Secure program partitioning for hardware-based software protection

RE-TRUST'08 - Trento Italy (15-16 October 2008)

LaBRI - CNRS - Université Bordeaux 1 (Talence, France)

This work is partialy supported by CNRS, Region Aquitaine and Cryscoe ANR

Serge Chaumette *chaumette@labri.fr*
Olivier Ly *ly@labri.fr*
Renaud Tabary *tabary@labri.fr*

Outline

1. HW/SW software protection
2. Data Protection
3. Secure program partitioning
4. Conclusion

Software protection

Goal

Our goal is to protect a (part of a) software application against :

1. Analysis of the program
2. Tampering of code and/or data

Applications:

- Intellectual property protection
  - Protect high-value algorithms
- Protect software against illegal modifications
  - Game cheating, malwares, license verification bypass ...

Software protection and RE-TRUST

RE-TRUST challenge

"How to ensure that a trusted program is running unmodified on an untrusted computer ?"

source: re-trust project

How software protection could contribute to RE-TRUST

- Protect critical parts of the software against tampering
- Protect the tag generation algorithm
- Protect the whole software ...

Software obfuscation

- Most of current software protection schemes are based on software **obfuscation**

Definition

Goal of obfuscation: transform a program into a functionnaly equivalent **virtual black box**

- Transform a program to make it hard to understand
  - By static analysis
  - By dynamic analysis
- Widely used ... but no satisfactory solution yet

! *Barak et al. - "On the (Im)possibility of Obfuscating Programs" (2001)*

HW/SW obfuscation

Solution: Hardware/Software obfuscation :

- The idea is to use a tamperproof trusted token along with the untrusted computer
- Trusted computing
  - The trusted token **validates** the software before it is executed on the untrusted computer
  - Not very flexible
- "Static" hardware protection
  - At production time, a critical part of the program is written into the trusted device
  - This critical part will be executed on the device, and thus stays protected
  - Not flexible, one application $\Rightarrow$ one device
- **Protected computing**

Protected computing

- The software is divided into two parts:
  - A **public part**, containing the low-value functions of the program
  - A **private part**, holding the critical functions of the software, that will be executed in the secure token
- No information on protected functions (besides input/output) can be obtained from the untrusted environment

⇒ Protected functions are **virtual black boxes**

How it works ?

- The private part of software is encrypted at production time with the token secret key
- The public part is executed on the untrusted computer
- At run-time, when a protected function needs to be executed:
  1. The encrypted function code as well as its inputs are sent to the trusted token
  2. The tamperproof hardware decrypts the code and executes it
  3. The outputs of the function are sent back to the untrusted computer

Protected computing

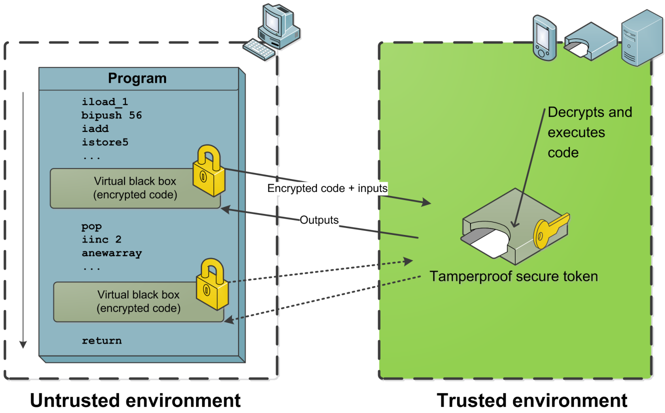


Figure: Protected computing

Open problems

- The 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)
- Nevertheless, some problems remain open:
  - What about data protection ?
  - What about protection of arbitrary-long functions ?

Outline

1. HW/SW software protection
2. Data Protection
3. Secure program partitioning
4. Conclusion

Data protection

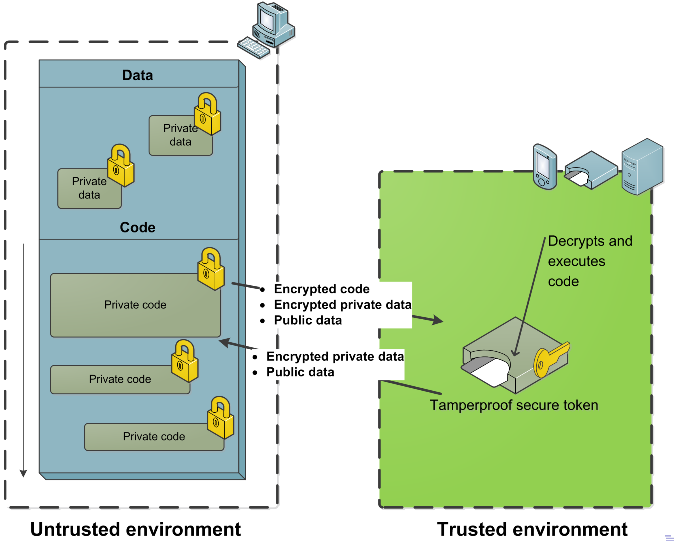
Our proposal: data protection

- The software manufacturer identifies
  1. The critical functions of the program
  2. The critical data of the software
- Then, the set of *private data* is computed
  - How: information flow analysis of the program
  - What: all data that could **leak information** of critical data
- Finally, the *private code* part is computed
  - Critical functions previously identified
  - Code reading or writing private data (`priv_var+=1`)
  - Code that **depends on** private data
    `if(priv_var==1){ ... } else{ ... }`

How it works

- Like private code, private data are stored **encrypted** on the untrusted host
- At execution time, when a private code block needs to be executed:
  - Encrypted code as well as needed data will be sent to the trusted device
  - The trusted device will decrypt private code and data
  - The protected code will then be exectued on the device
  - The modified data will be sent back to the untrusted computer
    - Public data + **Encrypted** private data
- **No information** on private code and data **leaks from the untrusted environnement**

Data protection: execution time

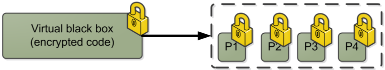


Outline

1. HW/SW software protection
2. Data Protection
3. Secure program partitioning
4. Conclusion

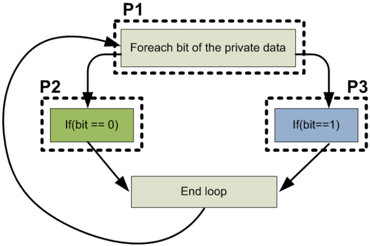
Considering limited devices

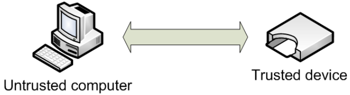
- Affordable tamperproof devices are often very limited
  - Smartcards: $\simeq$ 4ko RAM
  - Protected code blocks may be bigger
- A solution would be to divide each protected code block into **small parts**



- However, simple partitioning may reveal control flow
- ... and control flow may reveal private data !

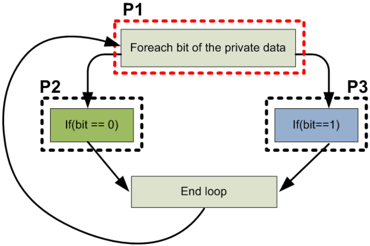
Example

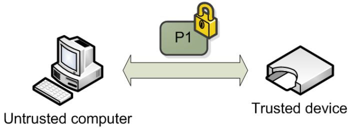




Attacker's view:

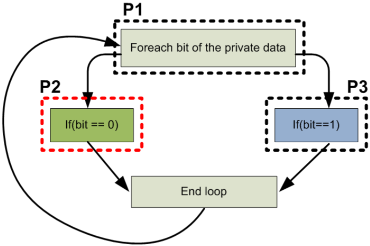
Example

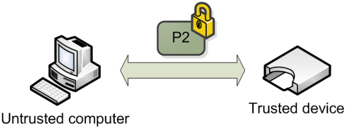




Attacker's view: P_1

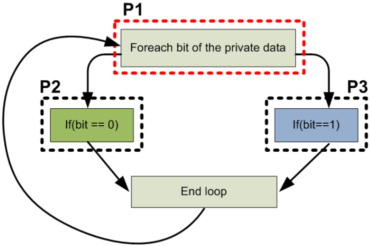
Example

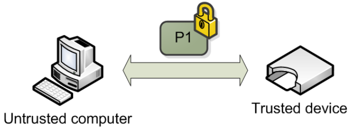




Attacker's view: $P_1 P_2$

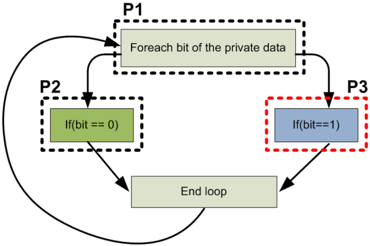
Example

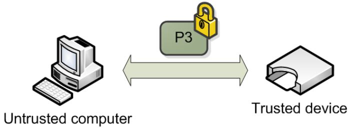




Attacker's view: $P_1 P_2 P_1$

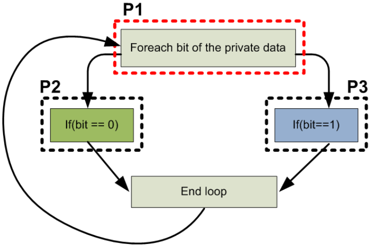
Example

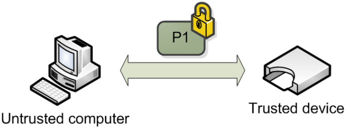




Attacker's view: $P_1 P_2 P_1 P_3$

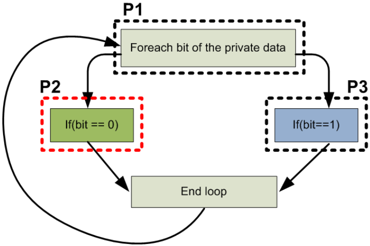
Example

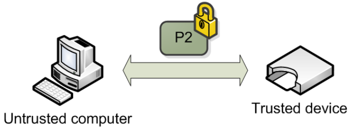




Attacker's view: $P_1 P_2 P_1 P_3 P_1$

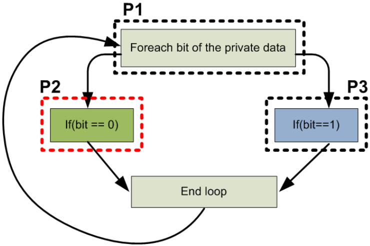
Example

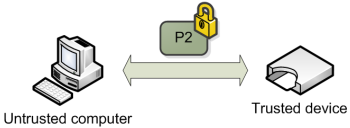




Attacker's view: $P_1 P_2 P_1 P_3 P_1 P_2$

Example



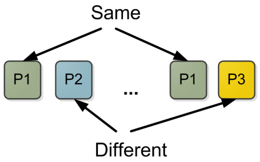


Attacker's view: $P_1\underline{P_2}P_1\underline{P_3}P_1\underline{P_2} \Rightarrow key = 010\ldots$ or $101\ldots$

Zhang's solution

- Solution is to compute a *minimal secure partitioning* that
  - minimizes partition size
  - keeps private data confidential
- T.Zhang "*Tamper-Resistant Whole Program Partitioning*" (2003)

  Unsafe partition sequence:

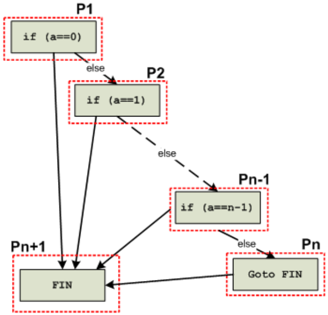


- Safe partitioning :
  - Do not generate this type of sequence

  ⇒ algorithm: do not cut loop bodies

Counter-example

- Do not catch all information leakages
- Counter example:



$\Rightarrow$ value of variable a $\simeq$ number of sent partitions

Secure partitioning

- T.Zhang's solution is not secure
- What we have done:
  - Formal definition of a secure partition flow
  - Formally proved secure partitioning algorithm
- What is a secure partition flow ?
  - A partition sequence should not leak information about **private** data
  - A partition sequence should be *independant* from private data
  - ... while public data may leak
- Partitioning algorithm:
  1. Identify code where control flow reveals private data (static analysis)
  2. Partition these blocks in a control-flow independant manner

Example

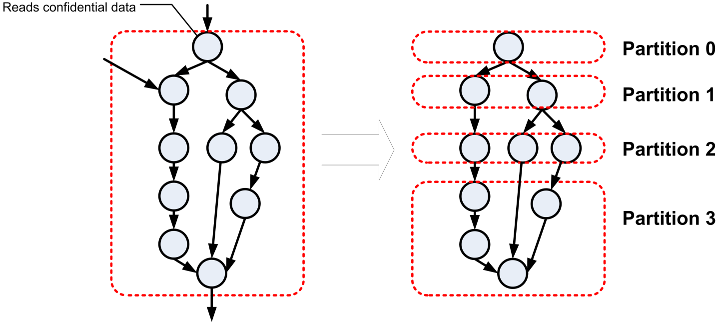


Figure: Example of a minimal secure partitioning. Partitions traffic will not depend on critical data.

Analysis of our solution

- Partitions size stays small
- No private information leaks from the untrusted environment
  - A partition sequence leaks no private data
  - Code and data are kept encrypted on the untrusted environment
- Some information may leak:
  - Public data
  - Some control flow information of private code
    - Existence of a loop, of a condition block
    - Loss of the virtual blackbox property
    - Is it really unsecure ?

Outline

1 HW/SW software protection
2 Data Protection
3 Secure program partitioning
4 Conclusion

Current work

- A proof of concept, `JCaProtect`, is under development
- Application to the protection of Java executables:
  - SAJE: Static Analysis for Java Executables
  - JCaExternalizer: partitioning and encryption
  - Lightweight Java interpreter hosted on the secure token
- Non intrusive: protection of java **object code**
- Cheap: partitioning allows the use of small secure devices
- Drawbacks
  - Performances: tests in progress
  - Partitioning not always feasible

Conclusion

- Effective software solution based on a hard problem
    - Reverse engineering of tamper-resitant devices
- Improvement of *protected computing*
    1. Data protection
    2. Externalization of functions unlimited in size
    3. Can be used on cheap tamper-resitant devices (smartcards, smartphones)
- Proof of concept under development

Questions

Questions ?