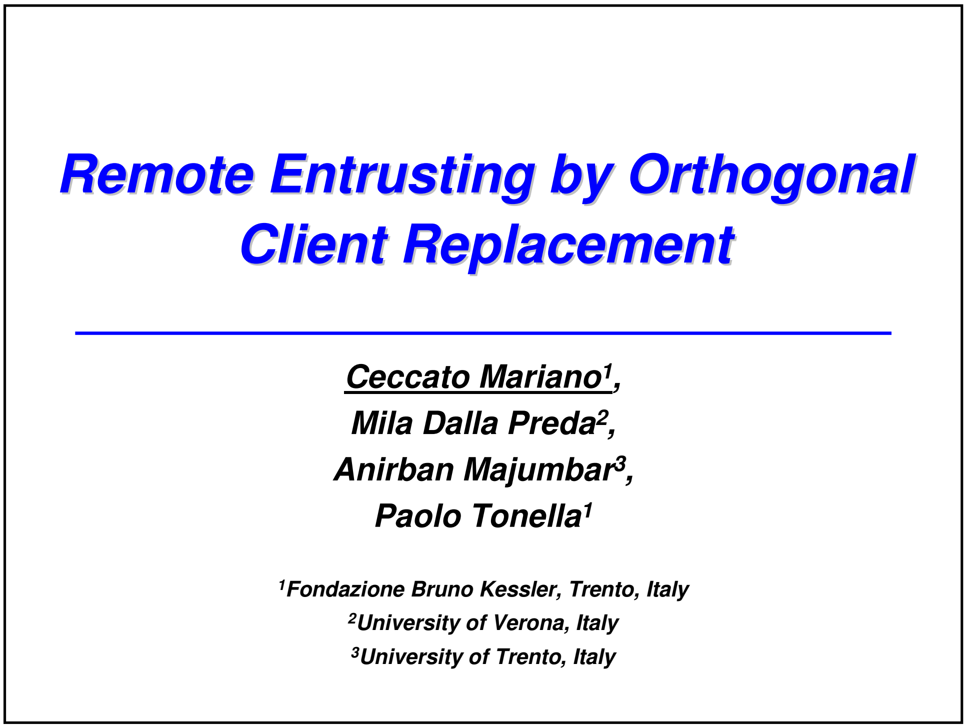


Remote Entrusting by Orthogonal Client Replacement

***Ceccato Mariano*[1],**

***Mila Dalla Preda*[2],**

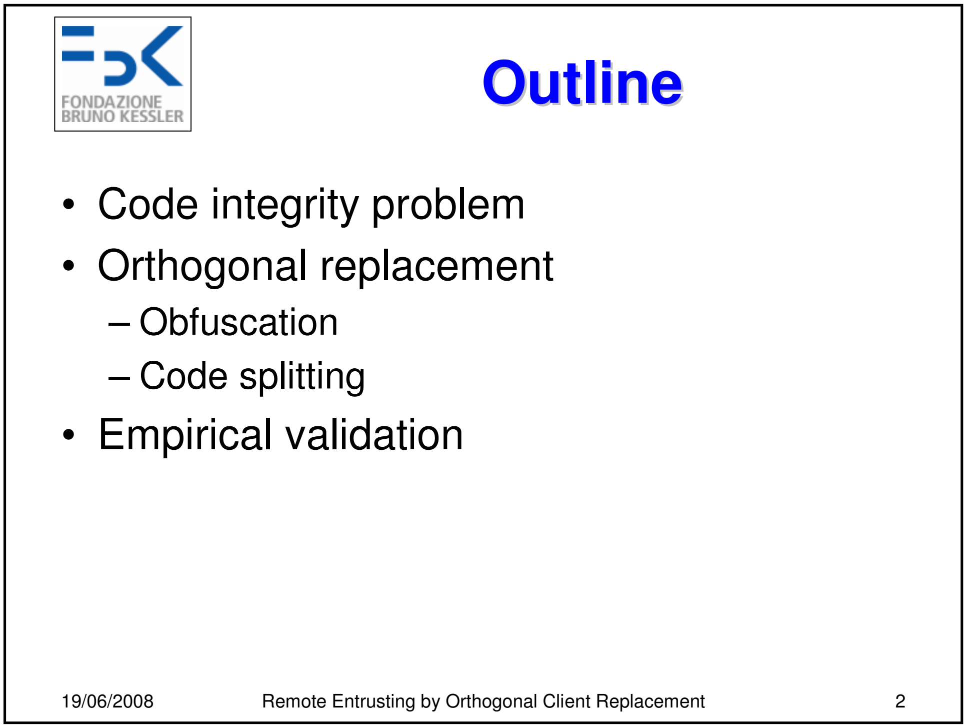
***Anirban Majumbar*[3],**

***Paolo Tonella*[1]**

[1]Fondazione Bruno Kessler, Trento, Italy

[2]University of Verona, Italy

[3]University of Trento, Italy

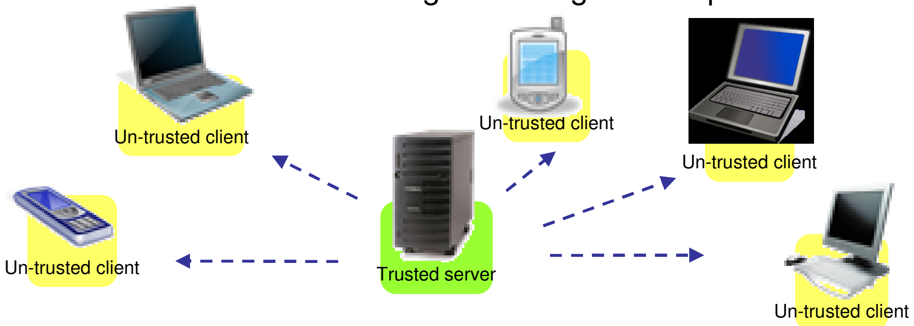


Outline

- Code integrity problem
- Orthogonal replacement
  - Obfuscation
  - Code splitting
- Empirical validation

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:

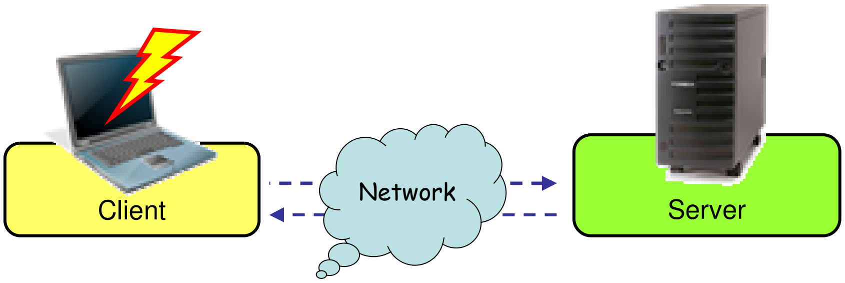
- Any dynamic/static analysis tool
- Any software (buggers, emulators, …)
- Read/write any memory location, register, network message, file.

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.

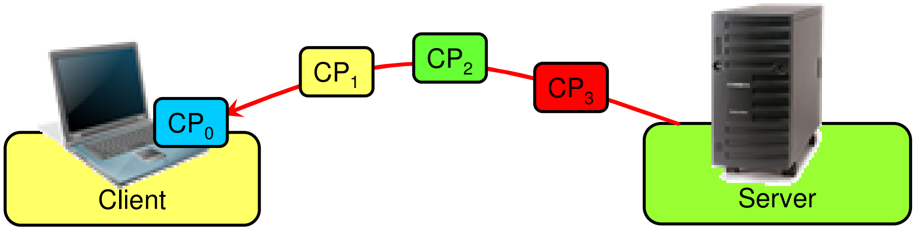
Attacker goal

- **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



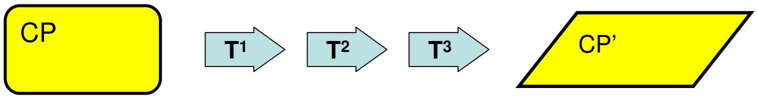
Our approach

- 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.
  - Potency: obscurity added to a program
  - Resilience: how difficult is to automatically de-obfuscate
  - Cost: computation overhead of P'



Obfuscation

```
Student guy = new Student( );
String name = "Mathematics";
Course course = new Course(name);
guy.apply(course);
course.commitChanges( );
```

```
y1 x1 = new y1( );
String x2 = "Mathematics";
y2 x3 = new y2(x2);
x1.z1(x3);
x3.z2( );
```

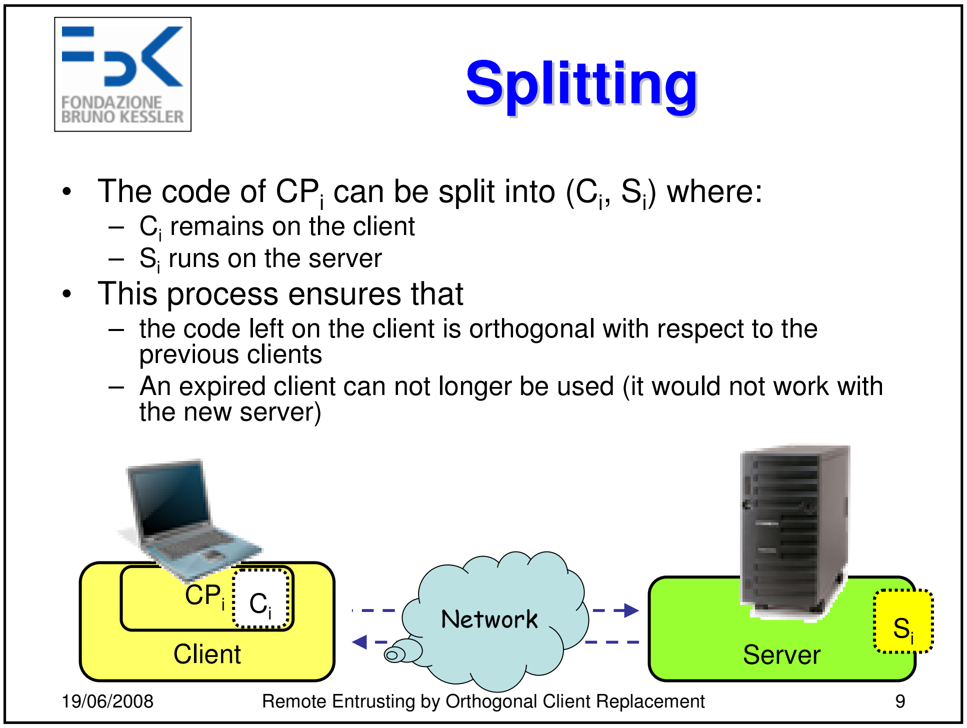
1. Layout obfuscation

```
Object[ ] data = new Object[100];
data[1] = new Student( );
data[2] = "Mathematics";
data[3] = new Course(data[2]);
guy.apply(data[3]);
data[3].commitChanges( );
```

2. Data obfuscation

```
while (s < 10) {
   Student guy = new Student( );
   if (a = 2)
      String name = "Mathematics";
   else
      String name = "Fisics";
   Course course = new Course(name);
   if (z > x)
      guy.apply(course);
   else
      gut.retire( );
   course.commitChanges( );
   s ++;
}
```

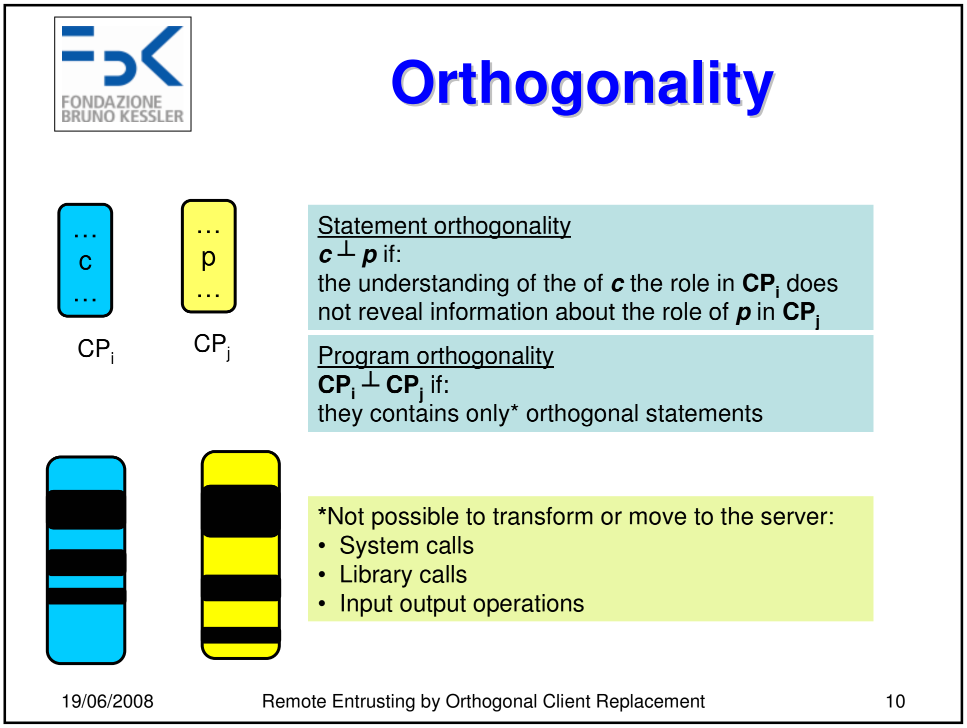
3. Control flow obfuscation



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)

CP_i
C_i
Client
Network
Server
S_i

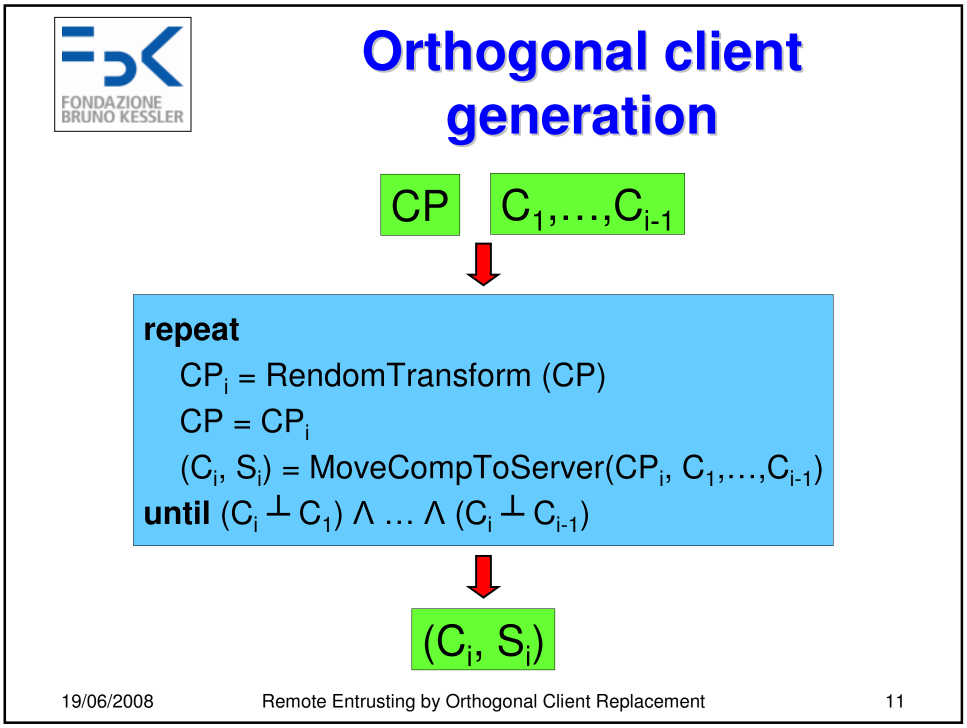


Orthogonality

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

Program orthogonality
$CP_i \perp CP_j$ if:
they contains only* orthogonal statements

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

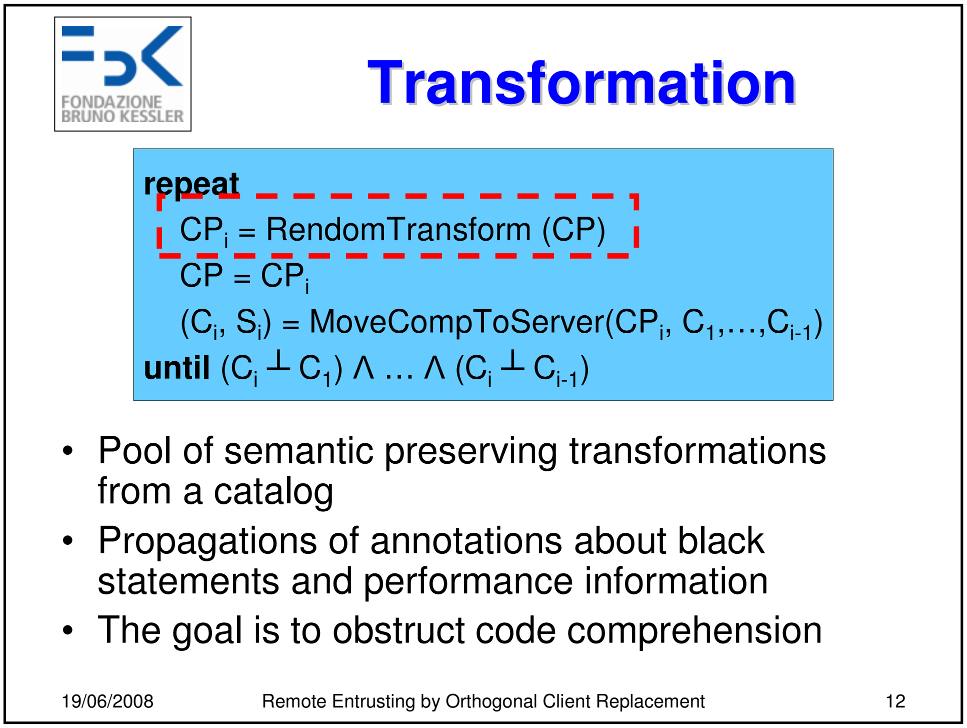


Orthogonal client generation

CP $C_1,\ldots,C_{i-1}$

```
repeat
    $CP_i$ = RendomTransform (CP)
    $CP = CP_i$
    $(C_i, S_i)$ = MoveCompToServer($CP_i, C_1,\ldots,C_{i-1}$)
until $(C_i \perp C_1) \wedge \ldots \wedge (C_i \perp C_{i-1})$
```

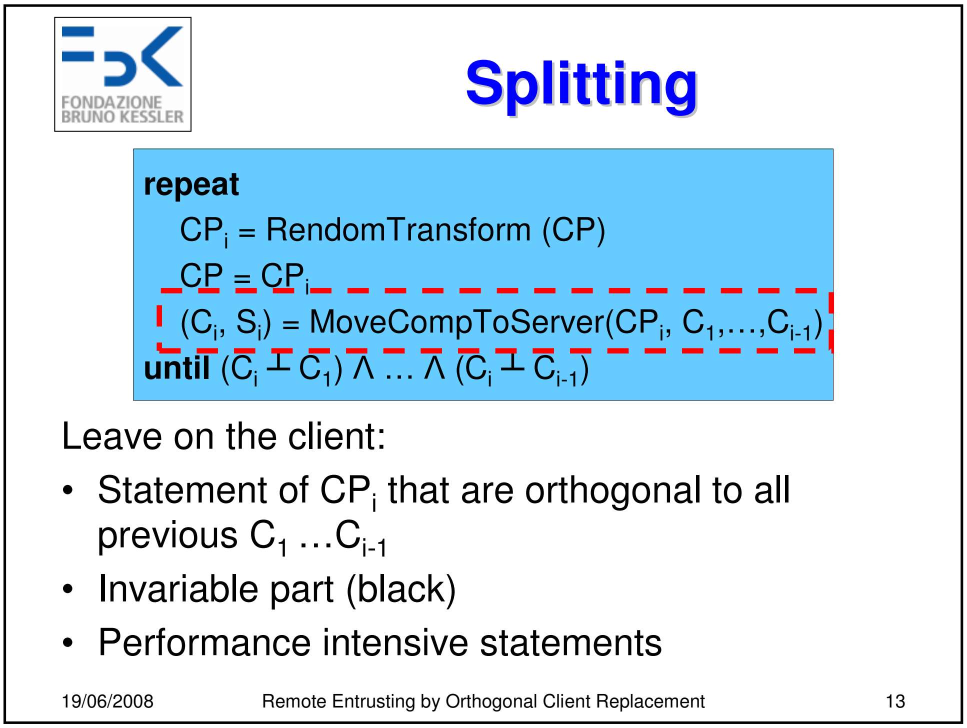
(C_i, S_i)



Transformation

```
repeat
    $CP_i$ = RendomTransform (CP)
    $CP = CP_i$
    $(C_i, S_i)$ = MoveCompToServer($CP_i, C_1,\ldots,C_{i-1}$)
until $(C_i \perp C_1) \wedge \ldots \wedge (C_i \perp C_{i-1})$
```

- Pool of semantic preserving transformations from a catalog
- Propagations of annotations about black statements and performance information
- The goal is to obstruct code comprehension

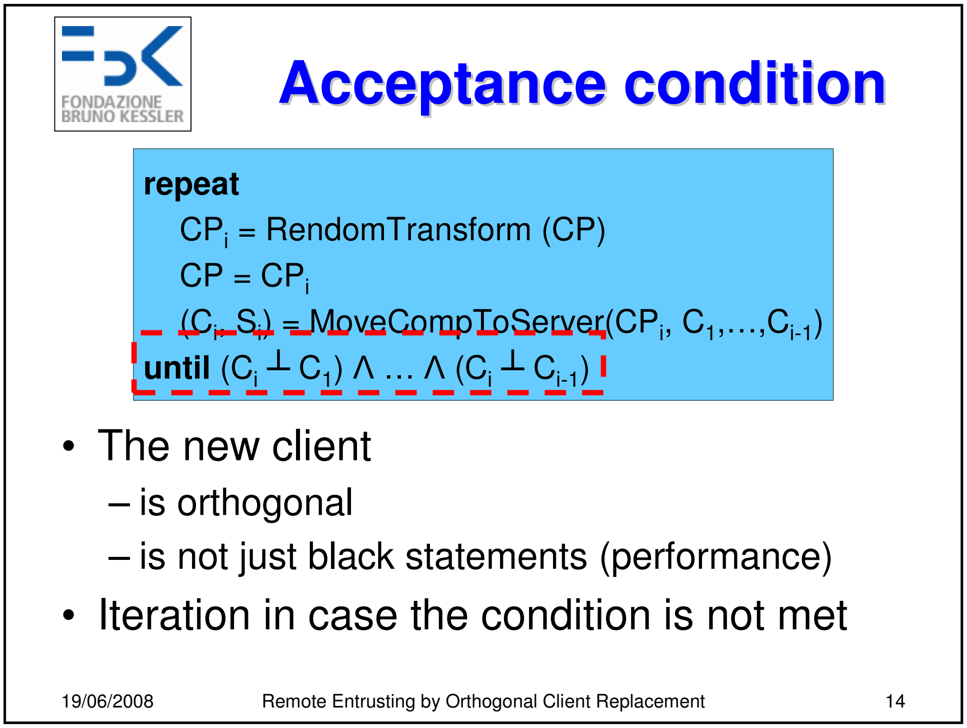


Splitting

```
repeat
    $CP_i$ = RendomTransform (CP)
    CP = $CP_i$
    $(C_i, S_i)$ = MoveCompToServer($CP_i$, $C_1,\ldots,C_{i-1}$)
until $(C_i \perp C_1) \wedge \ldots \wedge (C_i \perp C_{i-1})$
```

Leave on the client:

- Statement of CP_i that are orthogonal to all previous $C_1 \ldots C_{i-1}$
- Invariable part (black)
- Performance intensive statements



Acceptance condition

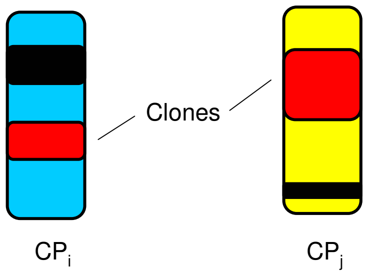
```
repeat
    $CP_i$ = RendomTransform (CP)
    CP = $CP_i$
    $(C_i, S_i)$ = MoveCompToServer($CP_i$, $C_1,\ldots,C_{i-1}$)
until $(C_i \perp C_1) \wedge \ldots \wedge (C_i \perp C_{i-1})$
```

- The new client
  - is orthogonal
  - is not just black statements (performance)
- Iteration in case the condition is not met

Empirical validation

Clone based orthogonality

- Orthogonality from a program comprehension point of view is hard to define and quantify
- Practical and computable approximation of orthogonality: based on clones



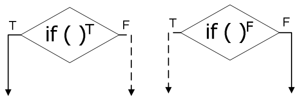
Statement orthogonality

$\boldsymbol{c} \perp \boldsymbol{p}$ if:

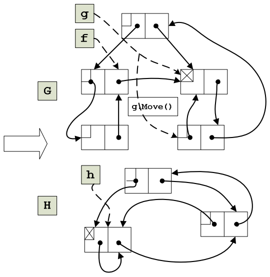
the understanding of the of ***c*** the role in **CP_i** does not reveal information about the role of ***p*** in **CP_j**

Alias based opaque predicates

- 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)? …
…
```



Alias based opaque predicates

Aliases :
f = = g
g ! = h
Update :
updateAlias ( )

```
class A {
int f1 ;
int f2 ;
void m ( ) {
  f1 = 1 ;
  f2 = f1 ++;
  int tmp = f1 ;
  tmp = tmp - f1 ;
  f1 = f1 + f2 ;
  }
}
```

```
class A {
int f1 ;
int f2 ;
void m ( ) {
   int tmp ;
   if ( f ==g ) {
        f1 = 1 ;
        updateAlias( ) ;
        f2 = f1 ++;
   }
   else {
        updateAlias( ) ;
        tmp = f1 +f2 / 5 ;
        f1 = f2 - tmp ;
   }
```

```
if ( g != h ) {
        updateAlias( ) ;
        tmp = f1 ;
        tmp = tmp - f1 ;
        updateAlias( ) ;
        f1 = f1 +f2 ;
    }
    else {
        f1 = tmp / f2 ;
        tmp = f2%59+f2 ;
        updateAlias( ) ;
        }
    }
}
```

Case studies

- CarRace (on-line game)
  - CP_{race} = 220 loc
- Chat application
  - CP_{chat} = 110 loc
- On line applications
- Written in java (~1K loc each)
- Source code is sensitive to malicious modifications

Clone size threshold

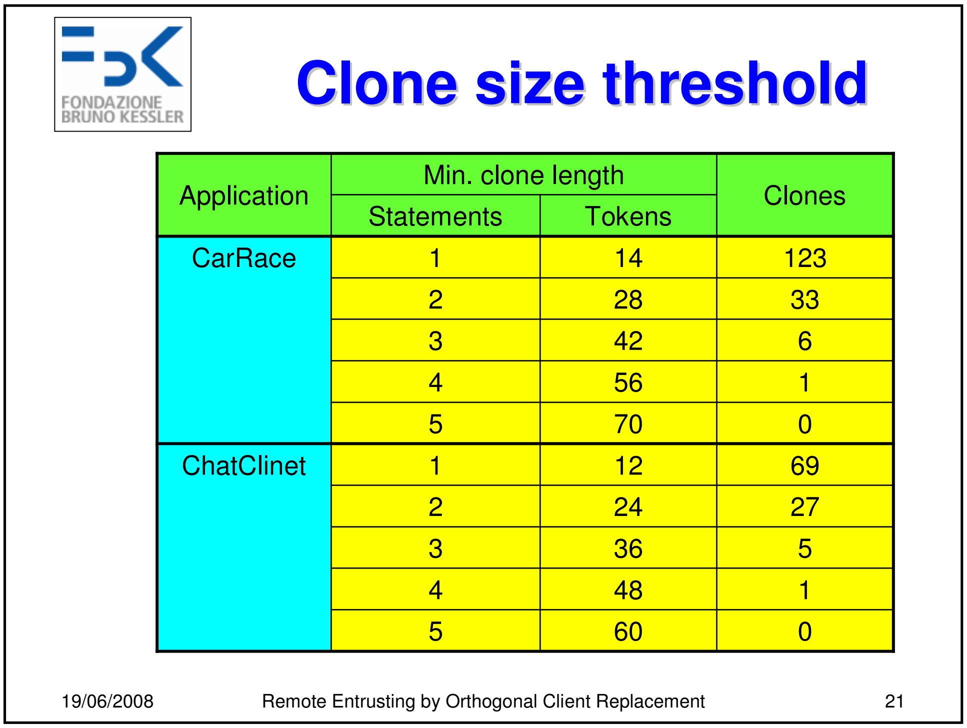
Small threshold

- Too many iterations of the algorithm
  - exponential grown of the source code
- Most of the reposted clones are false positives
- Improvements do not bring more security

Large threshold

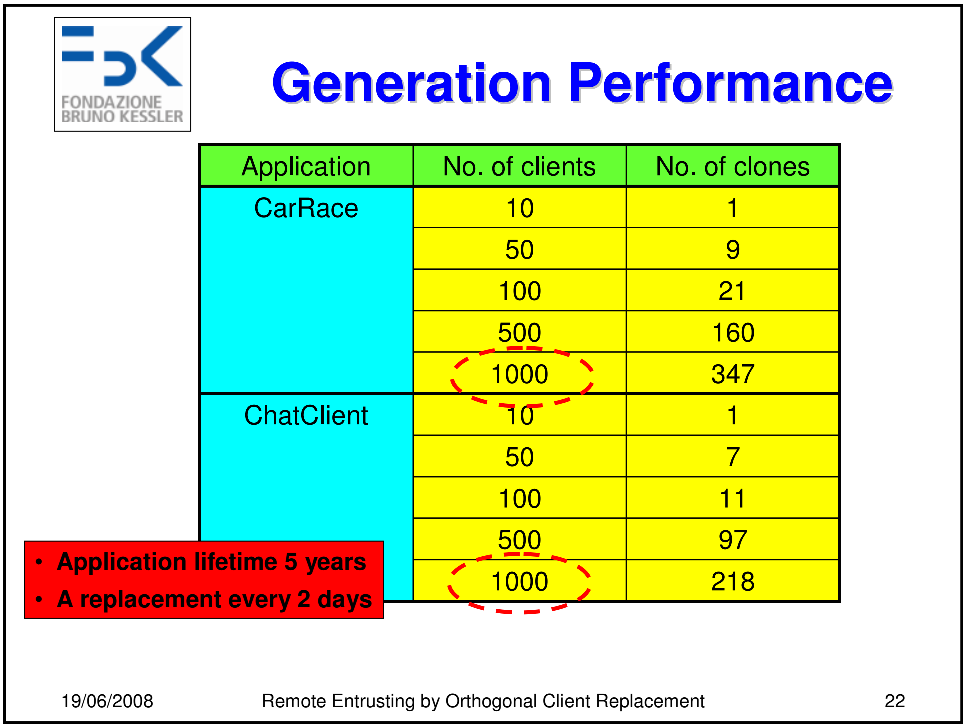
- Algorithm is fast
- Too many false positives
  - Clients contains clones that could leak information to an attacker

```
repeat
    $CP_i$ = RendomTransform (CP)
    CP = $CP_i$
    ($C_i$, $S_i$) = MoveCompToServer($CP_i$, $C_1$,…,$C_{i-1}$)
until $(C_i \perp C_1) \wedge \ldots \wedge (C_i \perp C_{i-1})$
```



Clone size threshold

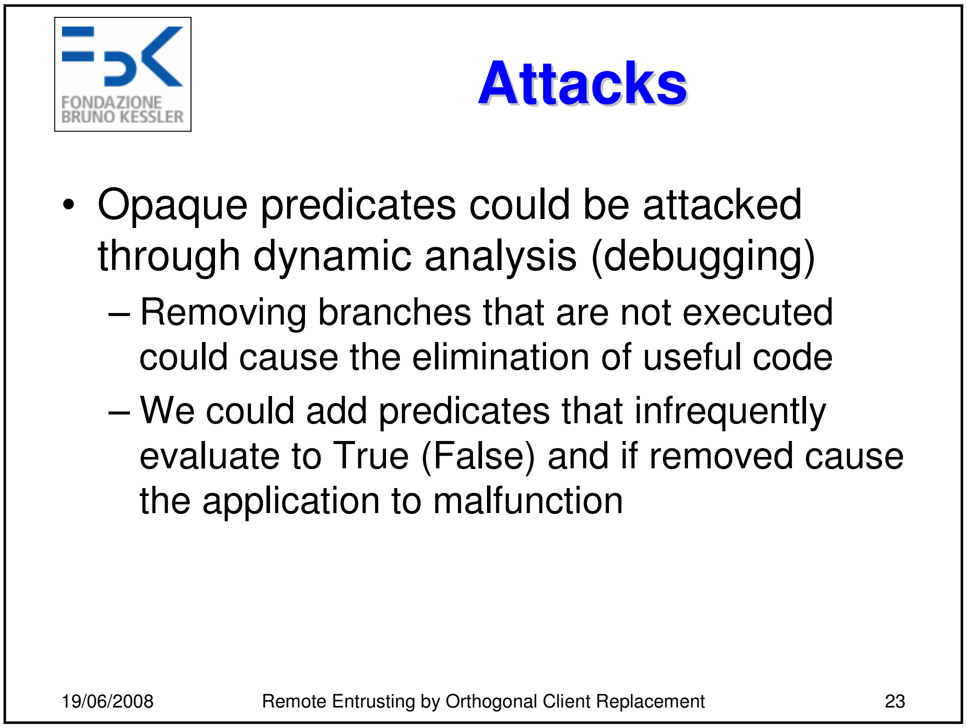
Application	Min. clone length		Clones
	Statements	Tokens	
CarRace	1	14	123
	2	28	33
	3	42	6
	4	56	1
	5	70	0
ChatClinet	1	12	69
	2	24	27
	3	36	5
	4	48	1
	5	60	0



Generation Performance

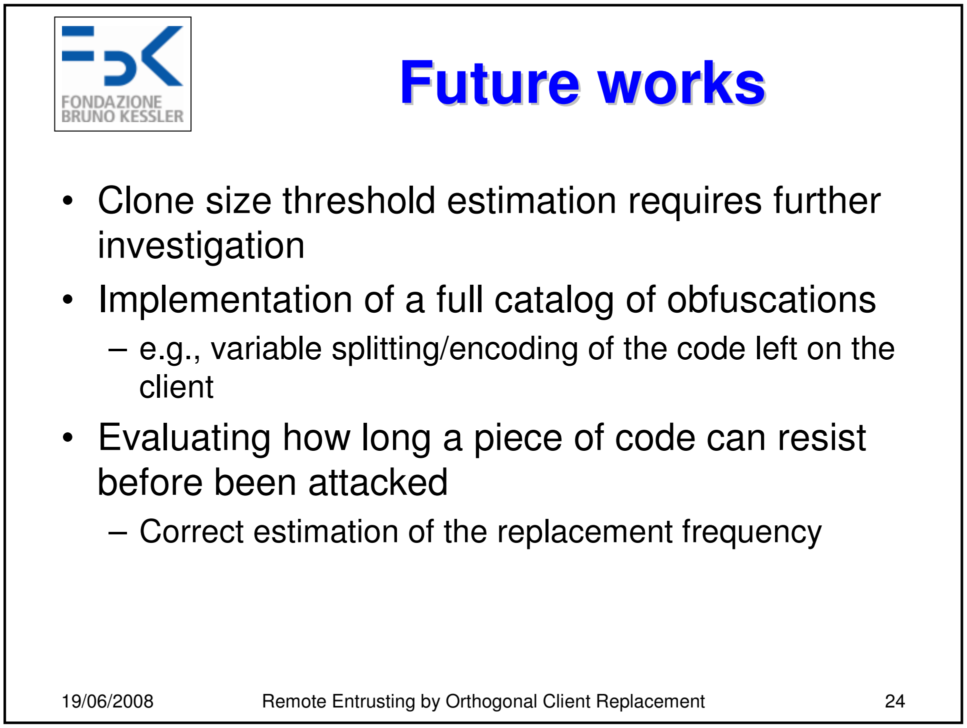
Application	No. of clients	No. of clones
CarRace	10	1
	50	9
	100	21
	500	160
	1000	347
ChatClient	10	1
	50	7
	100	11
	500	97
	1000	218

- **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



Future works

- Clone size threshold estimation requires further investigation
- Implementation of a full catalog of obfuscations
  - e.g., variable splitting/encoding of the code left on the client
- Evaluating how long a piece of code can resist before been attacked
  - Correct estimation of the replacement frequency