

Information flow analysis for small embedded systems

Isabelle Simplot-Ryl
join work D. Ghindici, and G. Grimaud

INRIA/LIFL/Univ. Lille 1

Paris, november 2008

Context

System and Networking for Portable Objects Proved to be Safe



Thinking POPS as an usual target for general purpose software:

Intelligence in operating system and framework instead of expertise of developers

- Hide the complexity of the exotic hardware and communication management
- Performance issues are important (from 8 to 32 bits processors, 1Kb of RAM, 64 Kb of E²PROM, 128 Kb ROM, limited electrical resources)
- Safety and security preoccupations are omnipresent: unsafe and untrusted deployment environment

Information flow

- Data confidentiality: Access control vs information flow control

Example

explicit flow

implicit flow

h (*secret variable*)
 l (*public variable*)

$l = h;$

```
if (h==0)
  l=0;
else
  l=1;
```

How to formalize the absence of information leak?

Idea: Modifying the **secret** input values of a program does not impact the **public** output values of the program

P verify the non-interference property if

$\forall h_1, \dots, h_n, \forall h'_1, \dots, h'_n, \forall l_1, \dots, l_m$, after the execution of $P(h_1, \dots, h_n, l_1, \dots, l_m)$ and et de $P(h'_1, \dots, h'_n, l_1, \dots, l_m)$, values of l_1, \dots, l_m are identical.

Safety of small autonomous embedded systems

- Openness, mobility, post insurance, . . .
- Need of static reasoning because of low performances
- Autonomicity because of potential hostile environments

Object-Oriented

- Virtual invocations \Rightarrow not possible to decide which code will be executed

Openness – Dynamic loading

- New sub-classes, new calling contexts \Rightarrow stated results on running systems must still hold
- Mobile code may come from hostile environment \Rightarrow mobile code must be "verified" on execution site (bytecode)

Object-Oriented + Open \Rightarrow Highly dynamic

Small and autonomous embedded systems

- Properties must be verifiable with few resources

Compositional static analysis based on *contracts*

A contract is associated to each method

- Depending on the property
- Natural grain for object-oriented systems

Support for dynamic loading and openness: Compositionality

- New code must respect required contracts
 - already established properties still hold
- New code uses contracts of old code
 - No need to re-analyse old code in new context

On small embedded systems

- Use of some light version of PCC to verify method contract when loading the method code
- Possibility to verify a method when the called methods are not available

Embedded verification

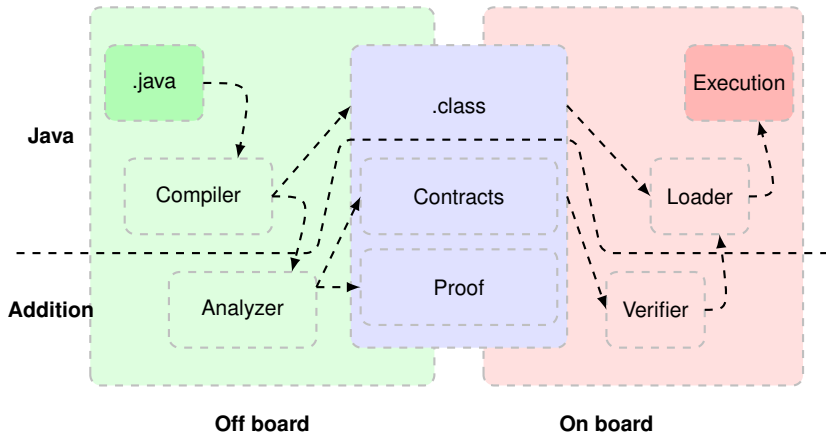
Lighthouse bytecode verification - Eva Rose

- Elegant Java bytecode verification in smart cards
- Idea: *"It is easier to verify a result than to establish it"*
- Two phases algorithm

Extension

- Extension of the technique using the contracts for all properties that have a lattice structure
- Contract repository of already loaded methods
- A method is loaded with:
 - Proof annotations
 - Necessary contracts
- Verification is then linear: a method is accepted if its contract can be verified and is coherent with the contract repository of the system

Analysis scheme



Abstract Interpretation of a method \mathcal{M}

- Lattice of "properties" as types \mathcal{T}
- Set $\Sigma_{\mathcal{M}}$ of "visible" elements of \mathcal{M}
- Set of abstract values of \mathcal{M} : $\mathcal{A}_{\mathcal{M}}$
- Extension of the notion of "signature" of methods $\mapsto \mathcal{S}_{\mathcal{M}} \subseteq \Sigma_{\mathcal{M}} \times \mathcal{T}$
- Intra-method analysis: given the semantics of instructions, following the control flow until a fix-point

$E = \{1\}$

$T = pc \times \{\mathcal{A}_{\mathcal{M}} \times \perp\}$

$S = pc \times \{S_0\}$

while $E \neq \emptyset$ *do*

extract i *from* E

apply the semantics of $PC(i)$ *to the current state*

done

$\mathcal{S}_{\mathcal{M}}$ is the projection on $\Sigma_{\mathcal{M}}$
of the unification of the
{ $T(i) \mid PC(i) = \text{return}$ }

Abstract Interpretation of a method \mathcal{M}

- Lattice of "properties" as types \mathcal{T}
- Set $\Sigma_{\mathcal{M}}$ of "visible" elements of \mathcal{M}
- Set of abstract values of \mathcal{M} : $\mathcal{A}_{\mathcal{M}}$
- Extension of the notion of "signature" of methods $\mapsto \mathcal{S}_{\mathcal{M}} \subseteq \Sigma_{\mathcal{M}} \times \mathcal{T}$
- Intra-method analysis: given the semantics of instructions, following the control flow until a fix-point

$E = \{1\}$

$T = pc \times \{\mathcal{A}_{\mathcal{M}} \times \perp\}$

$S = pc \times \{S_0\}$

while $E \neq \emptyset$ *do*

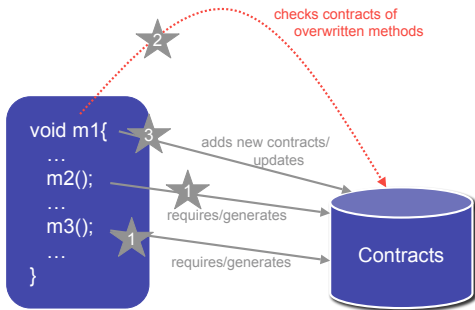
extract i *from* E

apply the semantics of $PC(i)$ *to the current state*

done

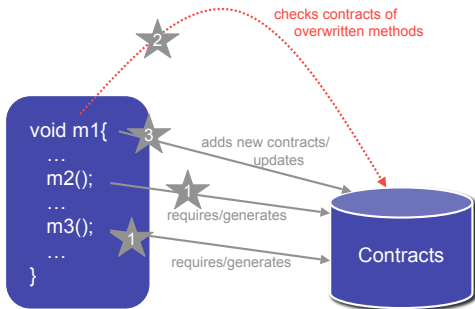
$\mathcal{S}_{\mathcal{M}}$ is the projection on $\Sigma_{\mathcal{M}}$
of the unification of the
 $\{T(i) \mid PC(i) = \text{return}\}$

Use of contracts



$$\frac{(\mathcal{V}, u_n :: \dots :: u_0 :: s, Mem, P)}{(\mathcal{V}, ret :: s, Mem', P \oplus C_m)} \quad C_m \quad \text{invoke } m$$

Use of contracts



$$\frac{(\mathcal{V}, u_n :: \dots :: u_0 :: s, Mem, P)}{(\mathcal{V}, ret :: s, Mem', P \oplus C_m)} C_m \text{ invoke } m$$

Inter-method analysis

Problems caused by openness

When analysing "invoke \mathcal{M} " on o in \mathcal{M}

- The contract of \mathcal{M}' may not be available (ex: $\mathcal{M} = \mathcal{M}'$)
- Exact type of o may not be known

Solution

- Analysis of a set of classes
 - Solve the recursivity problem
 - More flexible, add support for interfaces, abstracts classes, ...
- Use of approached contracts when the exact type is not known

Analysis of a group of classes

- Starts with all contracts at "bottom"
- Fix-point computation

Inter-method analysis

Problems caused by openness

When analysing "invoke \mathcal{M} " on o in \mathcal{M}

- The contract of \mathcal{M}' may not be available (ex: $\mathcal{M} = \mathcal{M}'$)
- Exact type of o may not be known

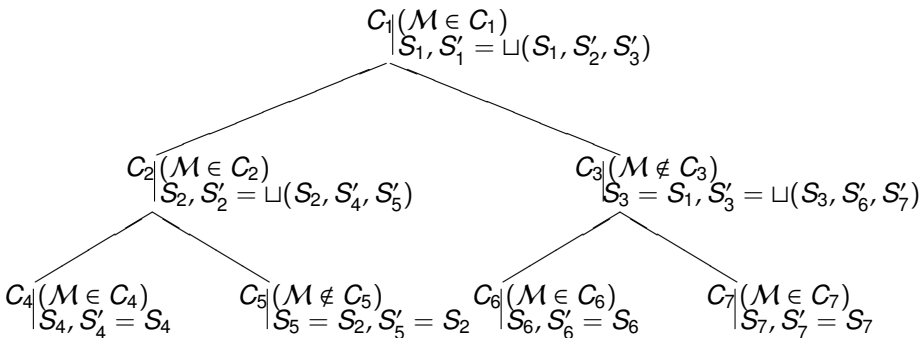
Solution

- Analysis of a set of classes
 - Solve the recursivity problem
 - More flexible, add support for interfaces, abstracts classes, ...
- Use of approached contracts when the exact type is not known

Analysis of a group of classes

- Starts with all contracts at "bottom"
- Fix-point computation

Approached signatures



Signature verification

```
0: void m(int x, A a) {  
1:   while( x > 0 ) {  
2:     this.p += a.foo(x);  
3:     this.s += x--;  
4:   }  
5: }
```

Proof elements

- JVM state for target bytecodes: Q_1^p, Q_4^p
- Signature: S_m^p
- External signatures: S_{foo}
- Fields policy

Class validation

- Compatible JVM states: $Q_1^c \leq Q_1^p, Q_4^c \leq Q_4^p$
- Compatible signatures: $S_m^c \leq S_m^p$

Signature management

Two dictionaries:

- **Temp** : temporary dictionary
- **Dico**: certified dictionary

- **Temp** : $S_{foo}^2 \sqcap S_{foo}^1$
- **Dico** : S_{foo}

Scenario

load class C

external A.foo with S_{foo}^2

load class D

external A.foo with S_{foo}^1

load class A

A.foo with S_{foo}

Example

class C

A.foo: S_{foo}^2

class D

A.foo: S_{foo}^1

class A

A.foo: S_{foo}

Class validation

$$S_{foo} \leq (S_{foo}^2 \sqcap S_{foo}^1)$$

Signature management

Two dictionaries:

- **Temp** : temporary dictionary
- **Dico**: certified dictionary

Scenario

load class C

external A.foo with S_{foo}^2

load class D

external A.foo with S_{foo}^1

load class A

A.foo with S_{foo}

Class validation

$$S_{foo} \leq (S_{foo}^2 \cap S_{foo}^1)$$

- **Temp** : $S_{foo}^2 \cap S_{foo}^1$
- **Dico** : S_{foo}

Example

class C

A.foo: S_{foo}^2

class D

A.foo: S_{foo}^1

class A

A.foo: S_{foo}

Signature management

Two dictionaries:

- **Temp** : temporary dictionary
- **Dico**: certified dictionary

- **Temp** : $S_{foo}^2 \sqcap S_{foo}^1$
- **Dico** : S_{foo}

Scenario

load class C

external A.foo with S_{foo}^2

load class D

external A.foo with S_{foo}^1

load class A

A.foo with S_{foo}

Example

class C

A.foo: S_{foo}^2

class D

A.foo: S_{foo}^1

class A

A.foo: S_{foo}

Class validation

$$S_{foo} \leq (S_{foo}^2 \sqcap S_{foo}^1)$$

Signature management

Two dictionaries:

- **Temp** : temporary dictionary
- **Dico**: certified dictionary

- **Temp** : $S_{foo}^2 \sqcap S_{foo}^1$
- **Dico** : S_{foo}

Scenario

load class C

external A.foo with S_{foo}^2

load class D

external A.foo with S_{foo}^1

load class A

A.foo with S_{foo}

Example

class C

A.foo: S_{foo}^2

class D

A.foo: S_{foo}^1

class A

A.foo: S_{foo}

Class validation

$$S_{foo} \leq (S_{foo}^2 \sqcap S_{foo}^1)$$

Signature management

Two dictionaries:

- **Temp** : temporary dictionary
- **Dico**: certified dictionary

- **Temp** : $S_{foo}^2 \sqcap S_{foo}^1$
- **Dico** : S_{foo}

Scenario

load class C

external A.foo with S_{foo}^2

load class D

external A.foo with S_{foo}^1

load class A

A.foo with S_{foo}

class C

A.foo: S_{foo}^2

class D

A.foo: S_{foo}^1

class A

A.foo: S_{foo}

Class validation

$$S_{foo} \leq (S_{foo}^2 \sqcap S_{foo}^1)$$

The verifier as a user-defined class loader

- The verifier as a JVM plug-in
 - KVM: built-in
 - Class loader hierarchy: user-defined class loader

Example

```
public class SafeClassLoader extends ClassLoader {  
    Dictionary dico;  
    ...  
}
```

- Extends the delegation model to the signatures lookup

Loading scenario

Scenario

Sc_2 loads class C
external $Sc_2.A.foo$ with S_{foo}^2
 Sc_1 loads class D
external $Sc_1.A.foo$ with S_{foo}^1
 Sc_1 loads class A
 $Sc_1.A.foo$ with S_{foo}

Class validation

$Sc_1.S_{foo} \leq Sc_1.S_{foo}^1$
 $Sc_1.S_{foo} \leq Sc_2.S_{foo}^2$

SCL: Sc_1

Classes loaded: D, A
Dico: S_{foo}
Temp Dico: S_{foo}^1

SCL: Sc_2

Classes loaded: C
Dico:
Temp Dico: S_{foo}^2

SCL: Sc_3

Classes loaded:
Dico:
Temp Dico:

Loading scenario

Scenario

Scl_2 loads class C

external $Scl_2.A.foo$ with S_{foo}^2

Scl_1 loads class D

external $Scl_1.A.foo$ with S_{foo}^1

Scl_1 loads class A

$Scl_1.A.foo$ with S_{foo}

Class validation

$Scl_1.S_{foo} \leq Scl_1.S_{foo}^1$

$Scl_1.S_{foo} \leq Scl_2.S_{foo}^2$

SCL: Scl_1

Classes loaded: D, A

Dico: S_{foo}

Temp Dico: S_{foo}^1

SCL: Scl_2

Classes loaded: C

Dico:

Temp Dico: S_{foo}^2

SCL: Scl_3

Classes loaded:

Dico:

Temp Dico:

Loading scenario

Scenario

Sc_2 loads class C
external $Sc_2.A.foo$ with S_{foo}^2
 Sc_1 loads class D
external $Sc_1.A.foo$ with S_{foo}^1
 Sc_1 loads class A
 $Sc_1.A.foo$ with S_{foo}

Class validation

$Sc_1.S_{foo} \leq Sc_1.S_{foo}^1$
 $Sc_1.S_{foo} \leq Sc_2.S_{foo}^2$

SCL: Sc_1

Classes loaded: **D**, A
Dico: S_{foo}
Temp Dico: S_{foo}^1

SCL: Sc_2

Classes loaded: **C**
Dico:
Temp Dico: S_{foo}^2

SCL: Sc_3

Classes loaded:
Dico:
Temp Dico:

Loading scenario

Scenario

Sc_2 loads class C

external $Sc_2.A.foo$ with S_{foo}^2

Sc_1 loads class D

external $Sc_1.A.foo$ with S_{foo}^1

Sc_1 loads class A

$Sc_1.A.foo$ with S_{foo}

Class validation

$Sc_1.S_{foo} \leq Sc_1.S_{foo}^1$

$Sc_1.S_{foo} \leq Sc_2.S_{foo}^2$

SCL: Sc_1

Classes loaded: **D, A**

Dico: S_{foo}

Temp Dico: S_{foo}^1

SCL: Sc_2

Classes loaded: **C**

Dico:

Temp Dico: S_{foo}^2

SCL: Sc_3

Classes loaded:

Dico:

Temp Dico:

Loading scenario

Scenario

Scl_2 loads class C
external $Scl_2.A.foo$ with S_{foo}^2
 Scl_1 loads class D
external $Scl_1.A.foo$ with S_{foo}^1
 Scl_1 loads class A
 $Scl_1.A.foo$ with S_{foo}

Class validation

$Scl_1.S_{foo} \leq Scl_1.S_{foo}^1$
 $Scl_1.S_{foo} \leq Scl_2.S_{foo}^2$

SCL: Scl_1

Classes loaded: **D, A**
Dico: S_{foo}
Temp Dico: S_{foo}^1

SCL: Scl_2

Classes loaded: **C**
Dico:
Temp Dico: S_{foo}^2

SCL: Scl_3

Classes loaded:
Dico:
Temp Dico:

Confidential data

- Resides in instance fields of persistent objects (e.g. PIN code, fidelity points, SSN)
- Field independent and *security level sensitive* analysis
 - all the fields of an object having the same security level are modeled as having the same location
- $L = \{s, p\}$: security levels

$$o^s = \{o.f_1 \dots f_n \mid \exists 0 < i \leq n, \mathcal{L}(\mathcal{T}(f_{i-1}), f_i) = s\}$$

$$o^p = \{o.f_1 \dots f_n \mid \forall 0 < i \leq n, \mathcal{L}(\mathcal{T}(f_{i-1}), f_i) = p\}$$

$\mathcal{L}(C, f) \in L$: security level of $C.f$

$\mathcal{T}(f)$: class type of f

Typing flows

- $\mathcal{F} = \{\mathbf{i}, \mathbf{v}, \mathbf{r}\}$
 - *reference* (\mathbf{r}) : alias
 - *value* (\mathbf{v}) : data transfer between primitive types
 - *implicit* (\mathbf{i}) : implicit flow
- $a^{s,p} \xrightarrow{\mathbf{r}, \mathbf{v}, \mathbf{i}} b^{s,p}$
- Example (on source code)

```
void m(int i, int b, A a) {  
    if(b)  
        this.x=i; //x public  
}
```

$$\begin{array}{l} p_0^p \xrightarrow{\mathbf{i}} p_2^p \\ p_0^p \xrightarrow{\mathbf{v}} p_1^p \end{array}$$

Security lattice

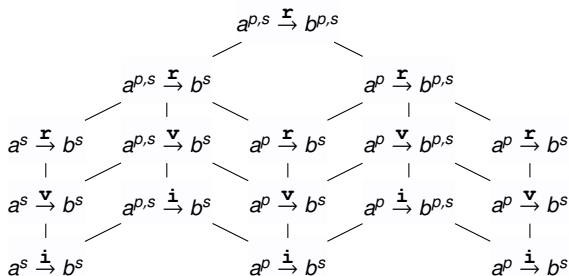
Order relations

$$p \leq p, s$$

$$s \leq p, s$$

$$\mathbf{i} \leq \mathbf{v} \leq \mathbf{r}$$

- lattice of 255 "possible flows"
- one byte to encode flows



Intra-method analysis

Flow signature of method m : S_m

- Potential flows generated by the execution of m
- Flows: between objects that survive method's execution (parameters) and some abstract values (*Static* for static fields, *R* for return, *IO* for I/O, *Ex* for exceptions)
- Context insensitive \Rightarrow results exploited *aposteriori*
- Bytecode abstract interpretation

```
void m(int i, int b, A a){  
    if(b)  
        this.x=i; //x public  
}
```

$$S_m = \{p_0^p \xrightarrow{\mathbf{i}} p_2^p, p_0^p \xrightarrow{\mathbf{v}} p_1^p\}$$

Results for IF

Benchmark	Classes	Methods	Prover (external)					Verifier (embedded)			
			Class iterations	Bytecode iterations	Analysis time (s)	Average memory (Kb)	Maximum memory (Kb)	Execution time CL (ms)	Verification time SCL (ms)	Average memory (Kb)	Maximum memory (Kb)
Dhrystone	5	21	3	1.47	5.4	4.26	35.94	121	402	0.78	3.80
fft	2	20	3	1.82	6.8	1.72	7.98	58	211	0.67	3.33
_201_compress	12	43	3	2.21	7.7	3.52	20.84	321	364	0.85	4.31
_200_check	17	109	4	1.20	15.2	5.04	34.60	128	810	1.26	6.64
crypt	2	18	3	1.66	9.8	2.22	20.78	72	268	0.83	6.96
lufact	2	20	3	2.31	3.9	4.35	23.33	526	359	0.81	2.29
raytracer	12	72	5	1.85	8.7	2.54	25.23	80	544	0.84	3.33
Pacap	15	92	4	1.06	7.5	6.62	92.84	30	385	1.01	6.01

Results for IF

Benchmark	Initial class size (Kb)	Annotated class (Kb)	Signatures (%)	Labels proof (%)	External methods (%)	External fields (%)
Dhrystone	8.2	14.4	8.00	52.22	5.15	0.20
fft	6.8	15.1	16.60	91.09	7.23	0
_201_compress	20.1	28.3	3.95	23.66	4.36	0.34
_200_check	46.3	97.7	12.27	85.05	6.75	0.04
crypt	7.0	17.0	12.27	118.28	5.90	0.07
lufact	9.3	17.0	8.44	64.31	4.07	0.39
raytracer	24.0	42.8	20.44	36.12	12.06	0.57
Pacap	26.8	52.0	18.36	55.72	9.03	0.37

Open questions

- Relate the formal framework and the concrete analysis? Class of approximation relation ?
- Extend the DSL for security policies:

```
 $T ::= (Class|Package)[, T]$   
 $R_s ::= T_1 \text{ shares with } T_2;$   
 $R_p ::= T \text{ strict secret ;}$   
 $P ::= (R_s|R_n)[, P]$ 
```

- Increase usability:
 - Declassification
 - Data structures: array, hashtable, ...
 - Input/output flux