Project Canyon

Lumberjacks Incorperated (2018)

Contents

1.Introduction

1.1 Project Abstract

Canyon is an implementation of the language model described in Project Ravine. It completes the objective of Project Ravine, which is to create a programming language that facilitates the 'permission driven programming' paradigm by embedding the 'clearance conformance' property at the language level, such that all programs expressed in Canyon will automatically inherit this property.

This part of the project is the concrete implementation of that language.

1.2 A Seperate High and Low Level Language

In order to both conform the model specification in Project Ravine, as well as provide a language of good utility to a programmer, Canyon has been separated the implementation into a high level language, and a low level language.

The high level language is designed to provide a useful interface to a programmer to give both ease of use as well as power, while the lower level language is designed specifically to conform to the Project Ravine specification.

The reason for this is that the direct implementation of the Project Ravine specification is not the most ideal language to program in, so we provide a higher level language with features closer to that of which programmers will find most practical.

1.3 Executing The Language

The high level language of Canyon is compiled into an equivalent program in the low level language of Canyon. The low level program is then executed on the fly as instructions that are interpreted within an interpreter written in the functional programming language, Haskell.

Thus, the base language that this language is both modelled and executed under is Haskell, although the high level language provides a much more imperative programming style than Haskell's functional style.

2.The Low Level Language

2.1 Expressions

Expressions are defined in the low level language identically to the Project Ravine Specification, with the addition of actual values through modal objects.

```
data Expr =
NULL | NOP | SEQ Expr Expr | INIT Model_obj_init TYPE Varname | VALUE Model_obj_value TYPE
| VAR Varname | ASSIGN Varname Expr | OP Model_obj_op Expr Expr | IF Expr Expr Expr | WHILE Expr Expr
| RAISE Security Expr | EXPR_ERROR String
```

The reason for an implementation identical aside from the inclusion of actual values through modal objects to to simplify the reasoning that our language is indeed an implementation of the Project Ravine specification.

We have implemented an inclusion in the program state of mappings between modal objects and their security type value. Since the proof did not rely on the implementation of actual values, only an association between all values and a security type, our implementation of values does not affect our conformance with the Project Ravine specification.

2.2 Execution of Expressions

The language semantics described in Project Ravine are big step specifications of the program evaluation. For the concrete implementation, we have written a 'single step evaluator' function that executes a single instruction at a time.

For each language construct (node in the AST), 'eval_step' first attempts to reduce each sub expression until we retrieve the results required in the preconditions of the inductive definitions. Finally, once and if we have successfully reduced the sub expressions after successive applications of eval_step, and the required preconditions are met, the language construct (node) is reduced into a final result and becomes a leaf in the AST.

In this way the AST is reduced bottom up and from left to right as eval_step is successively applied. eval_step then conforms to the inductive definition as successive applications of eval_step cause a given program to approach the result of a full reduction as defined by the big step semantics.

2.3 Low Level Language Conformance

The low level language conformance can be seen through reasoning introduced in the above two section, with parallels drawn between both expressions and their evaluation.

This means that programs written in this language and executed under this interpreter (implemented in Haskell) all inherit the 'clearance conformance' property, a fact proved in Project Ravine.

However, this language is not optimal as a practical programming language, so a higher level language for the programmer has been introduced to add further utility to the guarantee of the security properties that exist at this level.

3.The High Level Language

2.1 Expressions

Expressions are defined in the high level language to provide an interface closer to what imperative programmer languages are accustomed to.

```
data CanyonExpr =

    Dave | Jimmy | Block [CanyonExpr] | Init SecurityLevel CanyonType Variable |
    Assign SecurityLevel Variable CanyonExpr
    | Const SecurityLevel CanyonValue | Var Variable
    | If SecurityLevel CanyonExpr CanyonExpr CanyonExpr | While SecurityLevel CanyonExpr CanyonExpr
    | Un CanyonOp CanyonExpr | Bin CanyonOp CanyonExpr CanyonExpr
    | GetChar SecurityLevel Variable
```

2.2 Compilation of Expressions

The domain of reasoning we are interested in for the compilation of expression from the high to the low level language is that the security levels of data and operation constructs are preserved so that the translation cannot cause a low level language programs that subverts the security of the high level program.

To reason that the security within the program is preserved with respect to operation, data, and their assigned security levels, we discuss several parts.

First, there is a one to one mapping and hence correspondence between variables in the high level language and variables in the low level language.

Second, for all low level constructs, the is an analogous high level construct that both compiles into the low level language equivalent and preserves or raises the security of the construction.

Third, other constructions without an analogue in the low level language do not affect data directly or indirectly, and hence do not affect security within the program.

Hence, security of given variables, and operations, is preserved in the compilation process, such that the low level language program produced by compilation cannot subvert the intended security in this way of the high level language program.

4.Dealing with External Data and Operations

4.1 Introduction

In order to deal with both intaking data from an external source, as well and sending data to an external destination, we have adopted a boundary concept for any data passing through or leaving the boundary.

4.2 The Boundary Concept

For data entering the boundary, these values are assigned a security level at the crossing point. We are not attempting to assign an inherent security value to all data that exists. Instead, we are giving the programmer the ability to take control of the allowable flows of the data that enters the program boundary with respect to security levels within the program.

For data leaving the boundary, we have an programmer assignable security level attached to operations that distribute data externally. This is so that the programmer can control what security level of data can be sent externally at each of the external exit points with in a program.

4.3 Implementation of External Operations

External operations have been defined by combining core operations, and by directly accessing or modifying modal data stored within the core state between applications of eval_step.

For example, the GetChar operation is executed by defining in the core language; setting a flag, then assigning a variable with data from a previously defined getChar object, then resetting the flag. The char from getChar is injected directly into the state at a higher level after the setting of the flag, and before the assignment of the data.

This way we have wrapped external operations within the existing constructs of the low level language such that they conform the the Project Ravine specification (and hence maintain 'clearance conformance') and allow the programmer to take control from a security perspective of the external data entry and exit points within a program, in a manner adhering the the boundary concept described above.