Software based approaches

Mariano Ceccato

Content

- Continuous replacement
  Christian Collberg

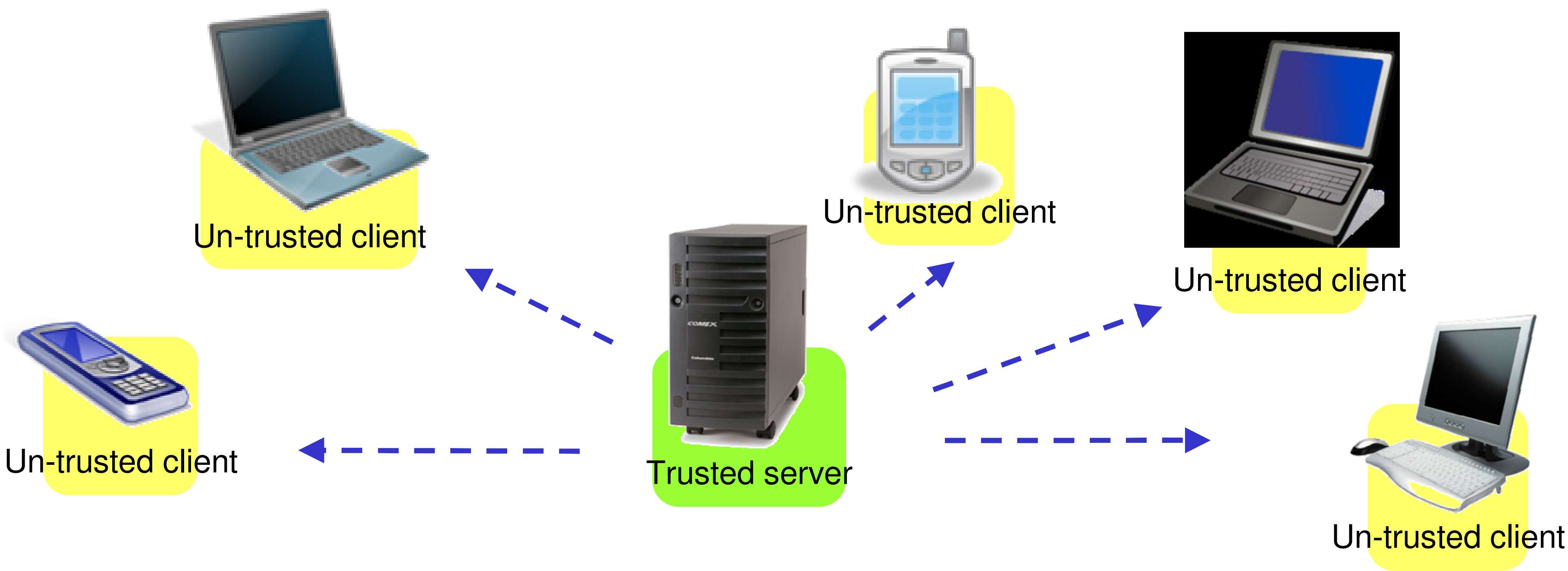
- White box remote procedure execution
  Amir Herzberg, Amitabh Saxena, Haya Shulman, Bruno Crispo

- Orthogonal replacement
  Ceccato Mariano, Mila Dalla Preda, Anirban Majumbar, Paolo Tonella

- Empirical evaluation of reverse engineering complexity
  Mariano Ceccato, Massimiliano Di Penta, Jasvir Nagra, Paolo Falcarin, Filippo Ricca, Marco Torchiano, Paolo Tonella

Remote software trusting

- *Remote software authentication*: ensuring a (server) that an un trusted host (client) is running a “healthy” version of a program (code integrity)
- Before delivering any service the server wants to know that the client is executing according to its expectations



Attack model

Attacker on un-trusted host:

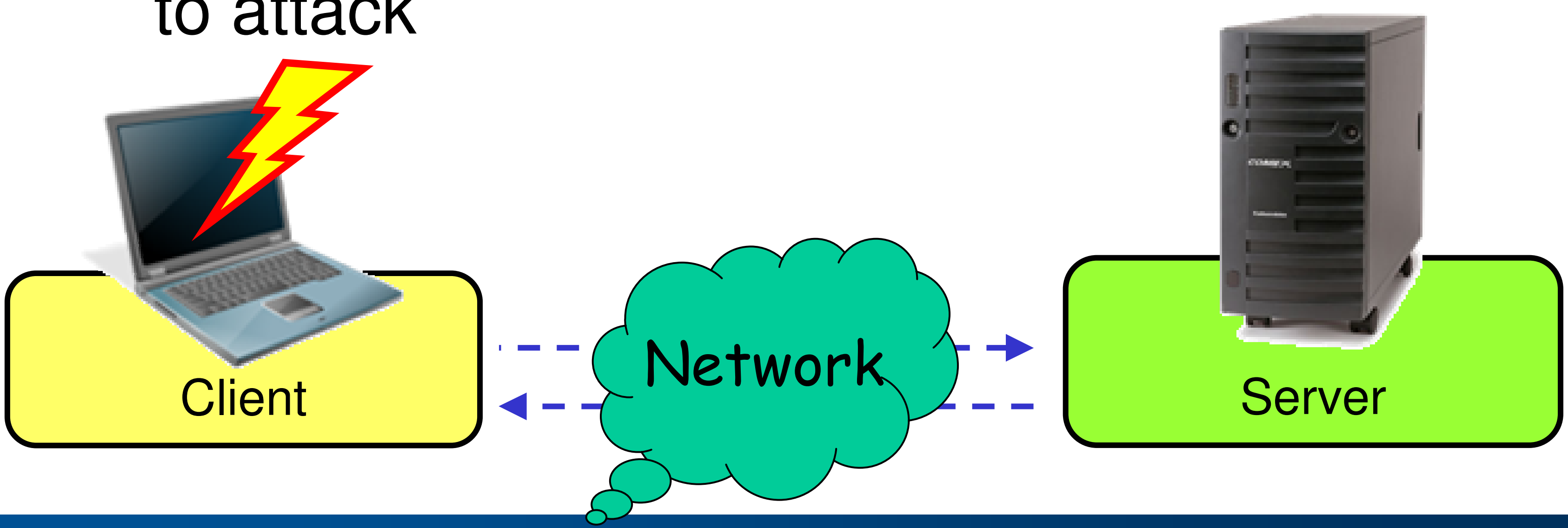
- Any dynamic/static analysis tool
- Any software (buggers, emulators, …)
- Read/write any memory location, register, network message, file.

Attacks:

- Reverse engineer and direct code change.
- Runtime modification of the memory.
- Produce (possibly tampered) copies of P that run in parallel.
- Interception and change of network messages.

Attacker goal

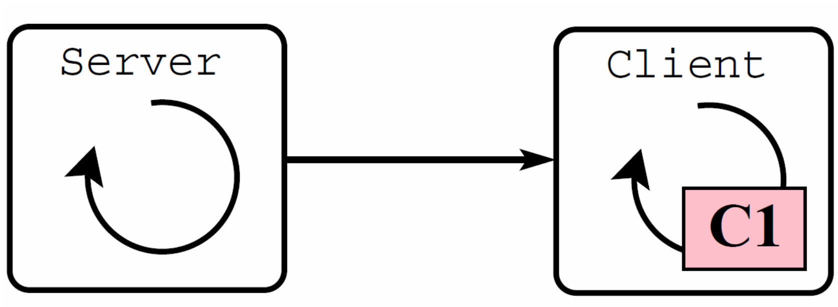
- **Goal:** to tamper with the application code without being detected by the server
  - Substantial program understanding effort by a human to understand the inner logic to attack



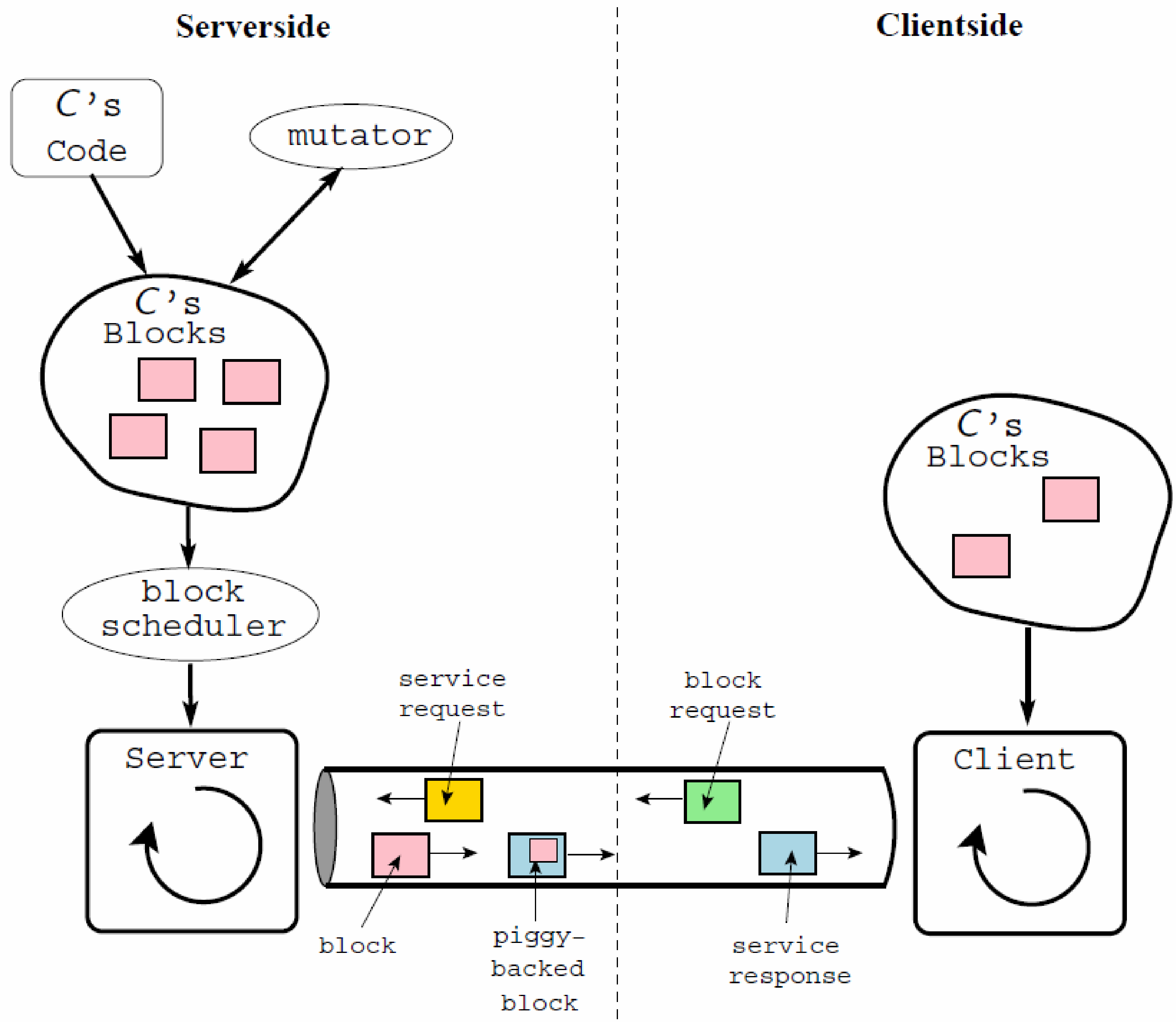
Remote entrusting by continuous replacement

- Basic idea: The server generates a continuous sequence of

Obfuscated blocks:



- If replacement is quick enough, and versions different enough, the attacker won't have time to analyse the code.
- C and a Java implementations are underway.

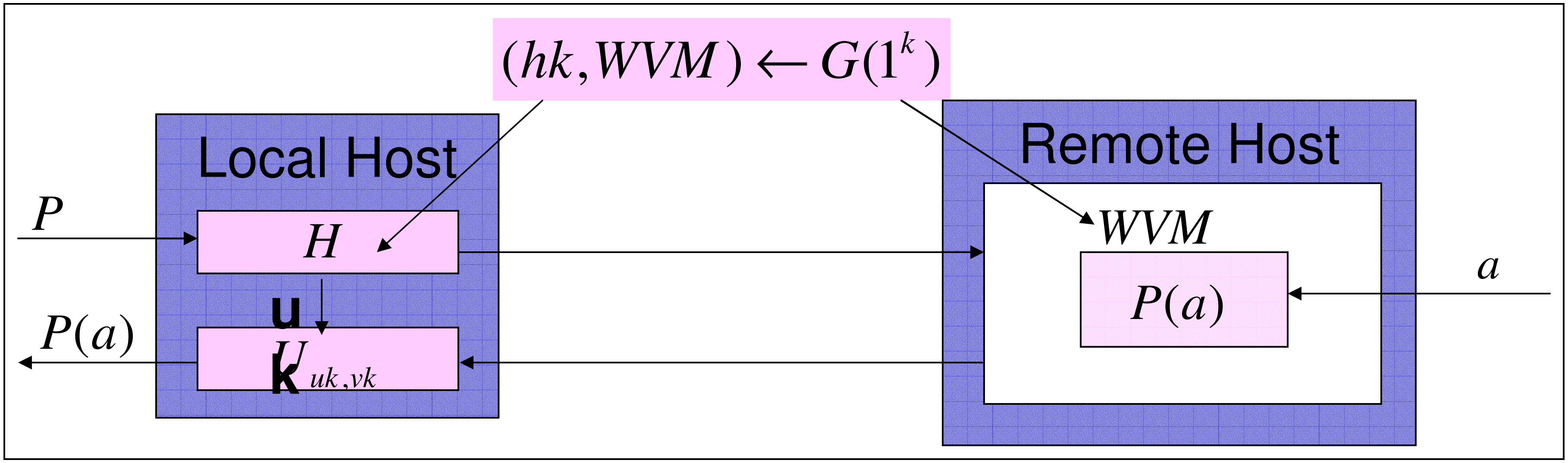

Serverside
Clientside
C's Code
mutator
C's Blocks
block scheduler
Server
service request
block
piggy-backed block
block request
service response
Client
C's Blocks

Remote entrusting by continuous replacement

- The level of tamperproofing you achieve through this setup is determined by
  1) the fraction of the total number of blocks that the server shares with the client,
  2) the rate by which the server generates mutated blocks and pushes them onto the client, and
  3) the rate by which the adversary can analyze the continuously changing program in the client's bag-of-blocks.

WBRPE:
White Box Remote Procedure Execution

- Goal: secure execution of programs in remote hostile environment
- New white-box security primitive:
  - The *WBRPE* is a tuple of *PPT* algorithms *<G,H,U>*

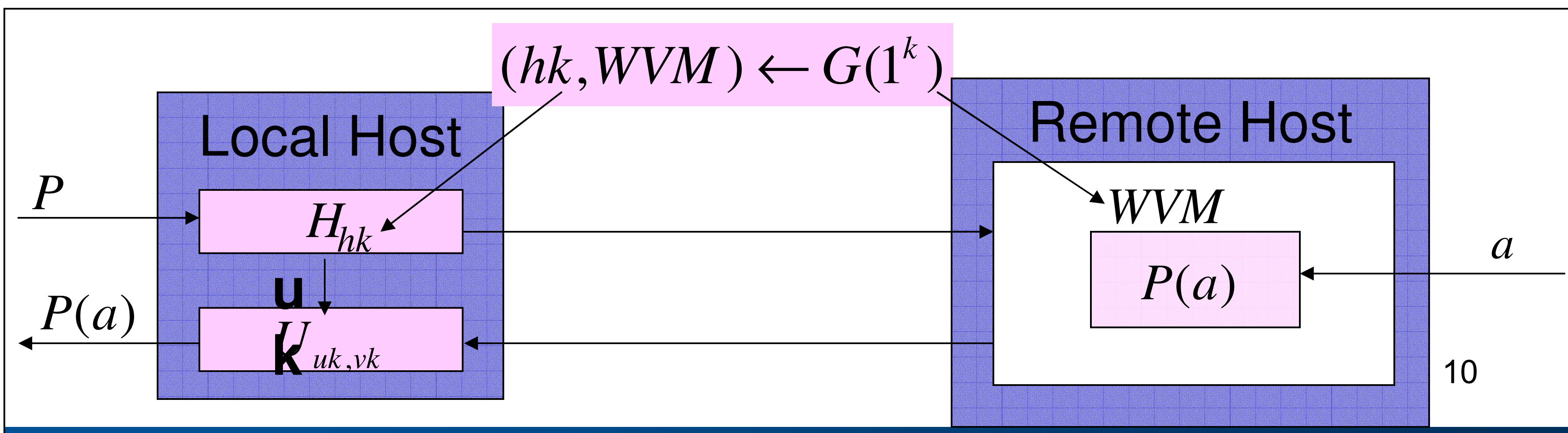


Remote Program Execution

- Applications: PIR, DRM, grid computing, mobile agents

- Threats
  - To local host
    - Exposure of program or of data embedded in program
    - Receive incorrect output
  - To remote host
    - Exposure of remote input *a*

- Ensure
  - To local host
    - confidentiality of inputs
    - Integrity of output
  - To remote host
    - Privacy of remote input *a*
    - By validating input programs

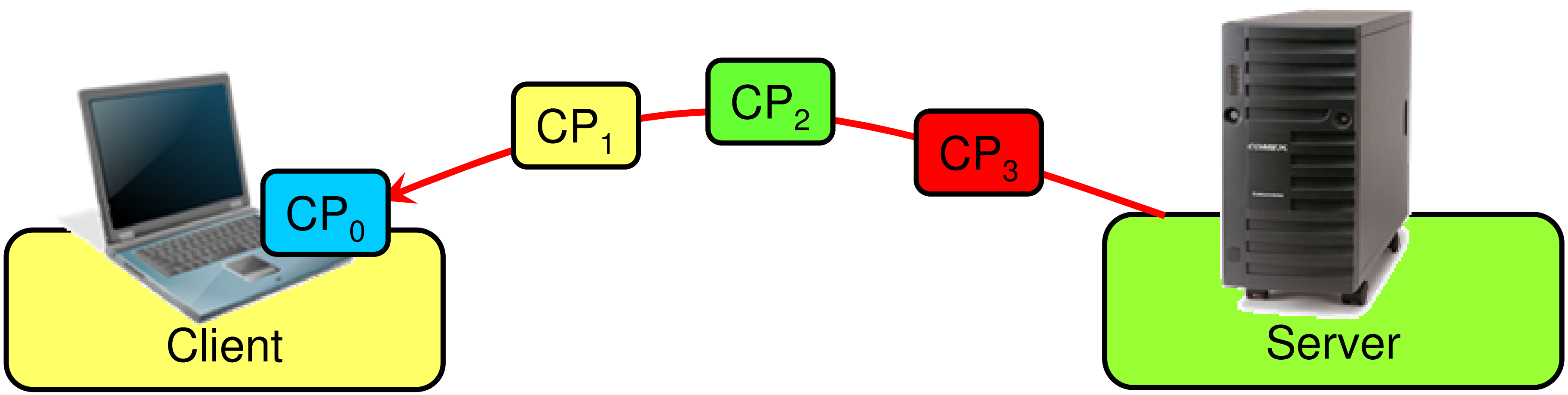


Remote Program Execution: Results

- New white-box security primitive, the *WBRPE*
  - Basic building block
  - Definitions & Security Specifications for remote programs execution
- Universal *WBRPE*
  - *WBRPE* for a specific program ➔ *WBRPE* for any program
- Provably secure theoretical feasibility result for *WBRPE*
  - Secure function evaluation implemented via garbled circuits
- Robust *WBRPE* combiner:
  - Combined *WBRPE* scheme *W''*○*W'* is secure, if either *W''* <u>or</u> *W'* is secure

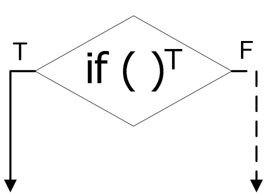
Orthogonal replacement

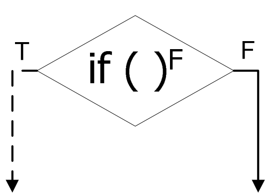
- Periodically replace the client code with a new version
  - Orthogonal (obfuscated)
  - Semantically <u>different</u>



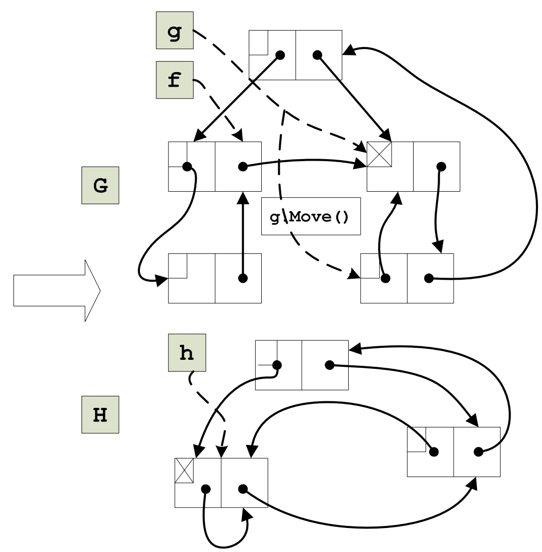
Obfuscation

- Transforming a program CP into an equivalent one CP' that is harder to reverse engineer, while maintaining its semantics.
- Opaque predicate:
  - conditional expression whose value is known to the obfuscator, but is difficult for an adversary to deduce statically
  - Precise inter-procedural static analysis is intractable



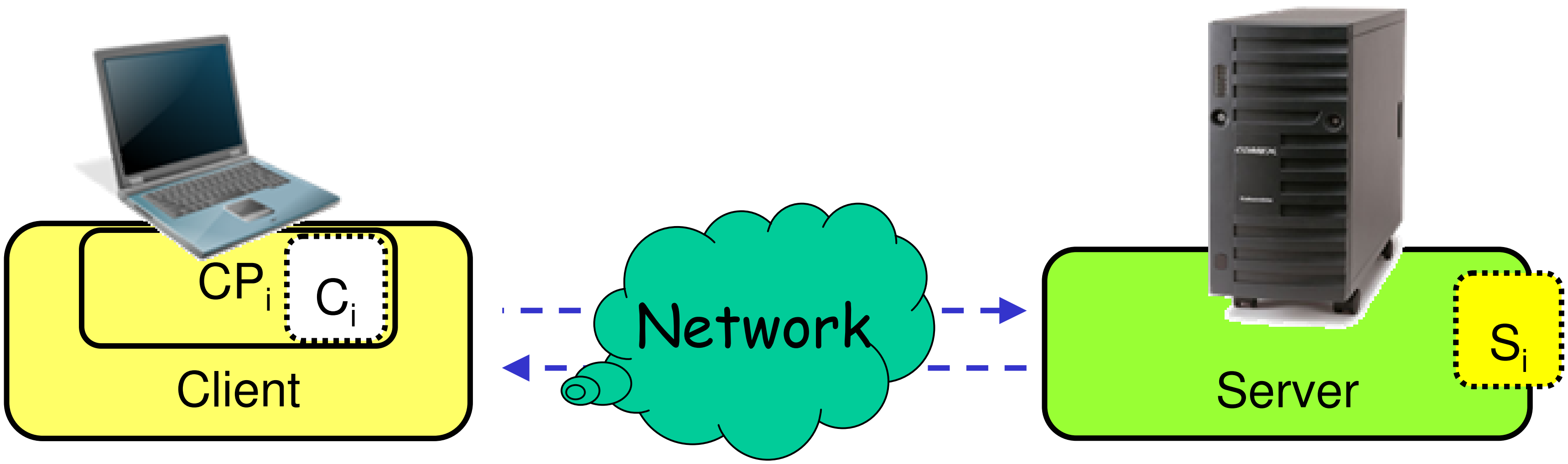



```
Node g, h;
Method P(…,Node f)
{
      g = g.Move();
      h = h.Move();
      h = h.Insert(new Node)
…
      if (f==g)? …
      if (g==h)F …
…
      f.Token = False;
      g.Token = True;
      if (f.Token)? …
…
```

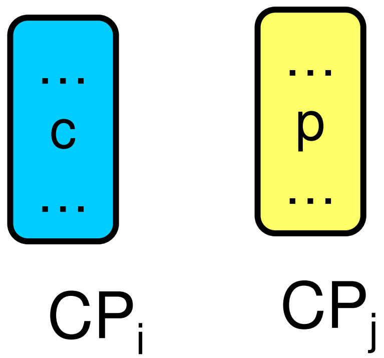


Splitting

- The code of CP_i can be split into (C_i, S_i) where:
  - C_i remains on the client
  - S_i runs on the server
- This process ensures that
  - the code left on the client is orthogonal with respect to the previous clients
  - An expired client can not longer be used (it would not work with the new server)

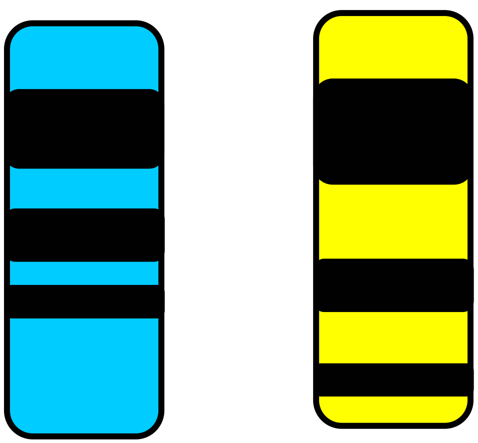


Orthogonality



<u>Statement orthogonality</u>
$c \perp p$ if:
the understanding of the of ***c*** the role in $\mathbf{CP_i}$ does not reveal information about the role of ***p*** in $\mathbf{CP_j}$

<u>Program orthogonality</u>
$\mathbf{CP_i} \perp \mathbf{CP_j}$ if:
they contains only* orthogonal statements



*Not possible to transform or move to the server:
- System calls
- Library calls
- Input output operations

Generation Performance

Application	No. of clients	No. of clones
CarRace	10	1
	50	9
	100	21
	500	160
	1000	347
ChatClient	10	1
	50	7
	100	11
	500	97
	1000	218

- **Application lifetime 5 years**
- **A replacement every 2 days**

Attacks

- Opaque predicates could be attacked through dynamic analysis (debugging)
  - Removing branches that are not executed could cause the elimination of useful code
  - We could add predicates that infrequently evaluate to True (False) and if removed cause the application to malfunction

Research questions

RQ1: To what extent the obfuscation reduces the capability of subjects to comprehend decompiled source code?

RQ2: To what extent the obfuscation increases the time needed to perform a comprehension task?

RQ3: To what extent the obfuscation reduces the capability of subjects to perform a change task?

RQ4: To what extent the obfuscation increases the time needed to perform a change task?

Experimental design

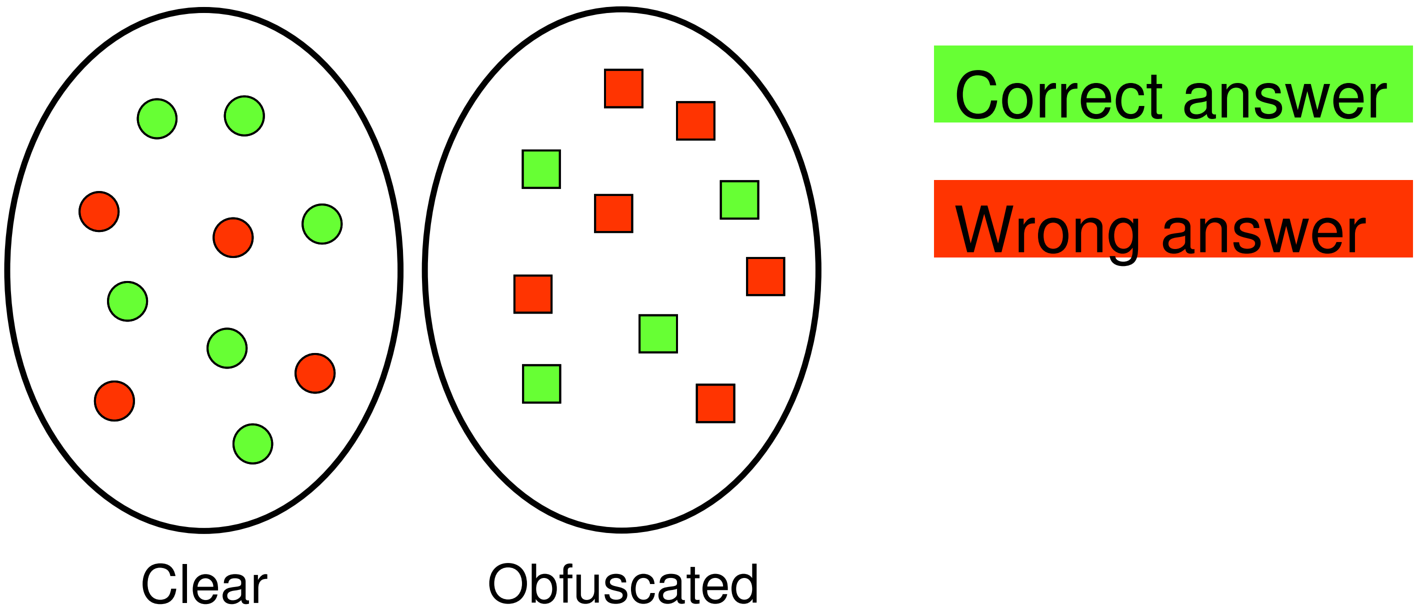
- Decompiled code
- Code browsing tools
- Debuggers
- API documentation
- Possibility to run the (modified) code

1st session	Clear	Obfuscated
App1	**G1**	**G2**
App2	**G4**	**G3**

2nd session	Clear	Obfuscated
App1	**G3**	**G4**
App2	**G2**	**G1**

Subject have been properly trained on:
- code obfuscation
- whole experimental environment

Descriptive statistics



1) Presence of statistical difference
   - correctness reduced
   - time increases

2) Determination of effect size
   - likelihood to change it wrong
   - magnitude of time increasing

Accuracy

	Comprehension		Attack		Overall	
Treat				Correct	Wrong	Correct
Clear				15	10	26
Obfuscated	12	8	12	8	24	16
P-value (Fisher test)	0.33		**0.009**		**0.006**	
Effect size (Odds ratio)	2.3		**7.1**		**3.8**	

P-value < 5% : statistical difference between treatments

Effect > 1 : relevant effect

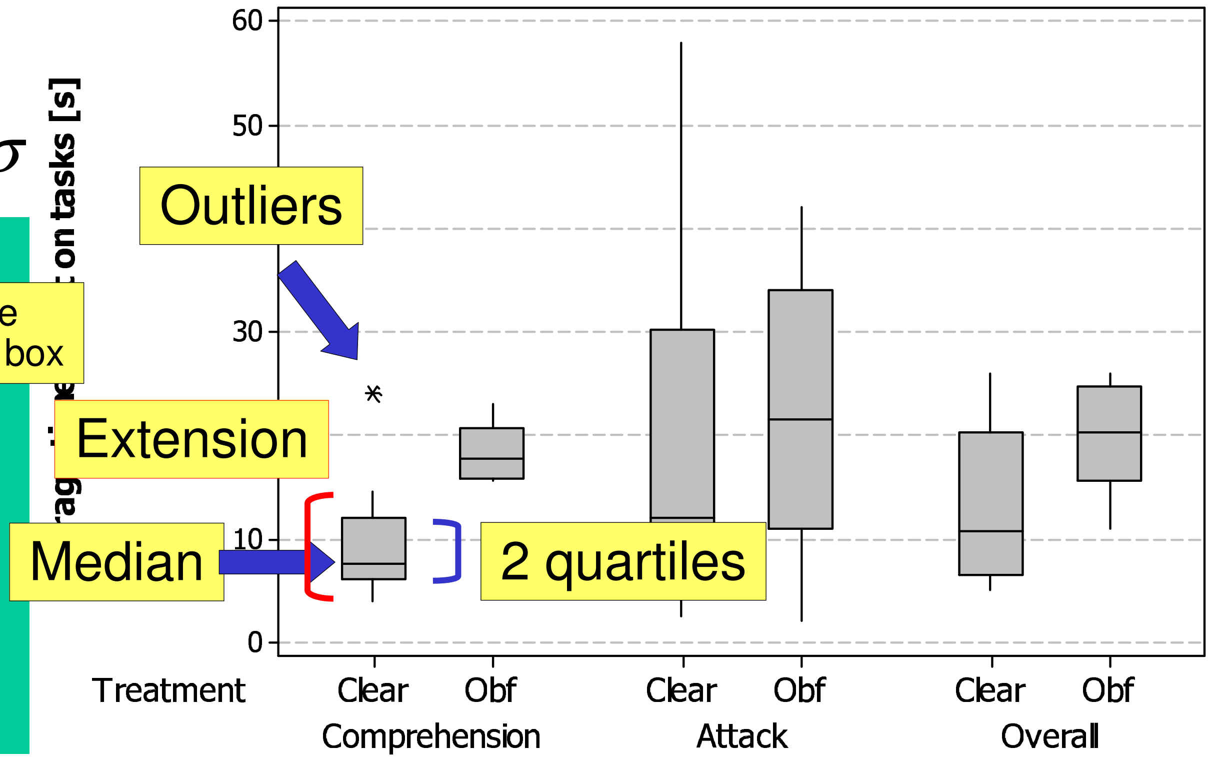
$$OR = \frac{q/(1-p)}{p/(1-q)}$$

An odds indicate how much likely is that an event will occur as opposed to it not occurring.

Time

$$d = (M_1 - M_2)/\sigma$$

The Cohen *d* effect size indicates the magnitude of a main factor treatment effect on the dependent variables



	Comprehension	Attack	Overall
P-value (Mann-Whitney)	**0.002**	0.19	**0.02**
Effect Size (Cohen d)	**1.8**	0.2	**1.03**

Conclusions

- **H01** The obfuscation does not significantly reduce source code comprehensibility.

HA2 The obfuscation significantly increases the time needed to perform code comprehension tasks
Effect size = 1.8

HA3 The obfuscation significantly reduces the capability of subjects to correctly perform a change task.
Odds ratio = 7.1

- **H04** The obfuscation does not significantly increase the time needed to perform a change task.