Design choices and testing

Niccolò Piazzesi
n.piazzesi@studenti.unipi.it
Language Based Technology for Security
Homework 1 report

April 28, 2022

1 Introduction

In this report i will describe the design choices of the developed language and how i have tested the implementation. In the final section i will also describe some possible extensions.

1.1 Running the tests and examples

To build and test the code, you must have **Dune** and **QCheck** installed on your system. This can be done with the following command:

```
$ opam install dune qcheck
```

The following commands can be executed from the root of the project:

- Running tests

  ```
  $ dune test -f
  # -f makes the tests run even if
    the target was already built
  ```

- Running the simple examples

  ```
  $ dune exec hw1
  ```

2 Choices

2.1 Language design

The language was designed to be as simple as possible but still expressive enough to test the relevant aspects. Since the requested language had to be functional, i based it on the λ-calculus as it is the formal model of computation best suited to represent these kind of languages.

Named functions. A major change that i made was the introduction of named recursive functions. I think that it is valuable to introduce them as recursion gives a lot of power to a language and makes it more challenging (and fun) to reason about its properties.

Environment and scoping. While i tried to keep the implementation as abstract as possible so to focus on relevant properties, i also added some concrete features such as symbol tables and scoping. This was useful especially to distinguish between variable names and function parameters and made it easier to track name access inside sandboxes.

Execute syntax. Execute takes a list of permissions to allow various operations inside of it. While this may seem counter-intuitive, i believe that it still is a secure design. Any problem relative to data leakage can be traced to an explicit permission that can be easily disabled. By default, no external name is accessible, so it is always explicitly defined by the host system what data can be accessed.

Permission model. The access permission models in an abstract way more concrete permissions. In a concrete language you would have, for example, filesystem read and write permissions and network communication permissions such as **send** to allow the communication of private values. In our simple example, these can all be modelled as permissions to access certain function which could be called "read_file", "write_file" or "send".

Nested executions. While allowing nested executions could case other issues, such as non-termination (think of a cyclic execute where some code calls some other remote code that calls the original code and so on...), I believe that the presented semantics allows them in a controlled and safer way.

Using the permission set by the host system and ignoring the internal execute permissions should guarantee that no new vulnerabilities are introduced by other actors than the host system itself.

One could argue that the best possible choice is to disable nested executes completely, but i think that it is interesting to think of safe ways to introduce such constructs.

3 Testing

The implementation has been checked by using property based testing and the QCheck library. More specifically, i verified the security relevant aspect of the interpreter, which meant verifying two relevant properties:

1. **Every insecure expression raises a security violation error.**
2. **Secure expressions never raise a security violation error.**

3.1 Generating test cases

The more challenging part of testing was handling the generation of random expression as test cases. Using uncontrolled randomness would result in many useless test, as the majority of possible generated expressions would make no computational sense (e.g applying an integer as a function or adding two lambdas). To generate useful tests i have used a type driven approach, which is popular in testing language implementations [1]. Every valid expression is assigned one of three possible types:

1. Int
2. Bool
3. Fun(t1,t2)

where Fun(t1,t2) represents a function from type t1 to t2. Handling variable typing was done using a typing context Γ which is a list of mapping **(x,t)**, where **t** represent the type of variable **x**. The expression generator takes a type T as input and generates a valid expression of type T. As an example, if we want to generate an integer expression, the generator chooses at random between a integer literal, a binary operation between two other integer expressions and resulting in an integer or a composite expression (let,letfun, if, or function call) that recursively generate valid sub expressions and evaluates to an integer. For simplicity the basic generator does not generate execute expressions, which are handled differently to generate insecure expressions.

3.2 Generating insecure expressions

Generating insecure expressions involves two things:

1. Generating a execute where we perform a security relevant action **A**.
2. Making sure that action **A** is disabled inside the sandbox.

The execute generator takes a random relevant action A to check and produces one of the following thing:

- If A is a **access(x)** it simply creates a variable access **x**.
- If A is **execute** it produces a new execute with a random nested expression.
- If A is **binary_ops** it creates a random binary expression.

To ensure that the expression in insecure we create a random list of permissions and remove the generated action A from it.

The generator does not create a composite expression. While this restricts the cases actually tested, i think that generating more complex insecure expressions adds a lot of complexity to the general expression generator. This added complexity does not result in many more relevant tests, since we always end up in one of the generated cases as sub expressions.

4 Possible extensions

The language presented is a very simple and minimal one. One could think of various linguistic constructs to enrich it and make it more powerful. Other than more syntactic extensions, i think of two possible extensions, that could make it more safe and secure:

- **Code signature**. Mobile code could be signed with a cryptographic key, and an interpreter could reject code that it does not trust entirely.
- **Type system**. A very basic type system could be added to the language itself, which would prevent many wrong expressions to be evaluated.

A more advanced solution,based on type systems (and formal logic in general), but out of scope for our simple example, could be **proof carrying code**[2].

The basic idea is the following: the host system can determine if it safe to execute untrusted code. To do this, the untrusted code must also send a *proof* that states that the code satisfies a certain security policy specified by the host system itself. The system can then check that the proof is valid before executing the code.

References

[1] Pedro Vasconcelos. Verifying a simple compiler using property-based random testing.

[2] George C Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 106–119, 1997.