indicates whether the abstract state encoded by the query is reachable (so the reachability query is `SATISFIABLE`) or unreachable (so the reachability query is `UNSATISFIABLE`). Consequently for the `bob.txt`, its single query being `UNSATISFIABLE` would indicate that it is safe: in this case it is impossible to reach its CALL instruction when reentering. A `SATISFIABLE` result indicates that it cannot be excluded that the CALL instruction is reached when reentering. Consequently, the contract could potentially be vulnerable (violate single entrancy). When invoking *minnieThor* on a contract with more than one CALL instruction, such a contract can only be considered safe if all queries that are produced for this contract (the queries for all the CALL instructions) give an `UNSATISFIABLE` result. This is as it needs to be excluded that any CALL instruction is reachable after reentering a contract.

Exercise 3.1: Abstract Semantics (15 points)

Implement a sound abstract semantics for tinyEVM Bytecode. To do so, extend the *HoRSt* specification in `abstract-semantics.txt` to implement abstract rules for all *tinyEVM* bytecode instructions (*task*). You can find a full account of the *tinyEVM* small step semantics in `smallstep.pdf`. To get started, have a look at our tutorial, where we show you how to proceed for the first two instructions. To check the correctness of your implementation, run the tests that we provide.

We prepared two test contracts, one for local instructions and one for the CALL instruction. You can run the local tests as follows:

```
minniethor local-contract.txt -s abstract-semantics.txt test-infrastructure.txt
                                 local-tests.txt
```

The CALL tests can be invoked by

```
minniethor call-contract.txt -s abstract-semantics.txt test-infrastructure.txt
                                call-tests.txt
```

Please have a look at the tutorial for an overview of how *HoRSt* tests are written and for getting familiar with our testing infrastructure. This will help you debugging.

As a final test, check that your analyzer identifies `bob.txt` to be potentially reentrant by running

```
minniethor bob.txt -s abstract-semantics.txt queries.txt
```

Exercise 3.2: Fixing reentrancy (5 points)

Fix the `bob.txt` contract not to exhibit a reentrancy vulnerability anymore. To this end create a *tinyEVM* contract `alice.txt` that implements the same functionality as `bob.txt`, but that is single-entrant (*task*). Explain why this contract is safe (*question*). Verify the result by checking `alice.txt` with your analyzer from the previous exercise.

Exercise 3.3: Soundness of the CALL rules (20 points)

Below we propose several modifications of the CALL abstraction presented in the lecture. Changes with respect to the original rules are highlighted in red.

- Which of them are sound, which are not? (*question*)
- For each sound set of rules, give an intuition why they are sound. To this end, refer to the three different ways of storage propagation presented in the lecture and explain how they are covered by the corresponding sound rule set (*question*).
- For each unsound set of rules, give a counter example smart contract that contradicts the soundness. To this end, prepare for each unsound rule set a file `counter-example`x`.txt` (where x is the number of the rule set) containing a *tinyEVM* bytecode program that is reentrant, but will be erroneously proven safe when replacing the CALL rules with the rule set x (*task*).

  For each of these smart contracts, describe a concrete reentrancy attack that would be possible (*question*).

Hint As a sanity check, implement all the presented CALL rules and run the resulting analysis on the example contract. Make sure that all sound analyses label your example contracts vulnerable, and each unsound analysis labels the corresponding counter example contract safe. Please also commit the implementations of your CALL rule implementations in separate files named `callx.txt` (where x is the number of the rule set) that contain only the respective CALL rule. Note that since you can provide several *HoRSt* files to *minnieThor*, the most convenient way of testing is to create a version of the abstract semantics (as implemented `abstract-semantics.txt`) without the CALL rule and to provide the different CALL rule variants as separate files.

1.

$$\mathsf{ExState}_{pc}((size, sa), ma, in, stor, cl) \\ \wedge\ size > 3 \\ \Rightarrow \mathsf{ExState}_{0}((0, \lambda x.0), \lambda x.0, \top, stor, 1)$$

$$\mathsf{ExState}_{pc}((size, sa), ma, in, stor, cl) \\ \wedge\ size > 3 \\ \wedge\ \mathsf{Halt}(stor_{inv}, r, 1) \\ \Rightarrow \mathsf{ExState}_{0}((0, \lambda x.0), \lambda x.0, \top, stor_{inv}, 1)$$

$$\mathsf{ExState}_{pc}((size, sa), ma, in, stor, cl) \\ \wedge\ size > 3 \\ \wedge\ sa[size-4] = \top \\ \Rightarrow \mathsf{ExState}_{pc+1}((size-3, sa_{\top}^{(size-4)}), \lambda x.\top, in, \lambda x.\top, cl)$$

$$\mathsf{ExState}_{pc}((size, sa), ma, in, stor, cl) \\ \wedge\ size > 3 \\ \wedge\ p = sa[size-4]\ \wedge\ p \in \mathbb{N} \\ \wedge\ \mathsf{Halt}(stor_{inv}, r, 1) \\ \Rightarrow \mathsf{ExState}_{pc+1}((size-3, sa_{\top}^{(size-4)}), ma_{r}^{p}, in, \lambda x.\top, cl)$$

2.

$$\mathsf{ExState}_{pc}((size, sa), ma, in, stor, cl) \\ \wedge\ size > 3 \\ \Rightarrow \mathsf{ExState}_{0}((0, \lambda x.0), \lambda x.0, \top, stor, 1)$$

$$\mathsf{ExState}_{pc}((size, sa), ma, in, stor, cl) \\ \wedge\ size > 3 \\ \wedge\ \mathsf{Halt}(stor_{inv}, r, 1) \\ \Rightarrow \mathsf{ExState}_{0}((0, \lambda x.0), \lambda x.0, \top, stor_{inv}, 1)$$

$$\mathsf{ExState}_{pc}((size, sa), ma, in, stor, cl) \\ \wedge\ size > 3 \\ \Rightarrow \mathsf{ExState}_{pc+1}((size-3, sa_{\top}^{(size-4)}), \lambda x.\top, in, stor, cl)$$

$$\mathsf{ExState}_{pc}((size, sa), ma, in, stor, cl) \\ \wedge\ size > 3 \\ \wedge\ \mathsf{Halt}(stor_{inv}, r, 1) \\ \Rightarrow \mathsf{ExState}_{pc+1}((size-3, sa_{\top}^{(size-4)}), \lambda x.\top, in, stor_{inv}, cl)$$

3.

$$\mathsf{ExState}_{pc}((size, sa), ma, in, stor, cl) \wedge size > 3 \Rightarrow \mathsf{ExState}_{0}((0, \lambda x.0), \lambda x.0, \top, stor, 1)$$

$$\mathsf{ExState}_{pc}((size, sa), ma, in, stor, cl) \wedge size > 3 \Rightarrow \mathsf{ExState}_{pc+1}((size - 3, sa_{\top}^{(size-4)}), \lambda x.\top, in, \lambda x.T, cl)$$

4.

$$\mathsf{ExState}_{pc}((size, sa), ma, in, stor, cl) \wedge size > 3 \wedge v = sa[size - 3] \Rightarrow \mathsf{ExState}_{0}((0, \lambda x.0), \lambda x.0, v, stor, 1)$$

$$\mathsf{ExState}_{pc}((size, sa), ma, in, stor, cl) \wedge size > 3 \wedge \mathsf{Halt}(stor_{inv}, r, 1) \Rightarrow \mathsf{ExState}_{0}((0, \lambda x.0), \lambda x.0, \top, stor_{inv}, 1)$$

$$\mathsf{ExState}_{pc}((size, sa), ma, in, stor, cl) \wedge size > 3 \Rightarrow \mathsf{ExState}_{pc+1}((size - 3, sa_{\top}^{(size-4)}), \lambda x.\top, in, \lambda x.\top, cl)$$

5.

$$\mathsf{ExState}_{pc}((size, sa), ma, in, stor, cl) \wedge size > 3 \Rightarrow \mathsf{ExState}_{0}((0, \lambda x.0), \lambda x.0, \top, stor, 1)$$

$$\mathsf{ExState}_{pc}((size, sa), ma, in, stor, cl) \wedge size > 3 \wedge \mathsf{Halt}(stor_{inv}, r, 1) \Rightarrow \mathsf{ExState}_{0}((0, \lambda x.0), \lambda x.0, \top, stor_{inv}, 1)$$

$$\mathsf{ExState}_{pc}((size, sa), ma, in, stor, cl) \wedge size > 3 \Rightarrow \mathsf{ExState}_{pc+1}((size - 3, sa_{1}^{(size-4)}, \lambda x.\top, in, \lambda x.\top, cl)$$

$$\mathsf{ExState}_{pc}((size, sa), ma, in, stor, cl) \wedge size > 3 \Rightarrow \mathsf{ExState}_{pc+1}((size - 3, sa_{0}^{(size-4)}, ma, in, \lambda x.\top, cl)$$

Submission Instructions

Please submit your solution by *24.07.2020* on TUWEL.

Your submission must be a single archive `your-group-number.zip` containing the following files:

Part	Required files
General	`report.pdf`
3.1	`abstract-semantics.txt`
3.2	`alice.txt`
3.3	`counter-example`x`.txt`, `call`i`.txt`

Where $i \in \{1, 2, 3, 4, 5\}$ and $x \in S$ where $S \subseteq \{1, 2, 3, 4, 5\}$ denotes the rule sets that you identified to be unsound. The `.pdf` file which contains all your answers and explanations should be generated using the `.tex` template provided on TUWEL.