An analysis framework for SecreC

Jaak Ristioja

Software Technology and Applications Competence Center (STACC)

June 28, 2010

Problem statement

- Operations in sharemind and SecreC are guaranteed to keep private data a shared secret
- This does not prevent the programmer from making stupid mistakes

```
toWikiLeaks = declassify(topSecretData);
```

Problem statement

- Operations in sharemind and SecreC are guaranteed to keep private data a shared secret
- This does not prevent the programmer from making stupid mistakes

```
toWikiLeaks = declassify(topSecretData);
```

Problem statement

- Operations in sharemind and SecreC are guaranteed to keep private data a shared secret
- This does not prevent the programmer from making stupid mistakes

```
toWikiLeaks = declassify(topSecretData);
```

The SecreC programming language

- Imperative, syntax based on C
- Types:
  - Data types: **`bool`**, **`int`**, **`unsigned int`**, **`string`**
  - Security types: **`private`** and **`public`**
  - Matrices and vectors
- Procedures
  - pass-by-value
  - overloading
- Statements
  - variable definitions (with initialization)
  - **`if`**, **`if-else`**, **`while`**, **`do-while`**, **`for`**
  - **`continue`**, **`break`**, **`return`**
  - expression statements
- Expressions
  - Binary +, -, *, /, %, ||, &&
  - Relational <, <=, =>, >, ==, !=
  - Unary -, !, Ternary `?:`
  - Assignments =, +=, -=, *=, /=, %=
  - Procedure calls and **`declassify`**`(e)`
  - Matrix multiplication **`#`**, element access `[e]`, `[*]`

The SecreC programming language

- Imperative, syntax based on C
- Types:
  - Data types: **`bool`**, **`int`**, **`unsigned int`**, **`string`**
  - Security types: **`private`** and **`public`**
  - Matrices and vectors
- Procedures
  - pass-by-value
  - overloading
- Statements
  - variable definitions (with initialization)
  - **`if`**, **`if-else`**, **`while`**, **`do-while`**, **`for`**
  - **`continue`**, **`break`**, **`return`**
  - expression statements
- Expressions
  - Binary +, -, *, /, %, ||, &&
  - Relational <, <=, =>, >, ==, !=
  - Unary -, !, Ternary `?:`
  - Assignments =, +=, -=, *=, /=, %=
  - Procedure calls and **`declassify`**`(e)`
  - Matrix multiplication `#`, element access `[e]`, `[*]`

The SecreC programming language

- Imperative, syntax based on C
- Types:
  - Data types: **`bool`**, **`int`**, **`unsigned int`**, **`string`**
  - Security types: **`private`** and **`public`**
  - Matrices and vectors
- Procedures
  - pass-by-value
  - overloading
- Statements
  - variable definitions (with initialization)
  - **`if`**, **`if-else`**, **`while`**, **`do-while`**, **`for`**
  - **`continue`**, **`break`**, **`return`**
  - expression statements
- Expressions
  - Binary +, -, *, /, %, ||, &&
  - Relational <, <=, =>, >, ==, !=
  - Unary -, !, Ternary `?:`
  - Assignments =, +=, -=, *=, /=, %=
  - Procedure calls and **`declassify`**`(e)`
  - Matrix multiplication `#`, element access `[e]`, `[*]`

The SecreC programming language

- Imperative, syntax based on C
- Types:
  - Data types: **`bool`**, **`int`**, **`unsigned int`**, **`string`**
  - Security types: **`private`** and **`public`**
  - Matrices and vectors
- Procedures
  - pass-by-value
  - overloading
- Statements
  - variable definitions (with initialization)
  - **`if`**, **`if-else`**, **`while`**, **`do-while`**, **`for`**
  - **`continue`**, **`break`**, **`return`**
  - expression statements
- Expressions
  - Binary `+`, `-`, `*`, `/`, `%`, `||`, `&&`
  - Relational `<`, `<=`, `=>`, `>`, `==`, `!=`
  - Unary `-`, `!`, Ternary `?:`
  - Assignments `=`, `+=`, `-=`, `*=`, `/=`, `%=`
  - Procedure calls and **`declassify`**`(e)`
  - Matrix multiplication `#`, element access `[e]`, `[*]`

The SecreC programming language

- Imperative, syntax based on C
- Types:
  - Data types: **`bool`**, **`int`**, **`unsigned int`**, **`string`**
  - Security types: **`private`** and **`public`**
  - Matrices and vectors
- Procedures
  - pass-by-value
  - overloading
- Statements
  - variable definitions (with initialization)
  - **`if`**, **`if-else`**, **`while`**, **`do-while`**, **`for`**
  - **`continue`**, **`break`**, **`return`**
  - expression statements
- Expressions
  - Binary `+`, `-`, `*`, `/`, `%`, `||`, `&&`
  - Relational `<`, `<=`, `=>`, `>`, `==`, `!=`
  - Unary `-`, `!`, Ternary `?:`
  - Assignments `=`, `+=`, `-=`, `*=`, `/=`, `%=`
  - Procedure calls and **`declassify`**`(e)`
  - Matrix multiplication `#`, element access `[e]`, `[*]`

The SecreC programming language

- Imperative, syntax based on C
- Types:
  - Data types: **`bool`**, **`int`**, **`unsigned int`**, **`string`**
  - Security types: **`private`** and **`public`**
  - Matrices and vectors
- Procedures
  - pass-by-value
  - overloading
- Statements
  - variable definitions (with initialization)
  - **`if`**, **`if-else`**, **`while`**, **`do-while`**, **`for`**
  - **`continue`**, **`break`**, **`return`**
  - expression statements
- Expressions
  - Binary `+`, `-`, `*`, `/`, `%`, `||`, `&&`
  - Relational `<`, `<=`, `=>`, `>`, `==`, `!=`
  - Unary `-`, `!`, Ternary `?:`
  - Assignments `=`, `+=`, `-=`, `*=`, `/=`, `%=`
  - Procedure calls and **`declassify`**`(e)`
  - Matrix multiplication `#`, element access `[e]`, `[*]`

The SecreC programming language

- Imperative, syntax based on C
- Types:
  - Data types: **`bool`**, **`int`**, **`unsigned int`**, **`string`**
  - Security types: **`private`** and **`public`**
  - Matrices and vectors
- Procedures
  - pass-by-value
  - overloading
- Statements
  - variable definitions (with initialization)
  - **`if`**, **`if-else`**, **`while`**, **`do-while`**, **`for`**
  - **`continue`**, **`break`**, **`return`**
  - expression statements
- Expressions
  - Binary `+`, `-`, `*`, `/`, `%`, `||`, `&&`
  - Relational `<`, `<=`, `=>`, `>`, `==`, `!=`
  - Unary `-`, `!`, Ternary `?:`
  - Assignments `=`, `+=`, `-=`, `*=`, `/=`, `%=`
  - Procedure calls and **`declassify`**`(e)`
  - Matrix multiplication **`#`**, element access `[e]`, `[*]`

Security types

```
public int pub;
private int pri;

pri = pri; // OK
pub = pub; // OK

pri = pub; // OK, classification occurs
pri = pub + pri; // OK, classification of pub

pub = pri; // ERROR

pub = declassify(pri); // OK (hopefully)
```

Security types

```
public int pub;
private int pri;

pri = pri; // OK
pub = pub; // OK

pri = pub; // OK, classification occurs
pri = pub + pri; // OK, classification of pub

pub = pri; // ERROR

pub = declassify(pri); // OK (hopefully)
```

Security types

```
public int pub;
private int pri;

pri = pri; // OK
pub = pub; // OK

pri = pub; // OK, classification occurs
pri = pub + pri; // OK, classification of pub

pub = pri; // ERROR

pub = declassify(pri); // OK (hopefully)
```

Security types

```
public int pub;
private int pri;

pri = pri; // OK
pub = pub; // OK

pri = pub; // OK, classification occurs
pri = pub + pri; // OK, classification of pub

pub = pri; // ERROR

pub = declassify(pri); // OK (hopefully)
```

Security types

```
public int pub;
private int pri;

pri = pri; // OK
pub = pub; // OK

pri = pub; // OK, classification occurs
pri = pub + pri; // OK, classification of pub

pub = pri; // ERROR

pub = declassify(pri); // OK (hopefully)
```

How to help the programmer?

Bad example

```
public int sum(private int a, private int b) {
    return declassify(a) + declassify(b);
}
```

- Both values are leaked

Good example

```
public int sum(private int a, private int b) {
    return declassify(a + b);
}
```

- Only the sum is leaked
- Not so easy to infer original values

How to help the programmer?

Bad example

```
public int sum(private int a, private int b) {
    return declassify(a) + declassify(b);
}
```

- Both values are leaked

Good example

```
public int sum(private int a, private int b) {
    return declassify(a + b);
}
```

- Only the sum is leaked
- Not so easy to infer original values

How to help the programmer?

Bad example

```
public int sum(private int a, private int b) {
    return declassify(a) + declassify(b);
}
```

- Both values are leaked

Good example

```
public int sum(private int a, private int b) {
    return declassify(a + b);
}
```

- Only the sum is leaked
- Not so easy to infer original values

How to help the programmer?

Bad example

```
public int sum(private int a, private int b) {
    return declassify(a) + declassify(b);
}
```

- Both values are leaked

Good example

```
public int sum(private int a, private int b) {
    return declassify(a + b);
}
```

- Only the sum is leaked
- Not so easy to infer original values

Current accomplishments

- Theoretical foundations for the core subset of SecreC:
  - Formal grammar
  - Formal rules for static checking
  - Formal operational semantics
- An general data-flow analyzer
  - C++ library
  - Takes SecreC source code as input
  - Able to run data-flow analyses given to it as objects
    - Accepts both forward and backward data-flow analyses
    - Can handle branched analyses
  - Can be used as a front-end for optimizing compilers
  - Can be used by IDE's (Secrecide)

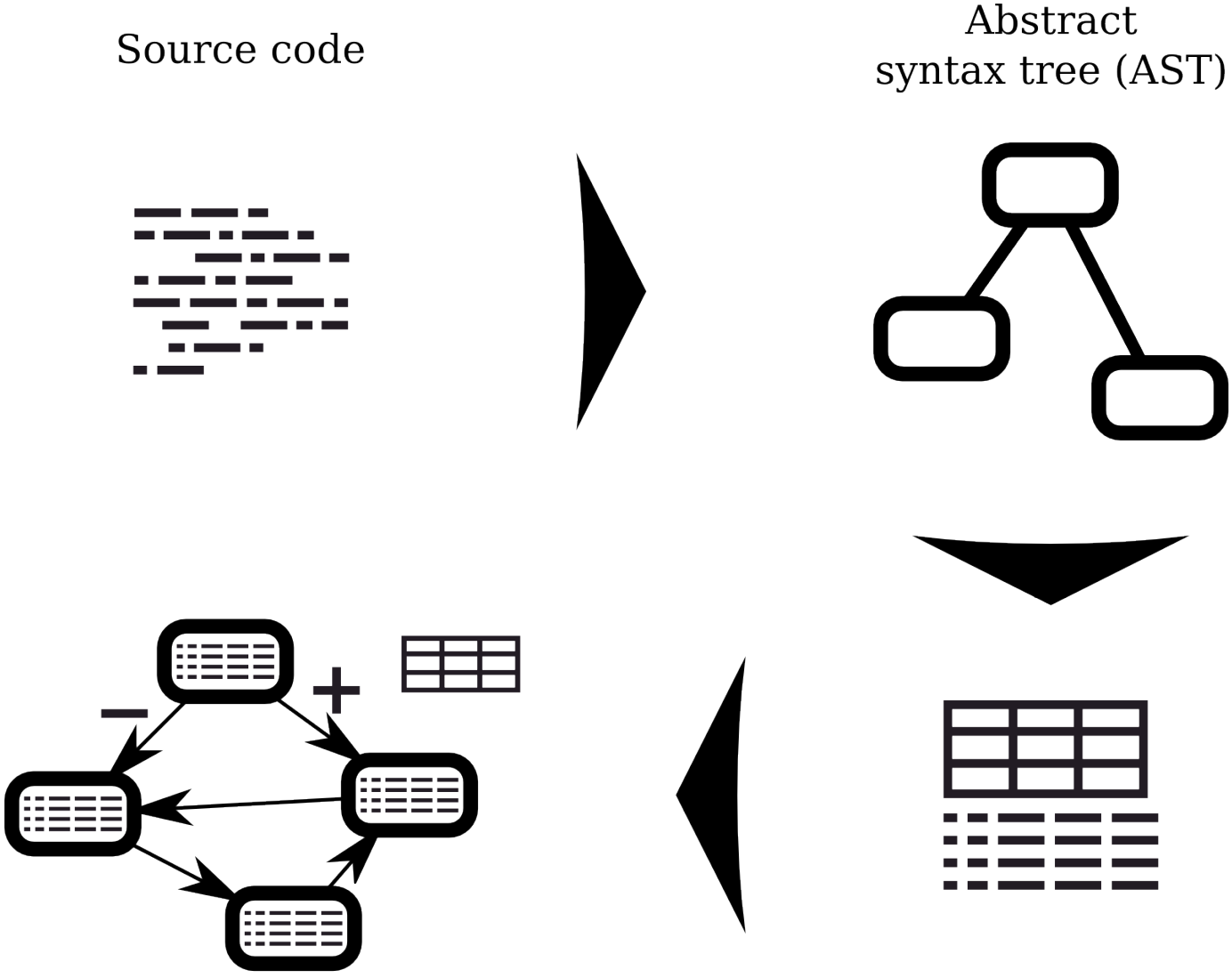
Current accomplishments

- Theoretical foundations for the core subset of SecreC:
  - Formal grammar
  - Formal rules for static checking
  - Formal operational semantics
- An general data-flow analyzer
  - C++ library
  - Takes SecreC source code as input
  - Able to run data-flow analyses given to it as objects
    - Accepts both forward and backward data-flow analyses
    - Can handle branched analyses
  - Can be used as a front-end for optimizing compilers
  - Can be used by IDE's (Secrecide)

Current accomplishments

- Theoretical foundations for the core subset of SecreC:
  - Formal grammar
  - Formal rules for static checking
  - Formal operational semantics
- An general data-flow analyzer
  - C++ library
  - Takes SecreC source code as input
  - Able to run data-flow analyses given to it as objects
    - Accepts both forward and backward data-flow analyses
    - Can handle branched analyses
  - Can be used as a front-end for optimizing compilers
  - Can be used by IDE's (Secrecide)

The SecreC Analyzer



The SecreC Analyzer

Example intermediate representation instructions:

```
d = arg1 + arg2;
d = !arg1;
d = DECLASSIFY(arg1);
if (arg1 < arg2) goto d;
POPPARAM d;
CALL d;
```

```
d = arg1;
d = CLASSIFY(arg1);
goto d;
if (!arg1) goto d;
PUSHPARAM d;
RETURN arg1;
```

Types of edges in the CFG:

- *regular* edges (fall-thru, jump)
- **`true`** and **`false`** edges (conditional jumps)
- `CALL`, `RETURN` and *call-pass* edges (procedure calls)

The SecreC Analyzer

Example intermediate representation instructions:

```
d = arg1 + arg2;
d = !arg1;
d = DECLASSIFY(arg1);
if (arg1 < arg2) goto d;
POPPARAM d;
CALL d;
```

```
d = arg1;
d = CLASSIFY(arg1);
goto d;
if (!arg1) goto d;
PUSHPARAM d;
RETURN arg1;
```

Types of edges in the CFG:

- *regular* edges (fall-thru, jump)
- **true** and **false** edges (conditional jumps)
- `CALL`, `RETURN` and *call-pass* edges (procedure calls)

Implemented analyses

- Reaching definitions
  - Not yet able to handle variables going out of scope
- *Reaching jumps*
  - Conditions that hold in parts of code

```
if (e) {
  // expression e holds
} else {
  // expression e doesn't hold
}
// We know that we tested e to get here
```

- Analysis to detect trivial information leakages
  - Some data is marked sensitive
  - Some operations make it unsensitive
  - Does sensitive data reach **`declassify`**?
  - Needs a better model

Implemented analyses

- Reaching definitions
  - Not yet able to handle variables going out of scope
- *Reaching jumps*
  - Conditions that hold in parts of code

```
if (e) {
  // expression e holds
} else {
  // expression e doesn't hold
}
// We know that we tested e to get here
```

- Analysis to detect trivial information leakages
  - Some data is marked sensitive
  - Some operations make it unsensitive
  - Does sensitive data reach `declassify`?
  - Needs a better model

Implemented analyses

- Reaching definitions
  - Not yet able to handle variables going out of scope
- *Reaching jumps*
  - Conditions that hold in parts of code

```
if (e) {
  // expression e holds
} else {
  // expression e doesn't hold
}
// We know that we tested e to get here
```

- Analysis to detect trivial information leakages
  - Some data is marked sensitive
  - Some operations make it unsensitive
  - Does sensitive data reach **`declassify`**?
  - Needs a better model

Experimental results

```
 4       private int s = getSecret(1);
 5       declassify(s); declassify(-s);
 6       declassify(0 + s);
 7       s = 0;
 8       if (e1) {
 9           if (e2) s = getSecret(1);
10           else    s = getSecret(2);
11           declassify(s);
12       }
13       declassify(s);
```

```
declassify at (5,5)(5,16)   leaks the value from:
    call to {proc}getSecret() at (4,21)(4,32)
declassify at (5,18)(5,30)  leaks the value from:
    call to {proc}getSecret() at (4,21)(4,32)
declassify at (11,9)(11,20) leaks the value from:
    call to {proc}getSecret() at (9,23)(9,34)
    call to {proc}getSecret() at (10,18)(10,29)
declassify at (13,5)(13,16) might leak the value from:
    call to {proc}getSecret() at (9,23)(9,34)
    call to {proc}getSecret() at (10,18)(10,29)
```

Future goals

- Formal grammar and semantics for the rest of SecreC
  - Extend the language
- Better analysis
  - Constant propagation
  - How much data is leaked?
- Integration with Secrecide
- A better compiler?

EOF

Questions?

EOF

Thank you for listening!