Project Ravine

Lumberjacks Incorperated (2018)

Contents

1.Introduction

1.1 Project Abstract

The goal of Ravine consists of two stages. The first is to create a new programming language which facilitates 'permission driven programming' at the language level. The second is to model this new language within the Isabelle Proof Assistant to show that the security properties required to facilitate 'permission driven programming' hold for all possible programs expressed within this new language.

This is an attempt to create a programming language that facilitates 'permission driven programming' at the language level, as well as provide guarantees with regard to specific security properties within all programs expressed in this new language.

We are pushing the burden of proof to the language level, so that it only has to be done once but the results will still extend to all programs written in this new language. This reduces the effort and complexity required for future programming in this domain by eliminating the need for programmer proofs in regards to the given security properties.

1.2 Permission Driven Programming and Clearance Conformance

'Permission driven programming' is where security levels exist for operations and data within a program, and it is important that information does not flow from a higher security level source to a lower security level destination.

We say that programs where information cannot flow from a higher security source to a lower security destination have 'clearance conformance'. Thus, when looking from a 'permission driven programming' perspective, we are interested in whether or not programs have 'clearance conformance'.

This can include programs which deal with sets of data that contain information from a number of different security or permission levels, such as file systems drivers or databases, where it is crucial that information from certain security levels cannot be obtained by other lower security level operations or data.

1.3 The New Language: Canyon

The new language, Canyon, is designed to be a fairly standard traditional imperative programming language, with one crucial addition.

This addition is the ability to create and maintain arbitrary and allocated security levels to both operations and data, in order to facilitate 'permission driven programming' at the language level.

1.4 'Clearance Conformance' at the Language Level

One key advantage to implementing this ability at the language level is that any program written in the new language (Canyon) will always have 'clearance conformance' by inheritance, rather than having to embed this property into the program design at the application level.

2.Embedding Clearance Conformance

2.1 A New Concept: Security Types

To embed ‘clearance conformance’ at the language level, we introduce the idea of a new ‘type’ to our language domain. Each of the data and operation constructs (operations, values, and variables) will be attached to a ‘security type’, which represents a relationship with a specific security level.

This security type is used by the language to control the flow of information in such a way as to prevent information flowing from a higher security source to a lower security destination. This new type system is introduced as a layer on top of the traditional type system used by most conventional type safe programming languages.

2.2 Creating a Tree-Structured Embedded Security Level Propagation

An expression in Canyon is recursively defined over the traditional imperative constructs such as ‘sequence’, ‘if’, ‘while’, etc. The crucial new addition is the attachment of a security level assigned to each of these constructs.

A valid program or expression in Canyon also constrains the possible values these security levels may take within the valid expression.

The general idea is that within any expression, all subexpressions contain security levels that are greater than or equal to the security level of the containing expression.

Example:

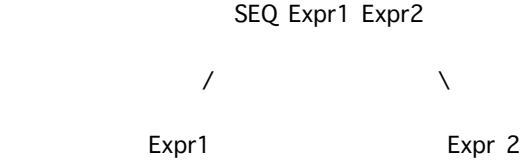
If

Sequence (Expression_1 (Security Level x)) (Expression 2 (Security Level y)) (Security Level m)

is a valid program, then security levels x and y are both greater than or equal to the security level m.

This results in a tree-like structure for all valid program expressions, where the security level at each tree level is greater than the security level of its parent level.

Example:



where the security levels of Expr1 and Expr2 are greater than or equal the security level of the SEQ expression.

2.3 Constraining Assignment And Initialisation

Assignment is constrained such that you cannot assign data with a certain security level to a variable with a lower security level. As well as this, you cannot initialise a variable with a lower security level than the security level of the containing expression.

This completes the creation of a tree-like nesting of increasing security levels with Canyon expressions and subexpressions, with further ‘leaf’ constraints to contain both assignment and initialisation.

2.4 The Combined Effect

This tree list nesting of increasing security levels within Canyon expressions and subexpressions combined with the assignment and initialisation constraints produce a language where in any possible expression information cannot flow from a higher security level source to a lower security level destination.

The assignment constraint means that this flow cannot occur directly, while the subexpression constraint ensures this flow does not happen indirectly, such as using a conditional expression based on higher security level data to influence assignment of lower level variables.

3.The Proof Process Structure

There are 3 distinct parts to this proof:

3.1 The formal programming language model in Isabelle

This part consists of modelling Canyon within the Isabelle Proof assistant. This model is kept at a sufficiently high level with all non-relevant details towards the key security property proofs removed, in an effort to keep this proof as simple and straight-forward as possible.

For instance, we have excluded many low level details of value and operation implementations not relevant to the proof.

3.2 The Language Model Lemmas

In order to later complete the proofs necessary to show that the Canyon model does satisfy the security properties listed, we have included 3 seperate groups of lemmas:

- Meta Lemmas:

  These are lemmas such as ‘mapping soundness’, that relate to statements made about the language itself.

- Inversion Lemmas:

  We have defined an inductive predicate ‘eval’, which encapsulates the big step semantics of our language. For each of the inductive case rules, we have written inversion lemmas, as auxiliary lemmas for the security statements later

- Completeness/Security Increasing Lemmas:

  These are lemmas relating to the completeness properties of the inductive cases of the ‘eval’ predicate. They also show the ‘security increasing’ composite nature of each case of ‘eval’, in that each subexpression contains an increasing/equivalent security level.

3.3 The Security Statements and Proofs

The final part of the proof process involves formulating lemmas that collectively define the property of ‘clearance conformance' applied to the Canyon model at the language level.

For the sake of both simplicity and clarity, the property has been broken down into multiple lemmas that collectively represent ’clearance conformance’, for ease of proof, and readability.

4. The Language Model

Canyon is similar to most traditional imperative programming languages, except that a security level type has been added, with security levels assignable to operations as well as data.

The language model includes type definitions, assistant functions, and an inductively defined predicate for the program expression definition as well as evaluation.

4.1 Auxiliary Function Definitions

List Length
- This is used to return the number of elements in a list as a 'nat'

```
fun list_length :: "'a list ⇒ nat" where
        "list_length Nil = 0" |
        "list_length (Cons n N) = Suc (list_length N)"
```

List Append
- This to add an element to the inner most position in a list

```
fun list_append :: "'a list ⇒ 'a ⇒ 'a list" where
       "list_append Nil a = Cons a Nil" |
       "list_append (Cons n N) a = Cons n (list_append N a)"
```

List Update
- Change the entry in a list related to a given position to a new entry

```
fun list_update :: "'a list ⇒ 'a ⇒ nat ⇒ 'a list" where
       "list_update Nil _ _ = Nil" |
       "list_update (Cons m M) v 0 = (Cons v M)" |
       "list_update (Cons m M) v (Suc n) = (Cons m (list_update M v n))"
```

Add Mapping
- Add a mapping to the given State value

```
fun add_mapping :: "state ⇒ mapping ⇒ state" where
       "add_mapping (State sl) m = State (m # sl)"
```

Empty State
- Used to represent a state which contains no mappings

```
definition empty_state :: "state" where
       "empty_state = State []"
```

Is Mapping
- Used to test whether a mapping exists within a State

```
inductive
        is_mapped :: "state ⇒ mapping ⇒ bool"
where
        is_mapped_0: "is_mapped (State (Cons (Map n t) M)) (Map n t)" |
        is_mapped_S: "n2 ≠ n  ⟹ is_mapped (State M) (Map n t)
                              ⟹ is_mapped (State (Cons (Map n2 t2) M)) (Map n t)"
```

*Instead of being defined as a function, this is defined as an inductive predicate for convenience

4.2 Type Definitions

Security Level

- The is the security level implementation, represented as a natural number

```
type_synonym security = nat
```

Variable Name

- The implementation of variable names in our language model are natural numbers

```
type_synonym var_name = nat
```

Security Type

- The is the security type implementation, represented as a wrapping around the security level implementation

```
datatype type = Type security
```

Variable to Security Level Mapping

- Values of this type represent a direct connection between a variable and its security level

```
datatype mapping = Map var_name type
```

Security State

- Values of this type are used to represent a given set of mappings between variables and their respective security levels as a state

```
datatype state = State "mapping list"
```

4.3 Meta Language and Auxiliary Lemmas

Is Mapped Inversion

- This lemma states that if is_mapped s m is true, then either the mapping exists on the outer most list position of the State, or it exists on an inner position in the State list.

lemma is_mapped_inversion:

is_mapped s m ⟹

((∃ n t M. (s = (State (Cons (Map n t) M)) ∧ m = (Map n t))) ∨ (∃ n2 n M t t2. (is_mapped (State M) (Map n t) ∧ n2 ≠ n ∧ (s = (State (Cons (Map n2 t2) M)) ∧ m = (Map n t)))))

Mapping Soundness

- This lemma states that after the add_mapping operation is performed with a specific state and additional mapping, that we can then check with is_mapped on the same state and mapping that the mapping does indeed exist within the state.

lemma mapping_soundness:

add_mapping s1 (Map n t) = s2 ⟹ is_mapped s2 (Map n t)

Mapping Preservation

- This lemma states the if add_mapping is used to add a mapping to a state, and this is done a further time on the resultant state, is_mapping returns true with the final resultant state, and the original mapping added.

lemma mapping_preservation:

n ≠ n2 ⟹ is_mapped s1 (Map n t) ⟹ add_mapping s1 (Map n2 t2) = s2

⟹ is_mapped s2 (Map n t)

Empty State Existence

- This lemma states the that no mapping exist in the empty state

lemma empty_state_existance:

¬(is_mapped empty_state (Map n t))

4.4 Language Expression Datatype Definition

This datatype is a recursive definition that encapsulates the different types of expressions in Canyon.

NULL
- This represents the end of a program when it is the only remaining expression

NOP
- This is a traditional 'no operation'

RAISE
- This is used to raise the security level of a given expression

SEQ
- This is used to combine two expressions in a sequence

INIT
- This is used to initialise a variable with a given security type

VALUE
- This is simply an expression containing a value with a security type

VAR
- This is a variable that is associated with a security type

ASSIGN
- This assigns the value of an expression to a variable

OP
- This represents operations of two values which result in a value

IF
- This represents the traditional conditional 'if'

WHILE
- This represents the traditional condition loop 'while'

```
datatype expr

    = NULL | NOP | SEQ expr expr | INIT type var_name |
      VALUE type | VAR var_name | ASSIGN var_name expr |
      OP expr expr | IF expr expr expr | WHILE expr expr
```

4.5 Big Step Semantics Inductive Definitions

For this language model, we use big step semantics for our programming model. This predicate is defined inductively over the expression datatype.

```
inductive
 eval :: "expr ⇒ state ⇒ security ⇒ expr ⇒ state ⇒ bool"
```

Constant
- This represents the evaluation for a value

```
Constant: "eval (VALUE t) s sec (VALUE t) s" |
```

NULL
- This represents the evaluation for a NULL

```
Null: "eval NULL s sec NULL s" |
```

Nop
- This represents the evaluation for a 'No Operation', which evaluates to NULL

```
Nop: "eval NOP s sec NULL s" |
```

Seq
- This represents the evaluation of a sequence of two expressions. A sequence of two expressions only evaluates if both expression both individually evaluates, and also that this must both evaluate at an equal or higher security level than the security level given to the original sequence instruction

```
Seq: "    eval e1 s1 sec1 NULL s2
⟹ eval e2 s2 sec2 NULL s3
⟹ sec1 ≥ sec3
⟹ sec2 ≥ sec3
⟹ eval (SEQ e1 e2) s1 sec3 NULL s3" |
```

Init
- This represents the evaluation of an initialisation. The variable to be initialised must not already by mapped, and the initialisation security level must be higher then the security level given to the initialisation expression

```
Init: "     ¬(∃t. is_mapped s' (Map n t))
⟹ s = (add_mapping s' (Map n (Type tsec)))
⟹ tsec ≥ sec
⟹ eval (INIT (Type tsec) n) s' sec NULL s" |
```

Assign
- This represents the evaluation of an assignment. The variable to assign to must already be mapped within the state. The security of the value to assign to the variable must be less than or equal to the security level of the variable. The security level of the variable must also be greater than or equal to the security level of the assignment expression.

```
Assign: "    is_mapped s' (Map n (Type vsec))
 ⟹ eval e s' sec (VALUE (Type tsec)) s
 ⟹ vsec ≥ tsec
 ⟹ vsec ≥ sec
 ⟹ eval (ASSIGN n e) s' sec NULL s" |
```

Var
- This represents the evaluation of a variable. A variable is evaluated to its value

Var: "is_mapped s (Map n t) ⟹ eval (VAR n) s sec (VALUE t) s" |

Op
- This represents the evaluation of an operation. The security of the the resulting value is the max of the two security levels given the security levels of each operand expression.

Op: " tsec3 = max tsec1 tsec2
  ⟹ eval e1 s1 sec (VALUE (Type tsec1)) s2
  ⟹ eval e2 s2 sec (VALUE (Type tsec2)) s3
  ⟹ eval (OP e1 e2) s1 sec (VALUE (Type tsec3)) s3" |

If Then
- This represents the evaluation of an If true case. The condition must evaluate to a value that has a security level greater to the security level than the expression in the either second position of the expression.

If_then: " eval econd s1 sec (VALUE (Type tsec)) s2
  ⟹ eval ethen s2 tsec NULL s3
  ⟹ eval eelse s2 tsec NULL s4
  ⟹ tsec ≥ sec
  ⟹ eval (IF econd ethen eelse) s1 sec NULL s3" |

If Else
- This represents the evaluation of an If false case. The condition must evaluate to a value that has a security level greater to the security level than the expression in the either second position of the expression.

If_else: " eval econd s1 sec (VALUE (Type tsec)) s2
  ⟹ eval ethen s2 tsec NULL s3
  ⟹ eval eelse s2 tsec NULL s4
  ⟹ tsec ≥ sec
  ⟹ eval (IF econd ethen eelse) s1 sec NULL s4" |

While
- This represents the evaluation of while loop expression, where the security level is preserved through the loop.

While: "eval (IF econd (SEQ eloop (WHILE econd eloop)) NULL) s' sec NULL s
  ⟹ eval (WHILE econd eloop) s' sec NULL s"

RAISE
- This represents the the raising of the contained expression as RAISE is evaluated.

Raise: "secr ≥ sec ⟹ eval e s' secr NULL s
⟹ eval (RAISE secr e) s' sec NULL s" |

4.5 Big Step Semantics Inversion Lemmas

We have written inversion lemmas for each case rule of the inductive 'eval' definition to assist in the security property proof process.

Seq Inversion
- If a seq expression evaluates, then each individual expression that composes the seq expression evaluates, and the security level of these individual expression is higher than the security level of the sequence binding expression.

lemma seq_inversion:

eval (SEQ e1' e2') s3' sec3 e3 s3 ⟹
(∃sec1 s4 sec2 . (eval e1' s3' sec1 NULL s4 ∧ eval e2' s4 sec2 NULL s3 ∧ sec1 ≥ sec3 ∧ sec2 ≥ sec3))

Init Inversion
- If a Init expression evaluates, then the resulting state is the first state with the mapping for the new variable added. Also, the security level of the variable added is greater than or equal to the security level of the expression.

lemma init_inversion:

eval (INIT t vn) s' sec e s ⟹
(∃ tsec. (e = NULL ∧ t = (Type tsec) ∧ tsec ≥ sec ∧ s = (add_mapping s' (Map vn (Type tsec))) ∧ ¬(∃ t. is_mapped s' (Map vn t))))

Assign Inversion
- If an Assign expression evaluates, then the expression assigned to the variable evaluates to a value with a security level less than or equal to the security of the variable being assigned to, and both of these security levels are greater or equal to the security value of the expression. It also requires that the variable have a valid mapping in the state.

lemma assign_inversion:

eval (ASSIGN n e) s' sec er s ⟹
(∃ vsec tsec . (er = NULL ∧ vsec ≥ sec ∧ vsec ≥ tsec ∧ eval e s' sec (VALUE (Type tsec)) s ∧ is_mapped s' (Map n (Type vsec))))

Var Inversion
- If an Var expression evaluates, then the resultant expression is a value, and the variable was mapped in the state.

lemma var_inversion:

eval (VAR n) s' sec e s ⟹ (∃ t. (e = (VALUE t) ∧ is_mapped s' (Map n t)))

Op Inversion
- If an Op expression evaluates, then both operand expression evaluate to a value, and the resultant security level is the max security level of these two values.

lemma op_inversion:

eval (OP e1 e2) s1 sec e s3 ⟹
(∃ tsec3 tsec1 tsec2 s2. (e = (VALUE (Type tsec3)) ∧ tsec3 = max tsec1 tsec2
∧ eval e1 s1 sec (VALUE (Type tsec1)) s2 ∧ eval e2 s2 sec (VALUE (Type tsec2)) s3))

If Inversion
- If an If expression evaluates, then the condition expression evaluates down to a value, and both other expression evaluate to NULL, and the security level of the condition is greater than or equal to the expression level of the IF expression, and the second expressions.

lemma if_inversion:

eval (IF econd ethen eelse) s1 sec ethen2_or_eelse2 s3 ⟹
(∃tsec s2 eelse2 ethen2 s5 s6. (eval econd s1 sec (VALUE (Type tsec)) s2
∧ tsec ≥ sec ∧ eval eelse s2 tsec NULL s5 ∧ eval ethen s2 tsec NULL s6 ∧ (s5 = s3 ∨ s6 = s3)))

While Inversion
- If a While expression evaluates, then the condition expression evaluates down to a value, and the security level of the condition is greater than or equal to the expression level of the While expression, and the second expression.

lemma while_inversion:

eval (WHILE econd eloop) s' sec e s ⟹
(e = NULL ∧ eval (IF econd (SEQ eloop (WHILE econd eloop)) NULL) s' sec NULL s)

4.6 Security Increasing and Completeness Lemmas

4.6.1 Purpose

These lemmas serve two purposes:

First, they show completeness for each inductive case rule, such that for expressions that evaluate in our language composite expression also evaluate.

Second, they show they for all embedded expression that make up the outer expression, the security level increases, which is important for the final security
statement proofs.

4.6.2 Definition

Seq Increasing Security And Completeness
- This lemma states that if a Seq expression evaluates, then each component expression evaluates, and each sub expression has increasing security.

lemma SecIncreasingComplete_Seq:

eval (SEQ e1' e2') s3' sec3 e3 s3
⟹ (∃ sec1 e1 e2 sec2 s2'. (eval e1' s3' sec1 e1 s2' ∧ eval e2' s2' sec2 e2 s3 ∧ sec1 ≥ sec3 ∧ sec2 ≥ sec3))

Assign Increasing Security And Completeness
- This lemma states that if a Assign expression evaluates, then the component expression evaluates, and the component expression has increasing security.

lemma SecIncreasingComplete_Assign:

eval (ASSIGN n ev') s' sec e s ⟹
(∃ s1' sec1 e1 s1. eval ev' s' sec1 e1 s1 ∧ sec1 ≥ sec)

Op Increasing Security And Completeness
- This lemma states that if a Op expression evaluates, then each component expression evaluates, and each sub expression has increasing security, and the resulting expression has a security greater than or equal to the max security of the component expressions.

lemma SecIncreasingComplete_Op:

eval (OP e1' e2') s3' sec3 e3 s3
⟹ (∃ sec1 e1 e2 sec2 s2' s4. (eval e1' s3' sec1 e1 s2' ∧ eval e2' s2' sec2 e2 s4 ∧ sec1 ≥ sec3 ∧ sec2 ≥ sec3))

RAISE Increasing Security And Completeness
- This lemma states that if a RAISE expression evaluates, then the security level of the second expression is greater than the security level of the first.

lemma SecIncreasingComplete_Raise:

eval (RAISE secr e) s' sec er s
⟹ (∃ sec2. (eval e s' sec2 NULL s ∧ er = NULL ∧ sec2 ≥ secr ∧ secr ≥ sec))

If Increasing Security And Completeness
- This lemma states that if a If expression evaluates, then each component expression evaluates, and each sub expression has increasing security, and the condition expression has a security greater than or equal to the max security of the component expressions.

lemma SecIncreasingComplete_If:

eval (IF e1' e2' e3') s3' sec3 e3 s3
⟹ (∃ sec1 e1 sec2 s2' s4 q1 q3 sec. (eval e1' s3' sec1 e1 s2' ∧ eval e2' s2' sec2 NULL s4 ∧ eval e3' q1 sec NULL q3 ∧ sec1 ≥ sec3 ∧ sec2 ≥ sec3 ∧ sec ≥ sec3))

While Increasing Security And Completeness
- This lemma states that if a While expression evaluates, then the component expression evaluates, the conditional expression evaluates, and the sub expression has increasing security, as well as the condition expression has a security greater than or equal to the max security of the component expression.

lemma SecIncreasingComplete_While:

eval (WHILE e1' e2') s3' sec3 e3 s3
⟹ (∃ sec1 e1 e2 sec2 s2' s4. (eval e1' s3' sec1 e1 s2' ∧ eval e2' s2' sec2 e2 s4 ∧ sec1 ≥ sec3 ∧ sec2 ≥ sec3))

4.7 Further Auxiliary Lemmas Lemmas Needed For Security Proof

Assignment Mapping
- This lemma states that if an assignment expression evaluates, then the variable was mapped in the initial state

lemma Assignment_Mapping:

eval (ASSIGN n e) s' sec e2 s ⟹ (∃vsec. is_mapped s' (Map n (Type vsec)))

Assignment Value Expression Completeness
- This lemma states that if an assignment expression evaluates, then the expression to be assigned evaluates to a value.

lemma Assignment_Value_Expression_Complete:

eval (ASSIGN n ev') s' sec e s ⟹ (∃ t s1. eval ev' s' sec (VALUE t) s1)

Init Validity
- This lemma states that if an Init expression evaluates, then the variable is mapped in the resultant state.

lemma Init_Validity:

eval (INIT t n) s' sec e2 s ⟹ is_mapped s (Map n t)

Var Validity
- This lemma states that if an Var expression evaluates, then the variable is mapped in the initial state.

lemma Var_Validity:

eval (VAR n) s' sec (VALUE t) s ⟹ is_mapped s' (Map n t)

Is Mapped Determinism
- This lemma states that for a all variables mapped in a state, they are always mapped to the same type (security level)

lemma is_mapped_determinism:

is_mapped s' (Map n t1) ⟹ is_mapped s' (Map n t2) ⟹ t1 = t2

Value Determinism
- This lemma states that for a given expression and state, it always evaluates to the same value

lemma value_determinism:

eval e s' sec1 (VALUE t1) s1 ⟹ eval e s' sec2 (VALUE t2) s2 ⟹ t1 = t2

5. The Security Statements

5.1 The Security Concept

The aim of the following security lemmas is to show the that following property holds for our language.

'Information can never flow from a higher security to a lower security level'

This means encapsulates two things, which are direct information flow, and indirect information flow.

Direct information flow across security levels means assigning information to a variable of a lower security level. Indirection information flow means using information indirectly to influence execution and gain knowledge about information of a higher security level in a lower security variable, such as using higher security expressions in a conditional expression that branches off into computations that use lower security variable assignments.

In order to state these properties apply to our language model, we have broken this concept down across several lemmas.

5.2 The Security Lemmas

Assignment Security Type
- This lemma states for an Assignment expression, where the sub expression evaluates to a value, the variable is mapped to a higher security level than the value security level being assigned to it. This means that direct information flow breaking the security flow of only low to high cannot happen.

lemma Assignment_Security_Type:

eval (ASSIGN n e) s' sec e3 s ⟹ eval e s' sec (VALUE (Type ve)) s
⟹ is_mapped s' (Map n (Type vsec)) ⟹ vsec ≥ ve

Assignment Security Context
- This lemma states for an Assignment expression, the variable to be assigned to has been mapped, and it is mapped to a security level higher than the security level of the expression.

lemma Assignment_Security_Context:

eval (ASSIGN n e) s' sec e3 s ⟹ is_mapped s' (Map n (Type vsec)) ⟹ vsec ≥ sec

Init Security Context
- This lemma states for an Init expression, the initialisation of a variable cannot be less than the security value of the expression.

lemma Init_Security_Context:

eval (INIT (Type tsec) n) s' sec e s ⟹ tsec ≥ sec

5.3 The Security Conclusion

Together with the security increasing lemmas, these show that direct information flow breaking the security flow of only low to high cannot happen, since higher security information cannot be assigned to a lower security variable.

Indirect information flow breaking security also cannot occur, since by the completeness and increasing lemmas, all conditional expressions used to influence program flow contain a higher security level always, as security can only increase through sub expressions.

Thus, information can neither directly nor indirectly flow from a higher security level information source to a lower security level destination, in any program written in the Canyon language.

This means that for all programs written in Canyon, they will by the very fact that they are written in the language, automatically inherit the property of clearance conformance.