Software Protection
Techniques for software and IP protection

Leonardo Regano

leonardo.regano@unica.it

Industrial Software Development

University Of Cagliari, Italy

Attacker model: Man-at-the-End

- an attacker that has full access and privileges on one endpoint
  - physical access to devices where the software runs
    - full control on all the components
  - unlimited access to analysis tools
    - static analysis: disassemblers, decompilers
    - dynamic analysis: debuggers, fuzzers
    - symbolic analysis, concolic analysis
    - simulators, virtualizers, emulators
    - full control of the central memory
    - side channel, fault injection
    - dedicated HW
  - tools are indispensable as they represent data in useful way
    - the human mind is the bottleneck
    - control flow graph, data dependency graph, call graph, symbolic/concolic states

Part of this presentation is based on the slides presented by Prof. Cataldo Basile in the Security Verification and Testing course at Politecnico di Torino.

Software protection

- …protect the assets in software applications
  - property of the developing company
  - reputation, marketing
- the most important assets?
  - intellectual property
    - algorithms, methods, architectures, protocols, patents
  - data
    - private, sensitive, personal, …
    - secrets, cryptographic secrets, passwords, …
  - other company values
    - GDPR, production halted
  - software protections mitigate risks associated to software attacks

Practical principles of software protection

most attackers are driven by the monetization

"if your code is too complex to attack, I'll find another SW"

defenders' aim is...

....discourage attackers by giving the (maybe true) impression that your software is well-protected and it will be hard to compromise it...
...so that they will compromise the code of some other companies…

Mors tua vita mea

Software protections: categorization

- by the attack steps prevented
  - anti-reverse engineering
    - avoid the use of specific classes of tools
      - ... without tools no way to finish an attack task in time
  - obfuscation
    - make the program much more difficult to understand for human beings
  - anti-tampering
    - avoid, detect or even react to non-authorized changes to program code or behaviour
- by where the protection is applied
  - online (remote) vs. offline techniques (local)
- by the abstraction where they operate
  - source code vs. binaries

How to evaluate protections

- Collberg introduced the idea of potency
  - just an abstract measure
    - tell how good the protection is
    - however, there is not a formula to measure it
  - two approaches
    - 1. objective metrics
      - LOC, Hasted complexity, cyclomatic complexity, I/O calls, etc.
        - » up to 44 theoretical metrics introduced in a recent paper
      - potency → formula based on objective metrics
    - 2. empirical experiments
      - controlled experiment that involve people (e.g., students)
        - » measure the times and the successes and derive evaluation of the effectiveness

https://iris.polito.it/retrieve/handle/11583/2747308/265778/190725_EMSE_AssessmentCodeSplitting.pdf

Overhead

- protection does not come for free
  - all the protections add several forms of overhead
- overheads compared to the original application
  - complex code is not as optimized as the original one
  - pieces of bogus code
  - pieces of code for checking the integrity
  - communications with remote servers
  - new data added only needed for the protections
  - switching to other processes for anti-tampering code, built-in debuggers
- overhead depends on both protections and original code
  - bandwidth, CPU cycles, memory, often
  - ...then software developers focus on user experience

Reverse engineering

- “the process of extracting the knowledge or design blue-prints from anything man-made”
- common practice in numerous fields
  - mechanical engineering
  - biology
  - military
- software reverse engineering a.k.a. program comprehension a.k.a program understanding
  - “the process of identifying software components, their inter-relationships, and representing these entities at a higher level of abstraction ”

Reverse engineering

- can be legitimate
  - a sw developer that must use a poorly documented API of a open-source library
- from a legal standpoint, it's legitimate…
  - …unless explicitely forbidden in sw EULA…
  - …but EU/US software rights law allow it for interoperability purposes
    - e.g. Microsoft SMB (Server Message Block) → Samba in Linux-based OSes
- still, we must protect software against it
  - anti-reverse engineering protections a.k.a software obfuscation

Obfuscation

- family of protection techniques that aim at reducing the understandability of the code
  - they aim at delaying the attacker
  - high-level methods and principles are well-known and stable
  - new versions (i.e., implementation) are presented
    - security-through obscurity by company
    - few public obfuscators
      - diablo (Ghent university) for binaries
      - tigress for source code (University of Arizona at Tucson)
      - LLVM also has a trivial obfuscator
  - the dream of perfect obfuscation has been rejected by a 2001 paper
    - “On the (im)possibility of obfuscating programs”
      - i.e., there are functions that cannot be obfuscated
  - obfuscation is also a form of anti-static analysis protection

iacr.org/archive/crypto2001/21390001.pdf

Obfuscation: aims and categories

- code obfuscation purposes
  - make the control flow unintelligible
    - control flow flattening
    - branch functions
    - hide external calls
    - add bogus control flow: opaque predicates
  - manipulate functions to hide their signatures
    - split/merge
  - avoid static reconstruction of the code, force dynamic analysis
    - just-in-time techniques, virtualization obfuscation, self-modifying code
  - analysis
    - anti-taint analysis, anti-alias
- data obfuscation
  - simple forms that hide constants and values
  - white-box cryptography to hide keys in code

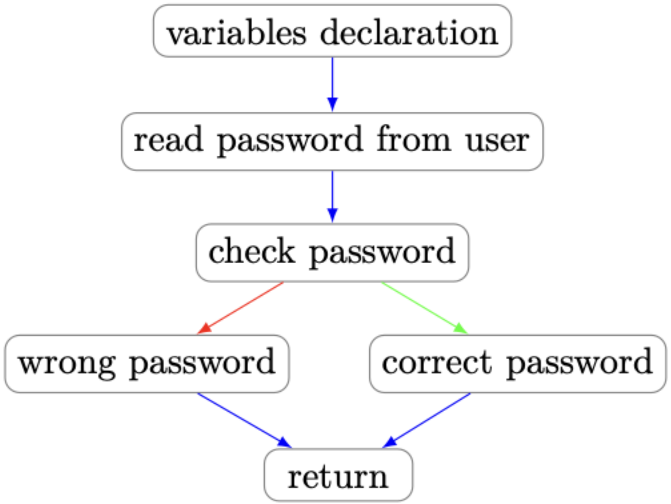
Control Flow Flattening (CFF)

- transform the code so that it hides its original control flow
  - increase the time and effort the attacker needs to understand the protected function logic
  - force attackers to run dynamic analysis
  - while usually CFG obtained with static analysis
- other technicalities
  - different types of "dispatch," i.e., how the next block is selected
    - switch, goto, indirect, call
  - the order of blocks can be randomized
- basic blocks kept intact or split up into statements

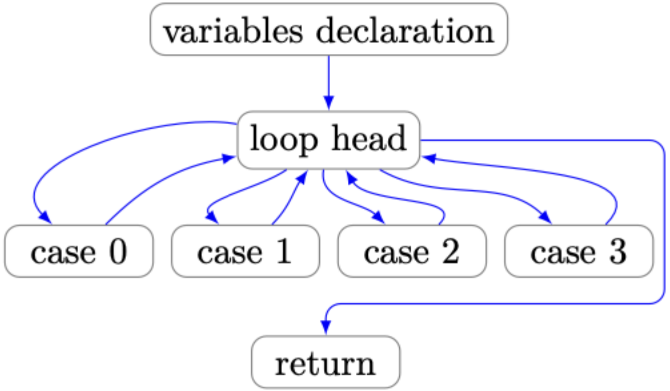
Control Flow Flattening (CFF): example

```c
#include <stdio.h>
#include <string.h>

char pwd[] = "hardcodedPassword";

int main()
{
        char temp[20] = "";
        printf("Insert password: ");
        scanf("%20s",temp);

        if(strcmp(temp,pwd)==0)
                printf("Correct password!\n");
        else
                printf("Wrong password!\n");
        return 0;
}
```

Control Flow Flattening (CFF): example



(a) original



(b) flattened

Control Flow Flattening (CFF): example

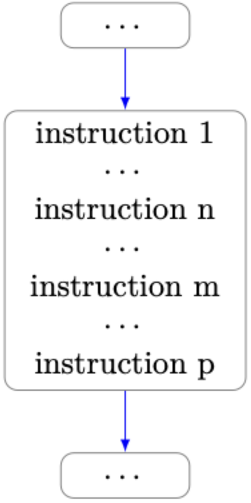
```c
#include <stdio.h>
#include <string.h>

char pwd[] = "myPassword";

int main()
{
        char temp[20] = "";
        int strcmp_result = 0;
        int control = 0;

        while (control != 4) {
                switch (control) {
                case 0:
                        printf("Insert password: ");
                        scanf("%20s", temp);
                        strcmp_result = strcmp(temp,pwd);
                        control = 1;
                        break;
                case 1:
                        if (strcmp_result == 0)
                                control = 2;
                        else
                                control = 3;
                        break;
                case 2:
                        printf("Correct password!\n");
                        control = 4;
                        break;
                case 3:
                        printf("Wrong password!\n");
                        control = 4;
                        break;
                }
        }

        return 0;
}
```

Opaque Predicates

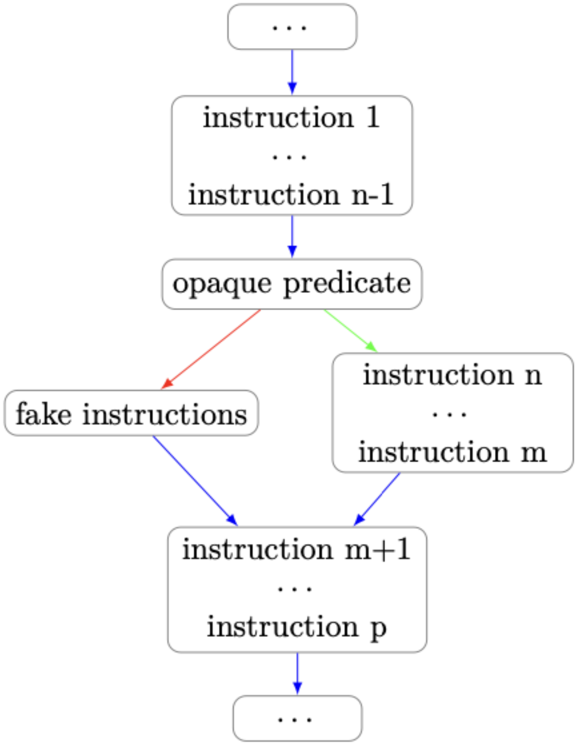
- boolean expressions that have always the same outcome at run-time
  - e.g., always true or always false
- their outcome is difficult to evaluate in a static way
  - e.g., by deobfuscators
- if employed as the condition of a branch
  - difficult to take the branch taken without executing the program
- fuzzying the program can indicate the likely presence of an opaque predicate
  - but without any formal assurance

Opaque Predicates: fake code insertion

- we can leverage opaque predicates to insert fake code
  - in order to increase the amount of code the attacker needs to understand
- we can also use the technique to split basic blocks
  - to hinder comprehension of contained code
  - the attacker will think that there is some decision logic where there is none
- in figure we use a opaque predicate for both purposes
  - the red branch will never be taken



(a) original



(b) with opaque predicate

Opaque Predicates: practical implementations

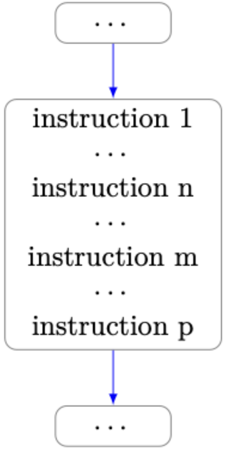
- diablo (binary-to-binary obfuscator)
  - predicates based on mathematical properties of conditional expression
    - e.g. $x^2 \geq 0$ (but they are more complex than this example)
  - fast runtime evaluation (low overhead) yet difficult to prove formally
  - the attacker may study the obfuscator to recognize the hardcoded predicates …
  - … thus predicates instruction generation is randomized
    - use of dead registers
    - constant randomization ($x^2 \geq N$ with random $N \geq 0$)

Opaque Predicates: practical implementations

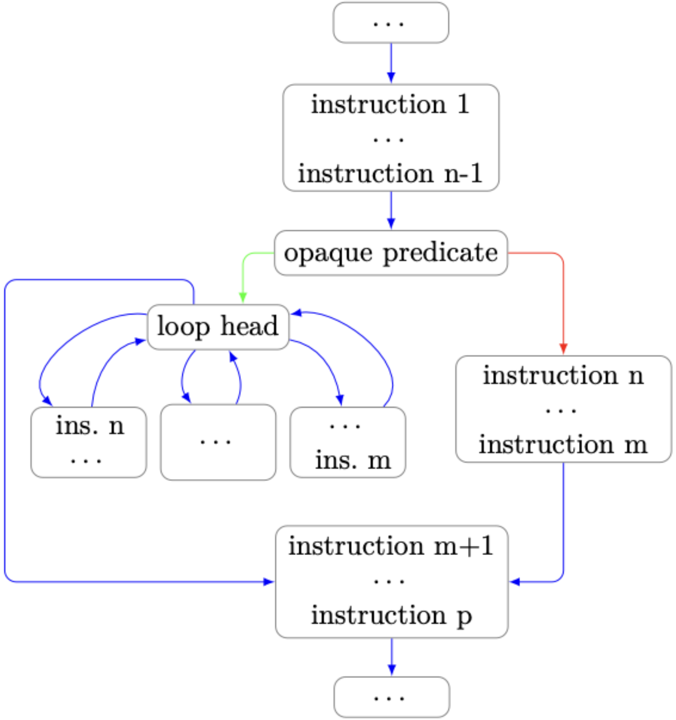
- tigress (source-to-source obfuscator)
  - conditions on pointers to data structures
    - custom data structures added to code for this purpose
  - example: consider two completely separated linked lists L1 and L2
    - we can define three pointers:
      - p1 and p2 point to nodes of L1
      - p3 pointing to an element of L2
    - we can add to program code instructions that move pointers to other nodes, but of the same list
    - we can define two possible kinds of opaque predicates
      - p1 != p3 and p2 != p3 → always true, we can use it for fake code insertion
      - p1 == p2 → true or false, depending on initial nodes pointed and consequent movement instructions
  - user should define the function initializing the data structures
    - should be executed before functions containing opaque predicates (e.g. main)
  - user should define the functions updating pointers/data structure
    - more updates, more complexity (good!) but more overhead at runtime (bad…)

Opaque Predicates: code duplication

- second kind of opaque predicate can be leveraged to duplicate code
- we take the original basic block and we flatten it
- we use the opaque predicate to insert both the original and the flattened version
- with this kind of opaque predicate we don't know which branch will be taken at runtime
  - and that's fine: same code logic
- the attacker needs to analyze the opaque predicate logic and the duplicated code



(a) original



(b) with opaque predicate and flattened control flow

Virtualization obfuscation

- transforms the code to protect so that real opcodes are hidden ◦ translates instructions in a specially devised instruction set
  - uses different opcodes, e.g., randomly selected
  - similar to executing code in a virtual machine
  - turns a function into an interpreter, whose bytecode language is specialized for this function
  - induces as much diversity as possible
  - each interpreter variant differs in the structure of its code as well as in its execution pattern
- at run-time, the code execution is delegated to the interpreter
  - translates each instruction that must be executed from the “virtual instruction set” to the original one
  - to be executed by the actual CPU

Data obfuscation

- this obfuscation works on data, objective:
  - prevent understanding of the value of the constants present in source code
    - during static code analysis
  - prevent understanding of the value of variables during the execution
    - during dynamic analysis
- constants: ad hoc techniques depending on data types
  - integers vs. strings
    - e.g., uses systems of equations for integers
    - automata to generate the strings
- variables: change the representation in memory
  - use ad hoc encoding mathematical function
  - e.g. based on mixed Boolean-arithmetic transforms, modular arithmetic
  - if you want more info/examples on mathematical transforms
    - look at papers cited on Tigress documentation page
      - https://tigress.wtf/encodeData.html

```
x=6;
y=7;
z=x*y;
print z;
```

⇨

```
x=Ek(6);
y=Ek(7);
z=x*y;
print Dk(z);
```

$$x+y=\begin{cases} x-\neg y-1 \\ (x\oplus y)+2\cdot(x\wedge y) \\ (x\vee y)+(x\wedge y) \\ 2\cdot(x\vee y)-(x\oplus y) \end{cases}$$

Literals obfuscation: Tigress implementation

- integer constants obfuscation
  - based on opaque predicates
  - e.g. how to obfuscate constant with value 0 → p1 == p2
    - op. pred. with pointers on linked lists that is always false
    - Boolean false in C is treated as a 0
- string literals obfuscation
  - transformed into calls to an encoder function
  - the encoder function will generate literals at run-time
  - blocks attacker search for literals (e.g. Linux command strings)
    - finding strings first step in attacker (e.g. finding “Wrong password!” in previous example)
  - encoder function logic very easy to understand for attacker
    - also easy to find
      - e.g. will see a call to this function every time a string is printed to console
    - we can protect the function with code obfuscation

Other techniques

- protections that prevent the use of specific tools (but are not considered a form of obfuscation)
  - e.g. anti-debugging protections
- anti-tampering
  - local checks: code guards
  - remote techniques: software and remote attestation
    - use remote server to perform verifications of integrity data produced at the client
- technique that limit the code available at the client
  - no static analysis without the full code
  - no stand-alone dynamic analysis
    - (diversified) pieces of code sent to the client only after the program starts
      - code mobility
    - some functions only executed on the server
      - client-server code splitting

Anti-debugging

- debuggers among most common tools used by attackers, useful for:
  - reverse engineering
    - dynamic inspection of application behavior
    - collection of execution traces for further analysis
  - code tampering
    1. halt application execution
    2. modify memory locations (code or data sections)
    3. resume application with altered logic
- anti-debugging: prevent a debugger from being attached to protected application
  - all techniques based on same assumption:
    - cannot attach more than one debugger at the same time
- basic (and practically useless) implementation
  - a debugger must call ptrace syscall to attach to target process
    - ptrace(PTRACE_ATTACH,pid,0,0)
  - program can ask to not be debugged by calling prctl syscall with following arguments
    - prctl(PR_SET_DUMPABLE ,SUID_DUMP_DISABLE ,0,0,0)
    - this resets a flag in /proc/sys/fs/suid_dumpable
  - easy circumvention by attacker: set flag again after program calls prctl

Anti-debugging: self-debugging

- application includes a self-debugger
    - the self-debugger attaches to main process immediately after starting execution
- attacker must remove the self-debugger to attach his own debugger
- part of instruction logic moved to self-debugger
    - application stop behaving correctly if self-debugger is simply removed
- simple schema (used to protect Starcraft II)
    - some jump instructions substituted with debug exceptions
        - exception includes the original program address launching the exception
    - control passes to self-debugger
        - static mapping: debug exception address → original jump instruction target address
    - easily circumvented: static analysis to locate debug exception handlers and reconstruct original jump instructions

Anti-debugging: Diablo self-debugging

- complex schema
  - patented but free for non-commercial use
- whole parts of protected application code can be moved to the self-debugger
- when execution arrives to moved code → debug exception launched
- self-debugger copies target process context (e.g. CPU register values)
- self-debugger executes instructions
  - can read/write protected process memory with syscall ptrace
    - called with constants PTRACE_PEEKDATA and PTRACE_POKEDATA
- removing Diablo self-debugger is not trivial
  - attacker should restore moved instructions in their original position in binary
  - problem: moved instructions do not use the original CPU registers
  - problem: memory read/write instructions substituted with self-debugger routines
    - using internal ptrace syscalls with PTRACE_PEEKDATA and PTRACE_POKEDATA constants

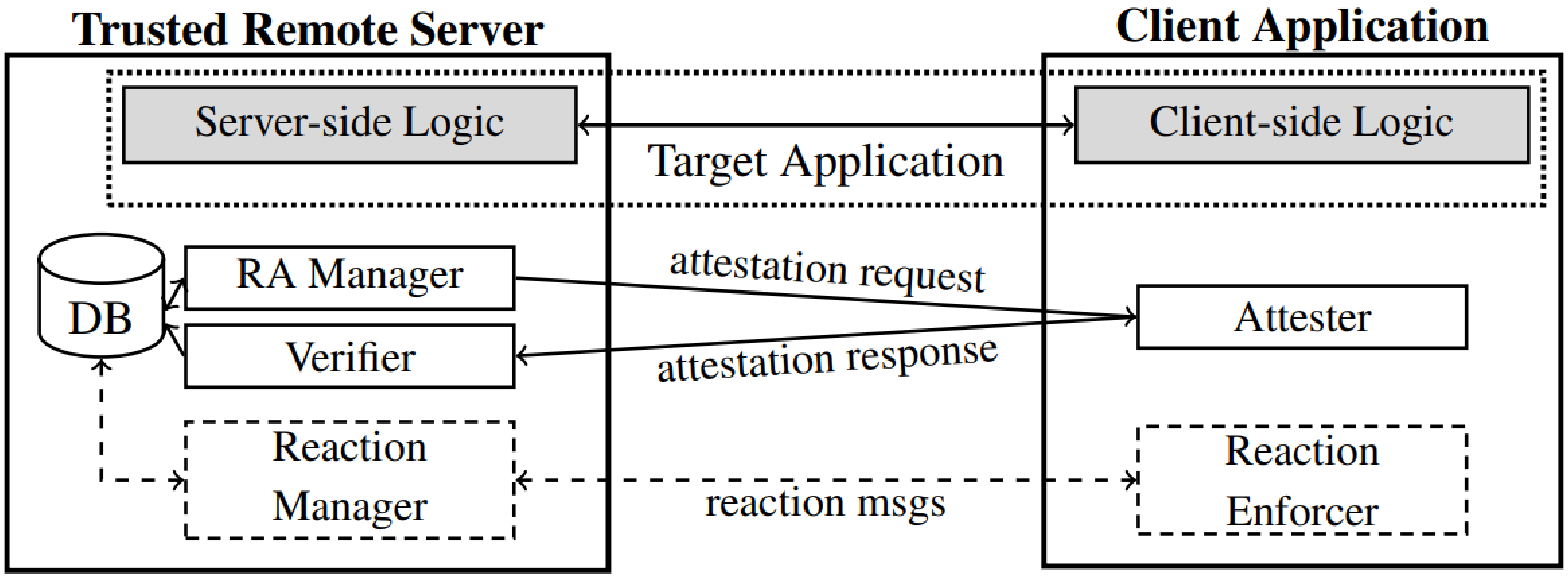
Anti-tampering

- category of protections that aim at making code changes more complex
  - changes should come with a cost
- security property
  - integrity (e.g., of the code)
  - execution correctness: much more complex to obtain
- different families of protections
  - local vs. remote
    - local if all the components are in the program
    - remote if they resort to external components (e.g., servers as the root of trust)
  - with or without secure hardware/secure coprocessors
    - when available, some computations can be offloaded to pieces of HW that cannot be tampered without local intervention

Software attestation

- verify that a program running on another system is behaving as expected
  - secure HW is not used or not available
    - better for portable devices, IoT, embedded systems, etc.
- usually implemented as “application integrity”
  - hypothesis: if binaries are correct, then also the application will behave correctly
    - checksum of the binary or configuration files stored in the file system
    - checksum of the binary loaded in memory
    - checked at load- or run-time
  - vulnerable to several attacks
    - dynamic code injection (i.e., with debuggers)
    - parallel execution of an untampered version of the device

Software attestation: architecture



Remote attestation

- methodology used to verify that a program running on another system is behaving as expected
  - …but with secure HW
    - TPM or other secure coprocessors
    - Intel SGX or ARM TrustZone used to have a root of trust
  - the most widespread approach defined by the Trusted Computing Group
    - TPM + well defined components + architecture + protocols
    - https://trustedcomputinggroup.org/wp-content/uploads/TCG_1_4_Architecture_Overview.pdf
    - workflow
      - attest the BIOS, then the loader, then the OS, then attest all the security sensitive applications, …
  - current RA methodologies limit functionality...
    - does not scale well for virtualization (e.g., for software networks)
      - new results available that seem to impact this field
    - in the best case, usability is very affected

Code guards

- pieces of code injected into the application
  - check other pieces of code of the same program for specific code properties
  - if checks are OK the program is assumed to be uncompromised
  - examples of checks
    - hash of bytes in memory (code or data)
    - hash of the executed instructions for unconditional code blocks
    - crypto guards: the next block is correctly decrypted if the previously executed blocks are the correct ones
- reactions prevent the correct execution of the rest of the application
  - graceful degradation
  - faults / crashes
  - reactions must be delayed to avoid the attacker to defeat them
    - or you have to resort to remote servers

Code mobility

- an online anti-RE+anti-tampering technique where the program is shipped without pieces of code
  - a local binder understands when a piece of code needs to be executed
  - a downloader obtains it from a trusted server
  - the downloaded code blocks become part of the application
  - they may be discarded when the application is stopped
- mobile blocks are usually security-sensitive pieces of code
  - may be protected
  - can be replaced
    - diversification techniques used to obtain sets of blocks with the same semantics
    - not always the same "cuts"
      - e.g., barrier slicing

Client-Server Code Splitting

- splits code of the application to protect
  - part of the code is executed on a trusted server
    - the sensitive ones
  - client and server code is interleaved
    - the server performs a sort of remote computation for each application
- proved with empirical experiments that it is better to split several small pieces
  - instead of big blocks with all the sensitive parts
  - more confusion and more links to follow
- problem: device should always be online to execute application
  - not a very big problem nowadays
  - can limit deployment to most sensitive algorithms
- problem: server overhead
  - delay introduced if high-load on server
  - potential risk of DoS

Techniques for diversification

- generates different semantically equivalent copies of code blocks
  - up to functions and entire programs
- avoid that exploits extend to large number of copies of the same software
  - risk mitigation
  - only a limited set of program copies are affected by a given exploit
- obtained in different manners
  - different compilation options
  - generators of diversity
  - obfuscating the code to diversify with different techniques
    - also using different parameters

PhD/MSc Seminar on Software Security and Protection

- 2 weeks - 12 hours/week (24 hours)
  - 2 ECTS for Master students
- To be held next July – announced on UniCa website
- First week: Secure programming
  - Secure programming: principles and guidelines
  - Security evaluation of software
  - Lab: software testing with automatic tools
- Second week: Software protection
  - Techniques for protection of software and Intellectual Property
  - Static analysis of software
  - Lab: software obfuscation
- Exam: group project
  - group of 3-4 students
  - find a FOSS application, test for vulnerabilities, find assets and protect them
  - prepare and submit a detailed report of your activities
- More details here