

Some Tools for Software Protection

Serge Chaumette, Olivier Ly, Renaud Tabary

LaBRI – Bordeaux University

Cryscoc Workshop – 2009 june 5

SECURE PROGRAM PARTITIONING FOR HARDWARE-BASED SOFTWARE PROTECTION

Software execution protection

Goal: protect sensible parts of the software

- Confidential data
- High added value algorithm

*End user not trusted,
neither the computer on which the software is executed.*

Is-it really secure ?


Example: banking client
(that you run within i-explorer at the moment)

Obfuscation

Transform a program into a functionally equivalent **virtual black box**.

Transform a program to make it hard to be understood

- by static analysis
- by dynamic analysis



Remember
the talk of
Louis !

Widely used, but no satisfactory solution yet ..

[see *Barak and al.* « On the (Im)possibility of Obfuscating Programs »]

Executable code externalisation / Protected computing

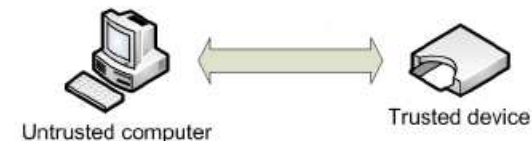
Execution is externalized to a trusted device (e.g. a smart card).

- ◆ Sensible algorithms are not given to the end user
- ◆ They are encrypted at production time

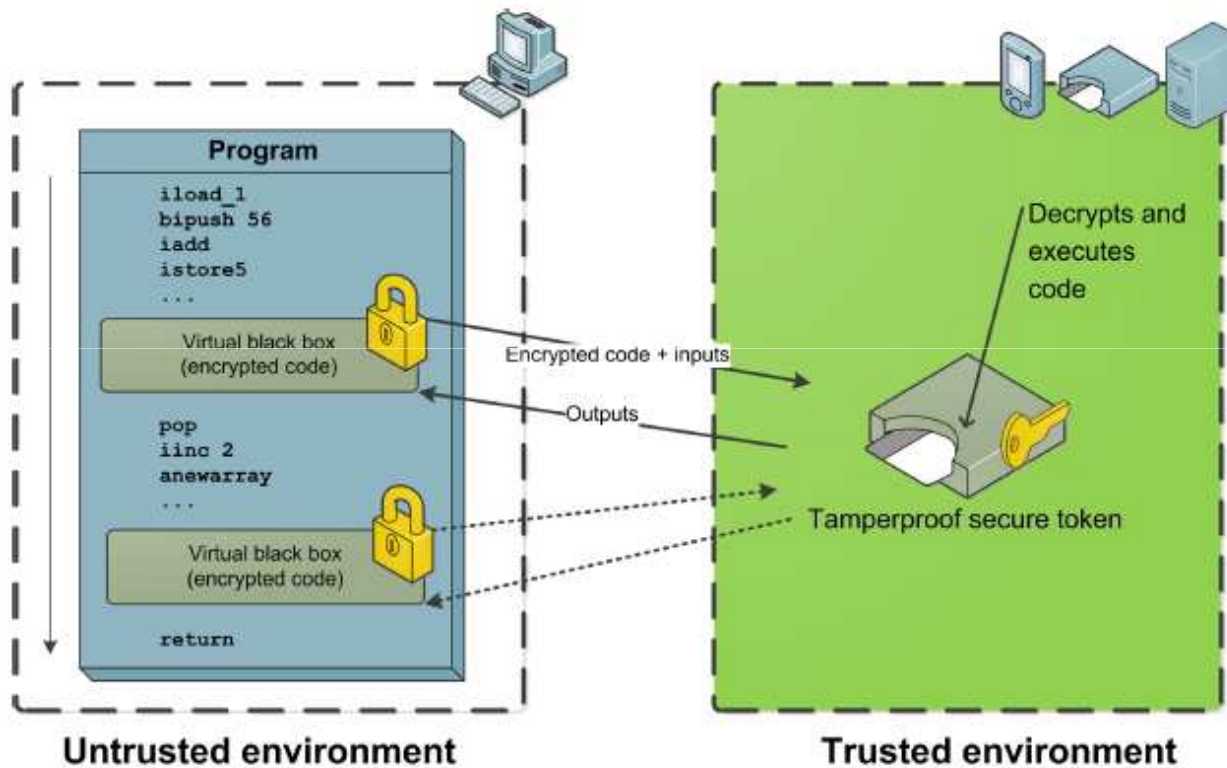


During execution,

- The public part is executed on the untrusted computer
- When a sensible processing is required:
 1. It is transmitted to the trusted device
 2. The trusted device decyphers it and entity executes it
 3. The trusted device gives back the result.



Protected Computing



Executable code externalisation / Protected computing

Idea is not new :

- I. Schaumüller-Bichl and E. Piller "A Method of Software Protection Based on the Use of Smart Cards and Cryptographic Techniques" (1984)
- Antonio Mana et al. "A framework for secure execution of software" (2004)

However some problem remains open:

- What about protection of arbitrary long function ?

Executable code externalisation

Our constraints:

- Protection should be transparent for the developer.

→ Static Analysis

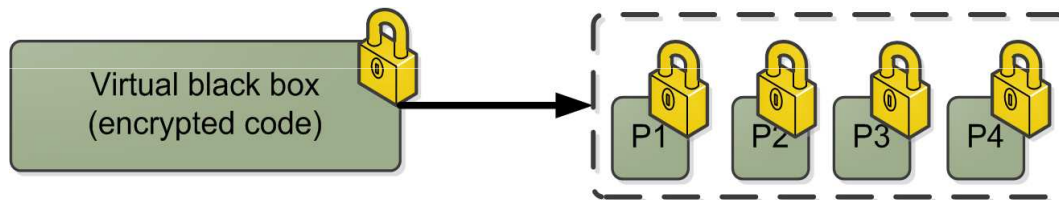
- The trusted device has a limited amount of resource in general (memory space).

→ Program Partitioning

Program Partitioning

The part of the processing to be protected is cut into small pieces.

→ Each part must fit in the trusted device

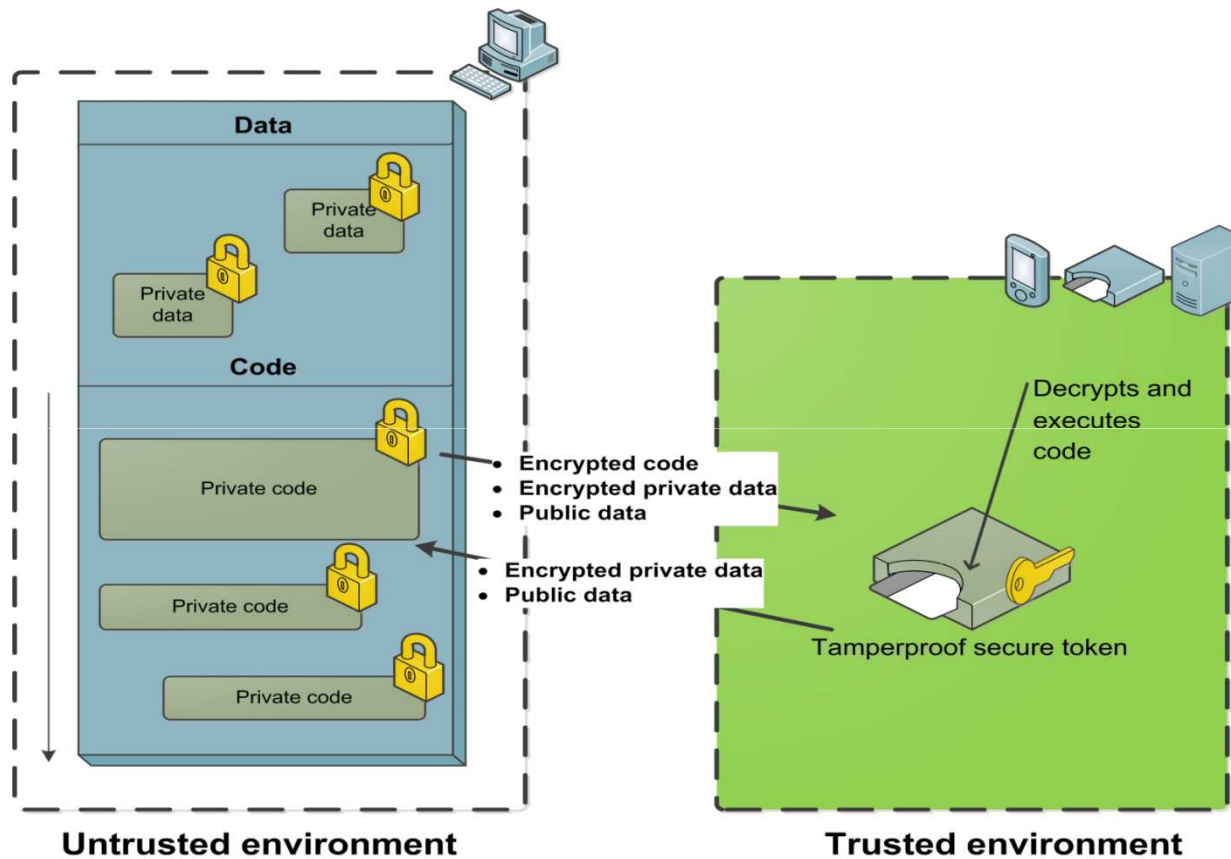


Execution of a sensible processing:

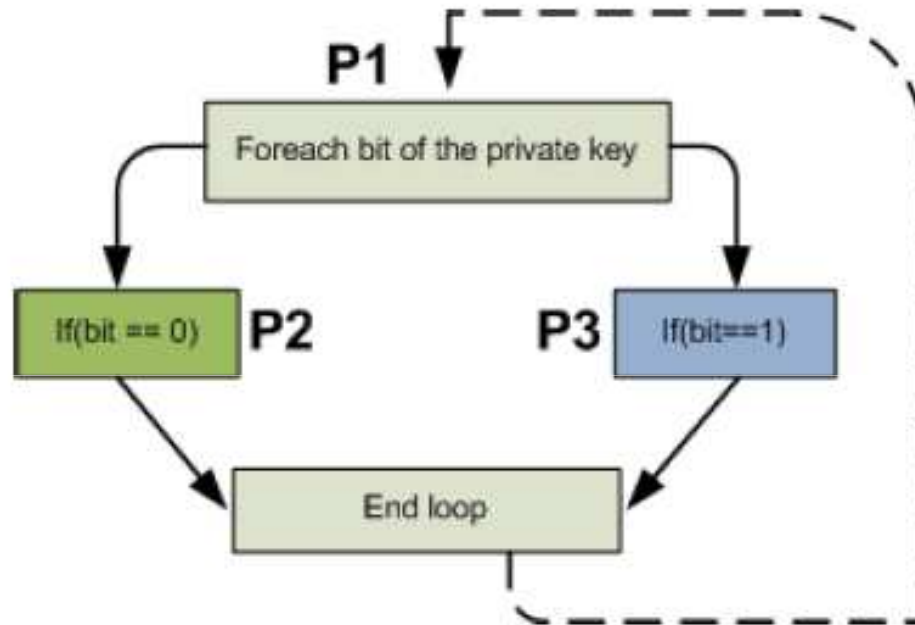
→ Pieces are transmitted one by one

The flow of piece must not leak any information !

Executable code externalisation



Program Partitioning



Code Piece Flow	P2	P3	P3	P2 ...
Related information	0	1	1	0 ...

Program Partitioning

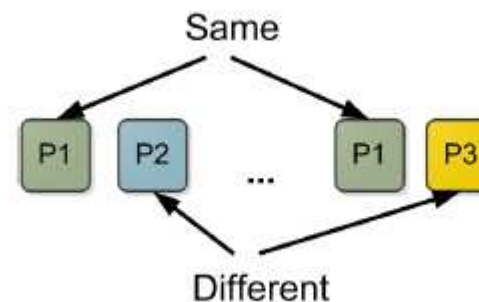
Zhang's solution

[T. Zhang "Tamper-Resistant Whole Program Partitioning" (2003)]

Compute a minimal secure partitioning that

- minimizes the partition size
- keeps private data confidential

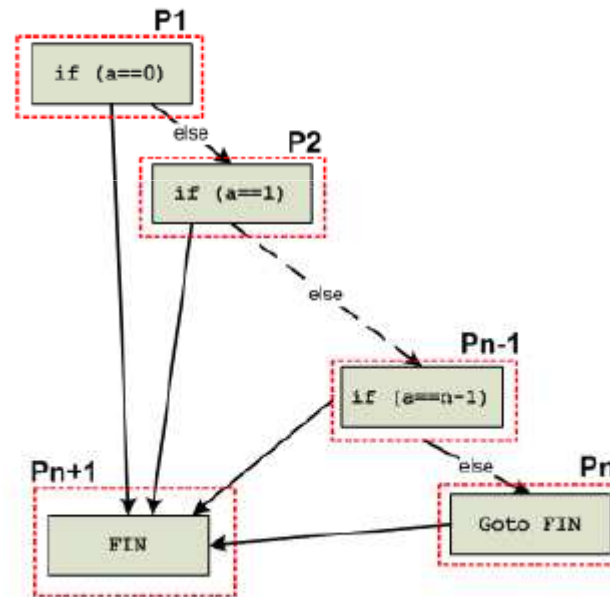
Safe partitioning : do not generate this type of sequence:



Program Partitioning

However, it remains some problems : some information leakage are not caught.

Here the number of partitions
Depends on the value of a



Program Partitioning

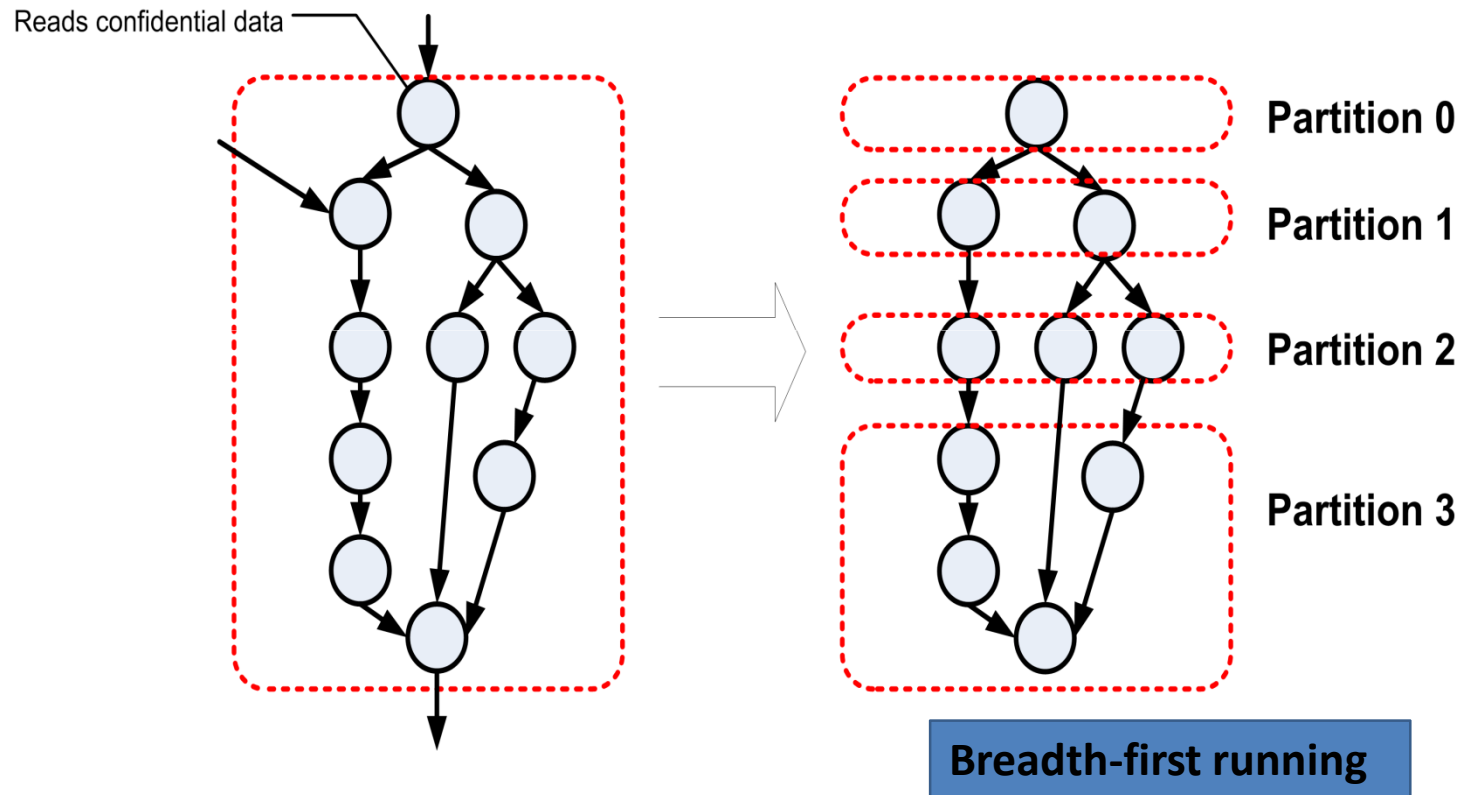
We define formally a secure program partitioning in term of non-interference:

Values of private variables do not interfere with the sequence of partitions.

(but public information may leak)

1. Identify private data (static analysis)
2. Identify code where control flow depends on private data
3. Partition these blocs in a control flow independent manner

Program Partitioning



Static Analysis

The developer points:

- Sensible functions
- Sensible data

Static analysis for data/function which can leak sensible data:

- Undecidable : one computes an upper approximation of the set of assets to be hidden.

Static Analysis

We consider pairs (L,P) where

- L is a **l-value**
- P is a **trace target** (a program point + execution history).

We compute dependencies as regular languages:

$$(L,P) \leftarrow (L', LP)$$

where LP is a regular language of trace targets describing the set of trace target where L depends on L'.

Current work: represent approximation of dependencies by rational transducers.

- ◆ We can consider abstractions of trace target (e.g. just a target point, or no loop)
⇒ compromise between efficiency / approximation

Implementation

Target language : **Java Bytecode**

- Bytecode Static Analysis
- Program partitioning
- Original code automatic modification
- Simulation in JCATools In progress
- Modification of the embedded Virtual Machine In progress

Analysis of our solution

- Static analysis of java programs – public/private parts
- Data protection
- Partition size remains small
- No information leaks about private data
- Public information may leaks !

**THE OPPOSITE WAY :
A TOOL FOR UNDERSTANDING OBFUSCATED
PROGRAMS**

Static Analysis of executable code

Desobfuscation !

- Viruses try to escape detection by obfuscating their own code. A virus may obfuscate it-self iteratively in order to hide its footprint.

Goal : Semi-automatic Semantic Analysis for the desobfuscation.

Static Analysis of executable code

Obstacles to static analysis of executable code:

- **Un-structured** programs: no explicit loops, no types, jumps, etc..
- **Dynamic jumps** forbid a classical - global analysis : the control flow is discovered step by step during analysis.

The structure of the program is not known and even not computable

Static Analysis of executable code

- Computation of the (uncomplete) Control Flow Graph
- Semantic of each elementary bloc : BDD
- Simplification of blocs (desobfuscation)
- Linear memory model
- Method of « weakest-precondition », invariants and induction proofs.

Perspectives:

- *Concolite execution* → *BINCOA ANR Project*.
- Use of *generic microcode*

Thank you !