LANGUAGED-BASED TECHNOLOGY FOR SECURITY
Homework Assignment 1

Due: April 23 2023

Introduction

In this homework, you will be asked to design a simple functional language with primitive abstractions to protect the execution of programs from untrusted code, i.e. code that has not been evaluated or examined for correctness and adherence to the security policy. It may include incorrect or malicious code that attempts to circumvent the security requirements.

Untrusted code adds new challenges for making software secure. Note that web applications have made untrusted code a part of everyday life: visiting a web page typically loads JavaScript code from multiple providers into browsers. Cloud services allow third parties to provide applications that are dynamically combined with the core cloud functionalities. Finally, even traditional desktop applications dynamically download plugins from external providers.

The adoption of untrusted code introduces a scenario where private and sensitive information is being pushed outside the perimeter of a trusted execution environment. Indeed, the benefits of freely sharing code come at a cost: dynamically combining code from multiple sources might yield an insecure application.

The assignment

In this homework you will be asked to design and build in OCAML the trusted execution environment of a functional language with linguistic primitives for managing the secure execution of untrusted code.

The programming language and its trusted execution environment support sharing of untrusted code while enforcing the security requirements. Specifically, the language provides a programming model that offers language-level integration of the (hardware-based) enclave mechanism to protect data and code from untrusted accesses. Intuitively, programmers can execute computations securely just by adopting the **enclave** primitive operation to group the code to be placed inside the enclave and the **gateway** primitive to declare the functions accessible from the non-enclave environment. Also, programmers can label secret data with the **secret** attribute to control the release of secret information to the non-enclave environment.

We illustrate the main feature of the language by the simple example below.

We first declare the enclave named myEnclave.

```
enclave myEnclave =
let secret string password = "abcd";
let checkPassword (guess:string): bool =
       password = guess;
gateway checkPassword: string -> bool
end
```

Both functions and data of the enclave are securely isolated inside the enclave. Within an enclave, the secret annotation is used to identify secret data to ensure that their values should not be leaked to the non-enclave environment. The gateway annotation act as the interface between the enclave and the non-enclave environments.

In our example, the *checkPassword* function is annotated with the gateway annotation. The *checkPassword* function accepts a string from the non-enclave environment and compares it with the (secret) password field, the result of the comparison is returned to the non-enclave environment as a boolean value. The return value of the gateway function must not leak the secret information.

In addition to enclave we introduce the operators **include** and **execute** to include and execute an untrusted code inside the program.

The code

```
include untruted extCode =
let string upass = "abcd" in
      if checkPassword(upass) then "abcd"
end
```

allows to include the unstruted code, named extCode, inside the program. The code

```
execute(extCode)
```

allows to execute the code named extCode.

The execution environment manages both computation inside/outside enclave and the interplay between trusted/untrusted functions. Note that the attacker controls the non-enclave environment by controlling the non-enclave code and data.

In our running example, since the attacker controls the data outside of the enclave. Hence, the attacker can manipulate the parameter passed to *checkPassword*. Also, the attacker can call functions, e.g., checkPassword, in any order, and thus can control the release of values.

The run time enforcement mechanisms must guarantee security against these types of active attackers and ensure that enclave functions do not leak secret data.

For instance, in our example the program

```
execute(extCode);
```

when evalutated yields a security exception since the secret data is leaked.

The assignment consists of two mandatory contributions:

1. Discuss the language design to support enclave programming.
2. Describe the implementation of its execution environment (interpreter and run-time support) and evaluate its applicability by presenting some simple case studies.

Note

You can design the programming language and its trusted execution environment in total freedom. For example, you can freely equip the language with further suitable linguistic abstractions. but you have to justify all the choices made.

Learning objectives

The goals of the homework are to appreciate and understand the trade-offs in the design and implementation of an experimental secure-aware programming language. Considering the issues of enclave programming in the setting provided by the assignment has the advantage of addressing a more general, abstract instantiation of the real problems.

Submission Format

The submitted homework assignment must include:

1. The definition of the programming language along with some examples.
2. The source code of the execution environment of the language.
3. A short description of the design choices.

The parts (1-3) above are *mandatory*.

The homework also includes an *optional* part briefly discussed below. Property-based testing (PBT) -- or automatic specification-based testing -- is a form of black-box testing where a software artifact is subjected to a large number of checks derived from formal statements of its intended behavioral properties. For example, if the program under test is a function **f** that takes a list of numbers as its input and (hopefully) returns the same list in sorted order, then a reasonable specification of **f** might be **sorted(f(oldlist)) permutation(f(oldlist),oldlist),** where sorted is a simple function that checks whether its input is ordered and permutation checks whether one of its argument lists is a permutation of the other. To test whether **f** satisfies this specification, we generate a large number of random input lists, apply **f** to each one, and check that sorted and permutation both yield true.

PBT was popularized in the functional programming world by the QuickCheck library for Haskell. QuickCheck has been ported to many other platforms (e.g. see https://github.com/c-cube/qcheck for OCAML).

The optional part of the homework consists in developing the property based testing of the proposed Trusted Execution Environment.

Submissions Guidelines

The assignment must be submitted no later than midnight of the day it is due.

More instructions on the submission next week.