Lightweight verification of customizable policies for mobile devices through program slicing

Gilles Barthe[1] Juan Manuel Crespo[1,3] Germán Puebla[2]
César Sanchez[1,2]

IMDEA Software

CLIP, Technical University of Madrid

National University of Rosario

August 29, 2008

1. Introduction
2. Overview of approaches to mobile code security
3. Model Generation
4. Further Work

Importance of mobile code security

- Applications today may run anywhere, with data and code moving freely between servers, PC's and portable devices.
- Since mobile code gets executed with the privileges of the user who downloaded the code the risk of damage due to malicious or faulty mobile code is very high.
- Many of the techniques currently deployed in computer security are not effective when it comes to mobile code.

1. Introduction
2. Overview of approaches to mobile code security
3. Model Generation
4. Further Work

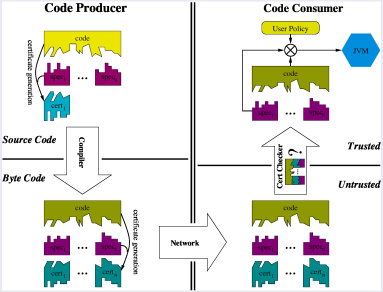
Approaches to Mobile Code Security

The ongoing use of mobile devices has motivated intensive research focused on deploying techniques suitable for ensuring security of mobile code:

- Sandboxing
- Proof Carrying Code [2, G.Necula, 1997]
- Model Carrying Code [4, Sekar et al, 2001]
- **Sound** Model Carrying Code

Proof Carrying Code

Overview



Description

- Code producer establishes security properties.
- Code producer performs a mathematical proof (certificate) stating that the code satisfies the property.
- Code consumer receives code and certificate and mechanically checks that the certificate is valid.

Caveats

- The burden of constructing the certificate is put on the code producer

Caveats

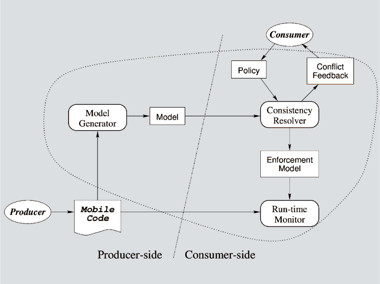
- The burden of constructing the certificate is put on the code producer
- The code producer must forsee the security needs of the consumers, which may vary widely.

Caveats

- The burden of constructing the certificate is put on the code producer
- The code producer must forsee the security needs of the consumers, which may vary widely.
- Certificate is not reusable. If the policy changes a new certificate must be constructed.

Model Carrying Code

Overview



Description

- Producer generates mobile code and program model.
- Consumer receives mobile code and model.
- Consumer mechanically checks wether the model conforms the policy.
- Based on the outcome, consumer may refine its security policy.

Caveats

- The burden of model generation is put on the consumer side.

Caveats

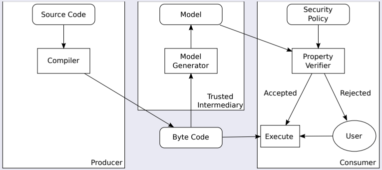
- The burden of model generation is put on the consumer side.
- The generated models in this work are unsound. Therefore, the fact that the model satisfies the security policy does not guarantee that the actual program does.

Caveats

- The burden of model generation is put on the consumer side.
- The generated models in this work are unsound. Therefore, the fact that the model satisfies the security policy does not guarantee that the actual program does.
- An enforcement model, with the consequent computational overhead, is instrumented even when the model satisfies the security policy.

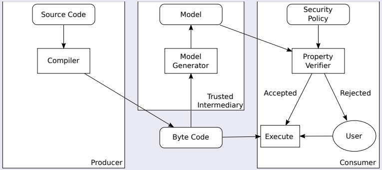
Sound Model Carrying Code

Overview



Sound Model Carrying Code

Overview



Description

- Producer writes source code and compiles it.
- Trusted intermediary generates model and digitally signs bytecode and model.
- Consumer mechanically checks wether the model conforms its policy.
- If the property is verified, program is authorized to be executed, otherwise, user may choose wether to execute it.

Contributions

A more flexible security model than PCC is proposed, that:

- allows model reuse.

Contributions

A more flexible security model than PCC is proposed, that:

- allows model reuse.
- does not put any additional burden on code producer side.

Contributions

A more flexible security model than PCC is proposed, that:

- allows model reuse.
- does not put any additional burden on code producer side.
- allows code consumer to customize the security policies.

Key Issues

How do we generate *sound* models?

We use program slicing[1, Hatcliff et al,1999]. When stating slicing correctness one proves:

$$P \equiv_C P_S$$

This notion of equivalence needs to entail soundness:

$$\Phi(P_S) \Rightarrow \Phi(P)$$

where Φ is a property.

Key Issues

Which language is suitable for expressing *high level* security properties?

We use a fragment of LTL. The idea is that the user will be able to specify properties such as:

- bounds on resource usage, e.g. this program will not send more than 3 SMS. (Safety property).
- protocol enforcement, e.g. every time a file is opened it should be eventually closed. (Liveness property).

Key Issues

How do we check wether a model conforms a security policy?

- We transform the model to obtain a finite state space model and then we perform exhaustive verification.
- This transformation is made based on the assumption that we have lower and upper bounds on the amount of iterations of every loop.
- This can be obtained either by forcing the programmer to insert annotations on loop headers or by performing conservative static analysis.
- So far we can treat loops with linear updates, and linear loop bounds

Key Issues

How do we generate the *sound* models?

We use program slicing. When stating slicing correctness one proves:

$$P \equiv_C P_S$$

This notion of equivalence needs to entail soundness:

$$\Phi(P_S) \Rightarrow \Phi(P)$$

where Φ is a property.

This will be the focus of the rest of the talk.

The Language: Syntax

The abstract syntax of the instructions is the following:

$$\begin{array}{lcl} i & ::= & nop \mid x := e \mid goto\ l \mid ifeq\ e\ l \mid call\ f(e \ldots e) \\ e & ::= & e \oplus e \mid x \mid v \end{array}$$

where

- x ranges over *Var* (variable names)
- l over *Loc* (locations)
- v over integers
- f over *Func* (function names)
- $\oplus$ represents common binary operations

The programs are abstracted as functions betwen locations *Loc* and instructions i.

The Language: Semantics

We define the structural operational semantics of the language in terms of the following:

- The state of the variables of the computation will be abstracted by
  $\Sigma = Var \rightarrow \mathbb{Z}$
- The set of non-terminal states of the program is captured by:
  $\Gamma_{NT} = Loc \times \Sigma$
- The set of terminal states of the program is captured by:
  $\Gamma_{T} = \Sigma$
- We assume the existence of a semantic function for expressions:
  $[\![\_]\!]_{Exp} : Exp \rightarrow \Sigma \rightarrow \mathbb{Z}$

The Language: Semantics (cont.)

We define the transition relation $\rightsquigarrow \subseteq \Gamma_{NT} \times \Gamma_T$ as the least relation satisfying the following rules:

$$\frac{P\ l = x := e \qquad [\![e]\!]_{Exp}\sigma \ = v}{(l,\sigma) \rightsquigarrow (succ(l), [\sigma \mid x : v])}$$

$$\frac{P\ l = goto\ l'}{(l,\sigma) \rightsquigarrow (l',\sigma)}$$

$$\frac{P\ l = ifeq\ e\ l' \qquad [\![e]\!]_{Exp}\sigma \ = 0}{(l,\sigma) \rightsquigarrow (succ(l),\sigma)}$$

$$\frac{P\ l = nop}{(l,\sigma) \rightsquigarrow (succ(l),\sigma)}$$

$$\frac{P\ l = call\ f(e_1 \ldots e_n)}{(l,\sigma) \rightsquigarrow (succ(l),\sigma)}$$

$$\frac{P\ l = return}{(l,\sigma) \rightsquigarrow \sigma}$$

$$\frac{P\ l = ifeq\ e\ l' \qquad [\![e]\!]_{Exp}\sigma \ \neq 0}{(l,\sigma) \rightsquigarrow (l',\sigma)}$$

The Language: Control Flow Graph

A control flow graph $G = (N, E, n_0, e)$ is a labeled directed graph in which:

- N is the set of nodes that represent the statements in the program.
- E is the set of labeled edges that represents the control flow between graph nodes.
- n_0 is the start node.
- e is the end node.
- We assume the programs satisfy the *unique end node property*.
- We assume the programs yield *reducible graphs*.

Slicing for model generation

- A program slice consists of the parts of the original program that potentially affect the variable values at a program point of interest[5, Weiser,1984]. The points of interest are called slicing criterion.

Slicing for model generation

- A program slice consists of the parts of the original program that potentially affect the variable values at a program point of interest[5, Weiser,1984]. The points of interest are called slicing criterion.
- We will consider the slicing criterion to be a set $C \subseteq N$ of nodes of the CFG.

Slicing for model generation

- A program slice consists of the parts of the original program that potentially affect the variable values at a program point of interest[5, Weiser,1984]. The points of interest are called slicing criterion.
- We will consider the slicing criterion to be a set $C \subseteq N$ of nodes of the CFG.
- The slicing transformation has three phases:
  1. Various forms of program dependencies between nodes are computed to form a program dependence graph (PDG).

Slicing for model generation

- A program slice consists of the parts of the original program that potentially affect the variable values at a program point of interest[5, Weiser,1984]. The points of interest are called slicing criterion.
- We will consider the slicing criterion to be a set $C \subseteq N$ of nodes of the CFG.
- The slicing transformation has three phases:
  1. Various forms of program dependencies between nodes are computed to form a program dependence graph (PDG).
  2. We compute a set that contains the nodes in C and is closed under the reflexive transitive closure of the union of the dependencies. This set is called $\mathit{Slice\ Set}(S_C)$.

Slicing for model generation

- A program slice consists of the parts of the original program that potentially affect the variable values at a program point of interest[5, Weiser,1984]. The points of interest are called slicing criterion.
- We will consider the slicing criterion to be a set $C \subseteq N$ of nodes of the CFG.
- The slicing transformation has three phases:
  1. Various forms of program dependencies between nodes are computed to form a program dependence graph (PDG).
  2. We compute a set that contains the nodes in C and is closed under the reflexive transitive closure of the union of the dependencies. This set is called *Slice Set*(S_C).
  3. We transform the instructions of the program depending on wether the node corresponding to them appears in S_c.

Program Dependencies

We will consider the following dependencies:

Program Dependencies

We will consider the following dependencies:

- Node n is data-dependent on m, written $m \xrightarrow{dd} n$, if there exists a variable v defined at node m and used at node n, without being updated in between.

Program Dependencies

We will consider the following dependencies:

- Node n is data-dependent on m, written $m \xrightarrow{dd} n$, if there exists a variable v defined at node m and used at node n, without being updated in between.
- Node n is control-dependent on node m, written $m \xrightarrow{cd} n$, if m has at least 2 successors, one from which every maximal path contains n, and the other in from which in any maximal path n does not occur, or m strcitly preceeds any occurrence of n.

Program Dependencies

We will consider the following dependencies:

- Node n is data-dependent on m, written $m \xrightarrow{dd} n$, if there exists a variable v defined at node m and used at node n, without being updated in between.
- Node n is control-dependent on node m, written $m \xrightarrow{cd} n$, if m has at least 2 successors, one from which every maximal path contains n, and the other in from which in any maximal path n does not occur, or m strcitly preceeds any occurrence of n.
- Node n is divergence-dependent on m, written $m \xrightarrow{\Omega d} n$, if m is a divergence point and there's a path in the CFG between m and n.

Program Dependencies

We will consider the following dependencies:

- Node n is data-dependent on m, written $m \overset{dd}{\rightarrow} n$, if there exists a variable v defined at node m and used at node n, without being updated in between.
- Node n is control-dependent on node m, written $m \overset{cd}{\rightarrow} n$, if m has at least 2 successors, one from which every maximal path contains n, and the other in from which in any maximal path n does not occur, or m strcitly preceeds any occurrence of n.
- Node n is divergence-dependent on m, written $m \overset{\Omega d}{\rightarrow} n$, if m is a divergence point and there's a path in the CFG between m and n.

The slice set is then defined as:

$$S_C \quad = \quad \{m : m\underbrace{(\overset{dd}{\rightarrow} \cup \overset{cd}{\rightarrow} \cup \overset{\Omega d}{\rightarrow})^*}_{d} n, n \in C\}$$

Slicing Transformation

Given program p the residual program p_s has the same nodes as p but its codemap is modified as follows:

1. forall $n \in S_C, code_2(n) = code_1(n)$.
2. forall $n \notin S_C$ we have the following cases:
   1. If $code_1(n) = goto\ m$ then $code_2(n) = goto\ m$
   2. If $code_1(n) = ifeq\ e\ m$ then $code_2(n) = goto\ k$ where k is the nearest postdominator of nodes $n + 1$ and m.
   3. If $code_1(n) = return$ then $code_2(n) = return$
   4. Else $code_2(n) = nop$

Correctness

Correct slice

Now, p_s is a correct slice of p w.r.t. C if for any initial store σ, the initial states are C-bisimilar [3, Ranganath et al,2007].

We have shown that:

- the slicing transformation defined previously produces correct slices w.r.t. to the last definition.
- the notion of correct slice entails soundness, that is, if a property is valid on the slice then is also valid on the original program.

1. Introduction
2. Overview of approaches to mobile code security
3. Model Generation
4. **Further Work**

Further work

- Extend the model conformance checking techniques: beyond linear updates in loops.
- Extend these ideas other language constructions: data types, abstraction, etc.
- More examples.
- Identify a class of properties for which this approach is also complete.
- Implementation.

M. B. Dwyer and J. Hatcliff.
Slicing software for model construction.
In *PEPM*, pages 105–118, 1999.

G. C. Necula.
Proof-carrying code.
In *POPL*, pages 106–119, 1997.

V. P. Ranganath, T. Amtoft, A. Banerjee, J. Hatcliff, and M. B. Dwyer.
A new foundation for control dependence and slicing for modern program structures.
ACM Trans. Program. Lang. Syst., 29(5), 2007.

R. Sekar, C. R. Ramakrishnan, I. V. Ramakrishnan, and S. A. Smolka.
Model-carrying code (mcc): a new paradigm for mobile-code security.

In *NSPW*, pages 23–30, 2001.

M. Weiser.
Program slicing.
IEEE Trans. Software Eng., 10(4):352–357, 1984.