Software Protection: How to Crack Programs, and Defend Against Cracking

Lecture 7: Tamperproofing II

Moscow State University, Spring 2014

Christian Collberg
University of Arizona

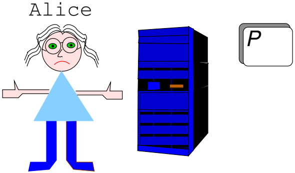
`www.cs.arizona.edu/~collberg`

Overview

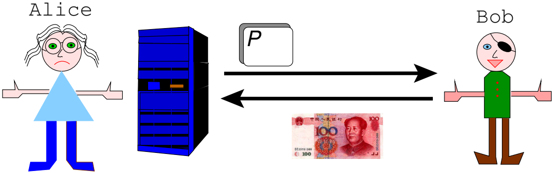
1. Distributed Software Protection Scenarios
2. R-MATE Protection Ideas
3. Algorithms
4. The Tigress System

R-MATE Scenarios

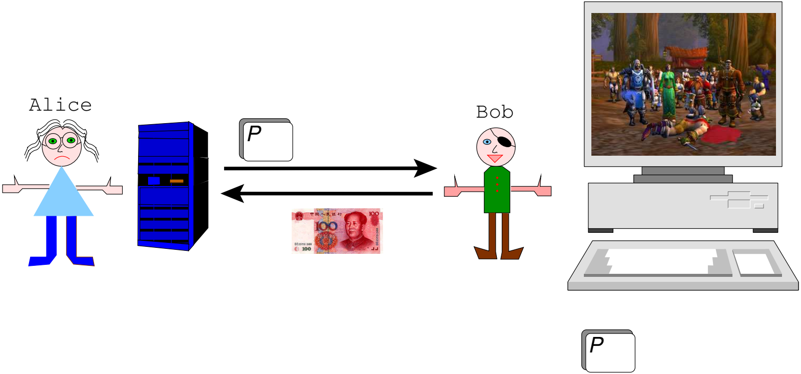
Scenario: Protecting networked computer games



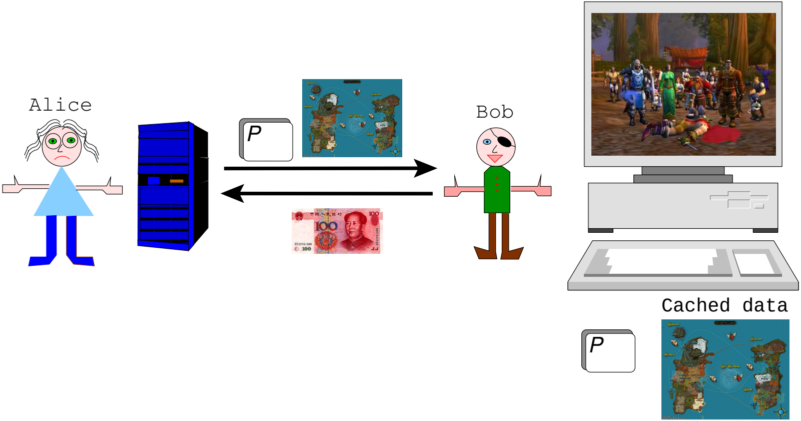
Scenario: Protecting networked computer games



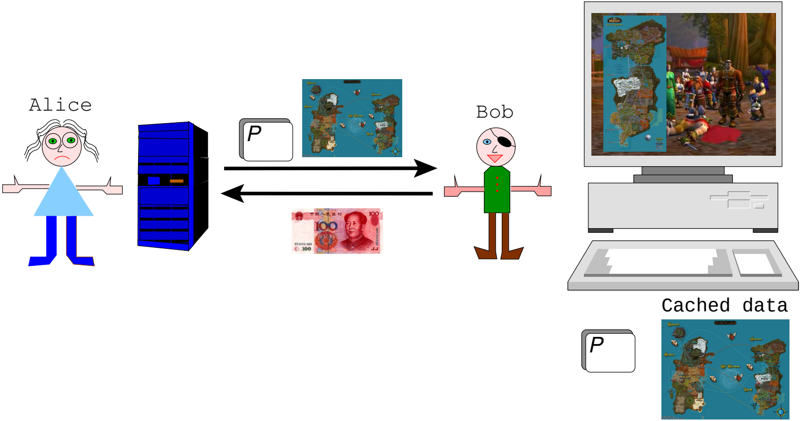
Scenario: Protecting networked computer games



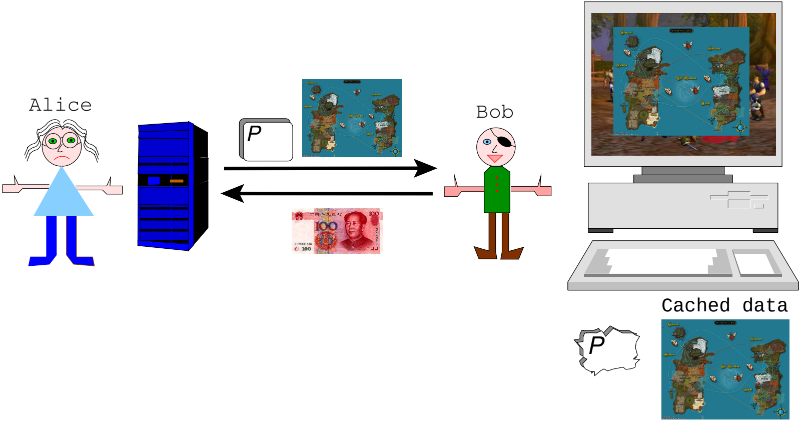
Scenario: Protecting networked computer games



Scenario: Protecting networked computer games



Scenario: Protecting networked computer games



Alice

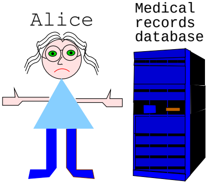
P

Bob

Cached data

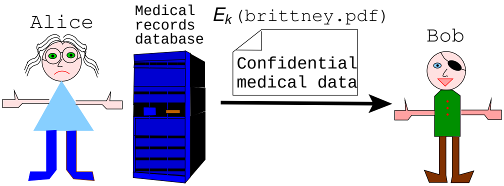
P

Scenario: Protecting medical records



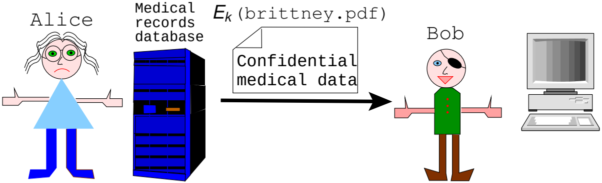
- Medical records must be protected from improper access and improper modification.
- Records are stored on one secure site, accessed from multiple (sometimes mobile) devices.

Scenario: Protecting medical records



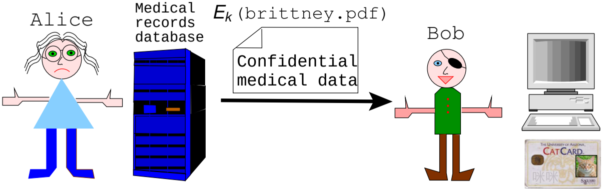
- Medical records must be protected from improper access and improper modification.
- Records are stored on one secure site, accessed from multiple (sometimes mobile) devices.

Scenario: Protecting medical records



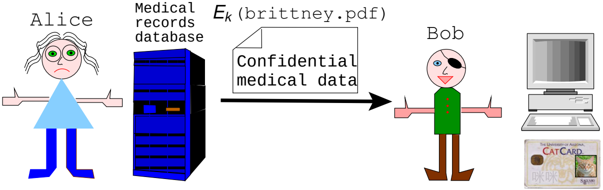
- Medical records must be protected from improper access and improper modification.
- Records are stored on one secure site, accessed from multiple (sometimes mobile) devices.

Scenario: Protecting medical records



- Medical records must be protected from improper access and improper modification.
- Records are stored on one secure site, accessed from multiple (sometimes mobile) devices.

Scenario: Protecting medical records



Alice

Medical
records
database

E_k (brittney.pdf)

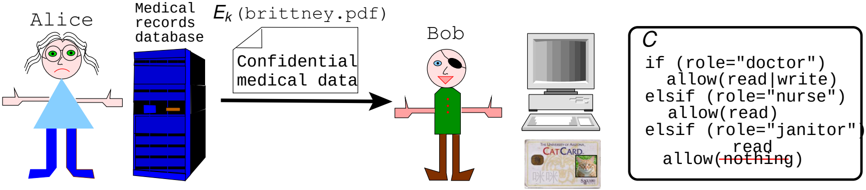
Confidential
medical data

Bob

```
C
if (role="doctor")
   allow(read|write)
elsif (role="nurse")
   allow(read)
elsif (role="janitor")

   allow(nothing)
```

- Medical records must be protected from improper access and improper modification.
- Records are stored on one secure site, accessed from multiple (sometimes mobile) devices.

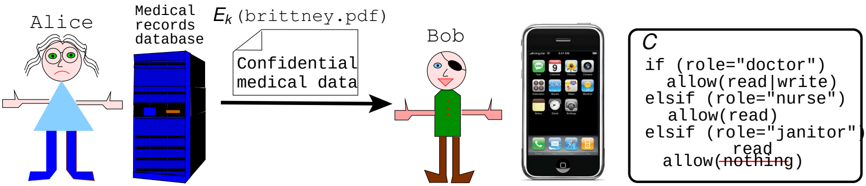
Scenario: Protecting medical records



```
if (role="doctor")
  allow(read|write)
elsif (role="nurse")
  allow(read)
elsif (role="janitor")
             read
  allow(~~nothing~~)
```

- Medical records must be protected from improper access and improper modification.
- Records are stored on one secure site, accessed from multiple (sometimes mobile) devices.

Scenario: Protecting medical records

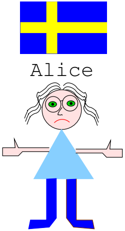


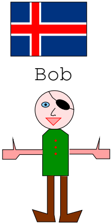
C

```
if (role="doctor")
   allow(read|write)
elsif (role="nurse")
   allow(read)
elsif (role="janitor")
                 read
  allow(~~nothing~~)
```

- Medical records must be protected from improper access and improper modification.
- Records are stored on one secure site, accessed from multiple (sometimes mobile) devices.

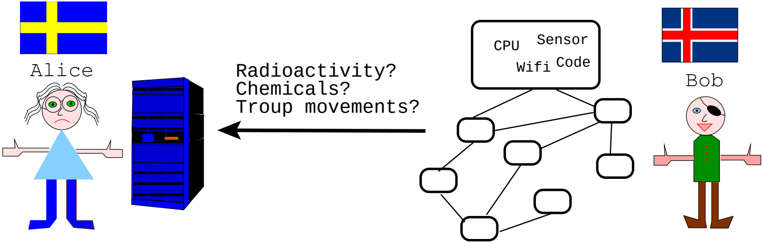
Scenario: Wireless sensor networks





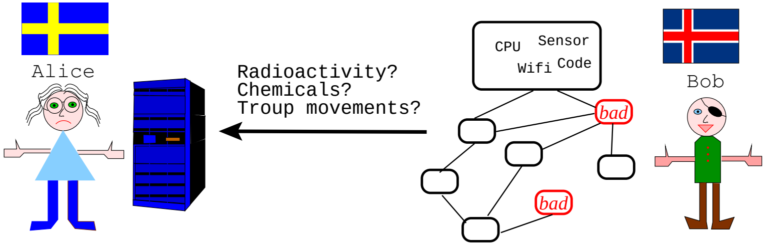
- Sensor networks are common in military scenarios.

Scenario: Wireless sensor networks



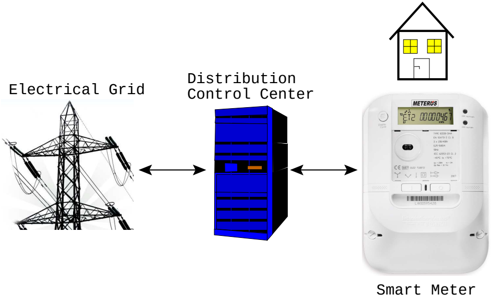
- Sensor networks are common in military scenarios.

Scenario: Wireless sensor networks



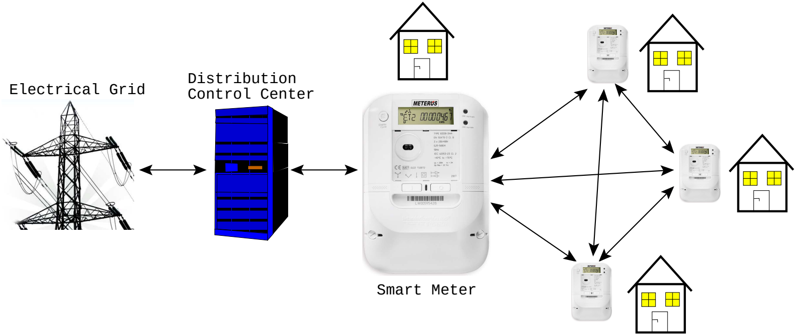
- Sensor networks are common in military scenarios.
- The enemy can intercept/analyze/modify sensors.

Scenario: Advanced Metering Infrastructure



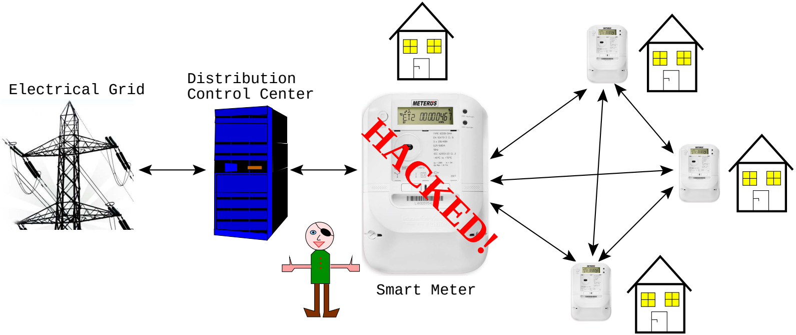
- Selective black-outs, consumers can adjust usage based on current costs, small-scale energy production, . . .

Scenario: Advanced Metering Infrastructure



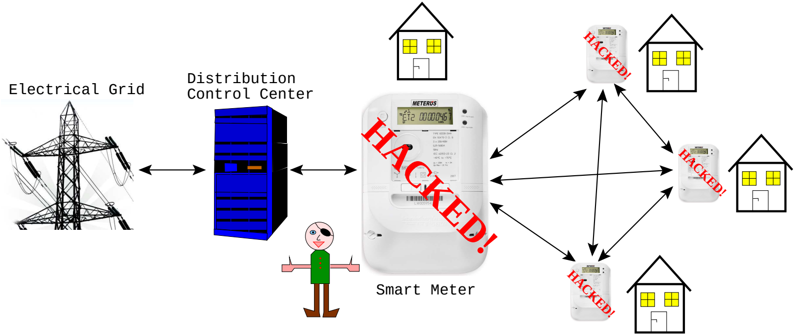
- Selective black-outs, consumers can adjust usage based on current costs, small-scale energy production, . . .

Scenario: Advanced Metering Infrastructure



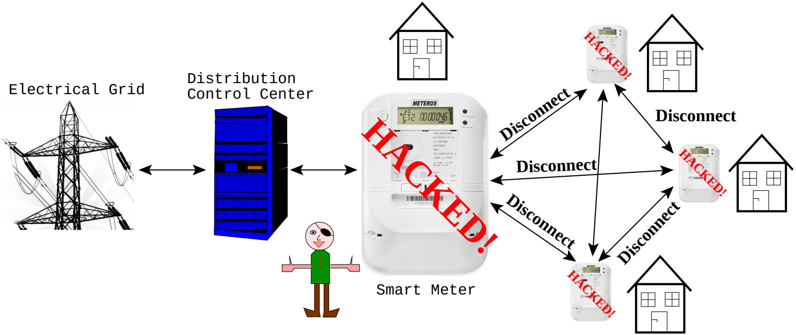
- Selective black-outs, consumers can adjust usage based on current costs, small-scale energy production, . . .

Scenario: Advanced Metering Infrastructure



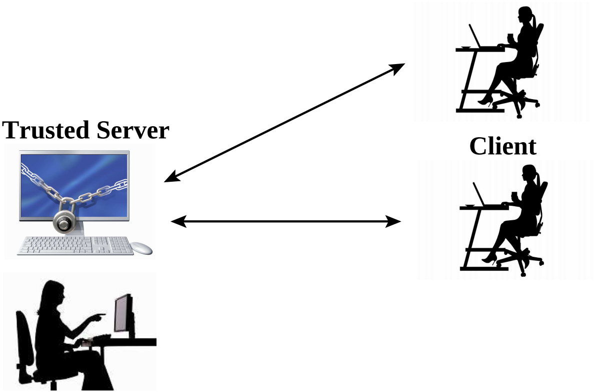
- Selective black-outs, consumers can adjust usage based on current costs, small-scale energy production, . . .

Scenario: Advanced Metering Infrastructure



- Selective black-outs, consumers can adjust usage based on current costs, small-scale energy production, . . .

The Remote Man-At-The-End Problem

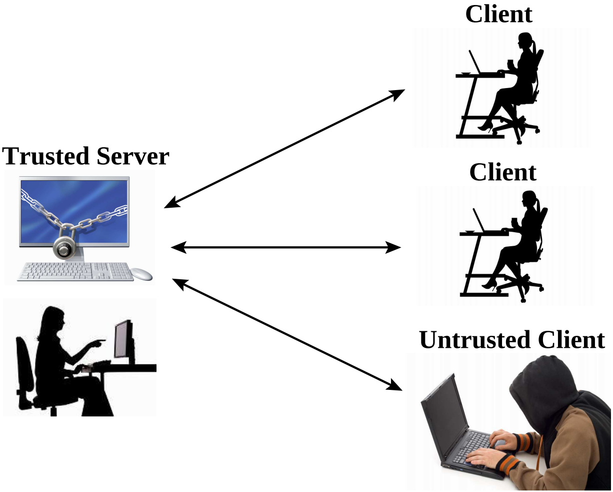


Client

Trusted Server

Client

The Remote Man-At-The-End Problem



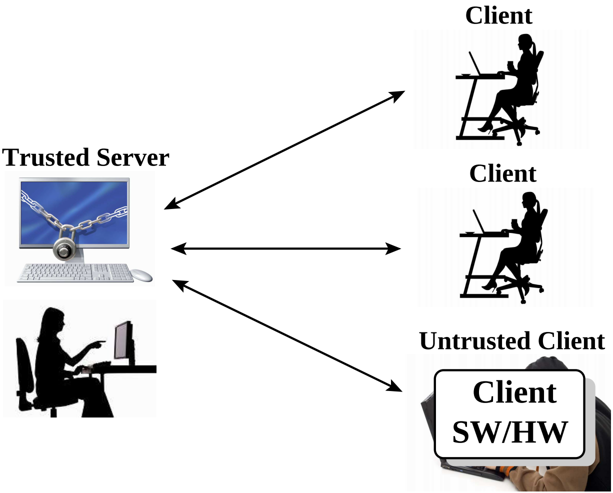
Client

Trusted Server

Client

Untrusted Client

The Remote Man-At-The-End Problem



Trusted Server

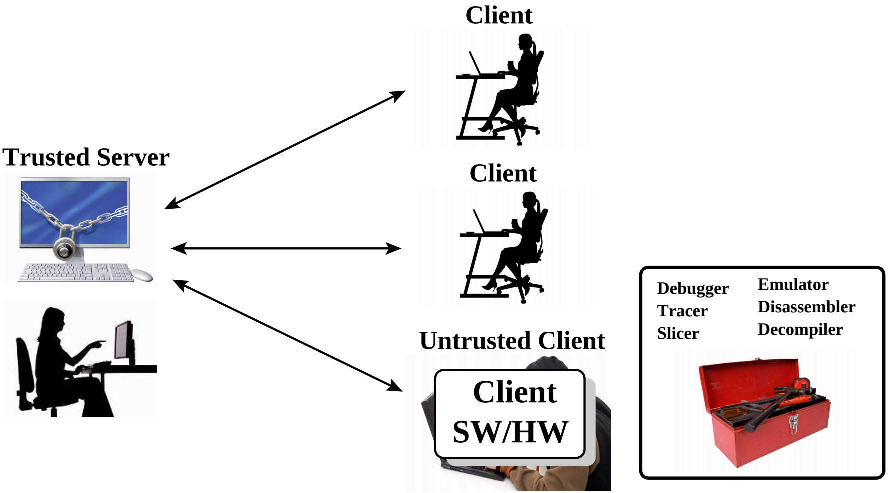
Client

Client

Untrusted Client

Client SW/HW

The Remote Man-At-The-End Problem



Trusted Server

Client

Client

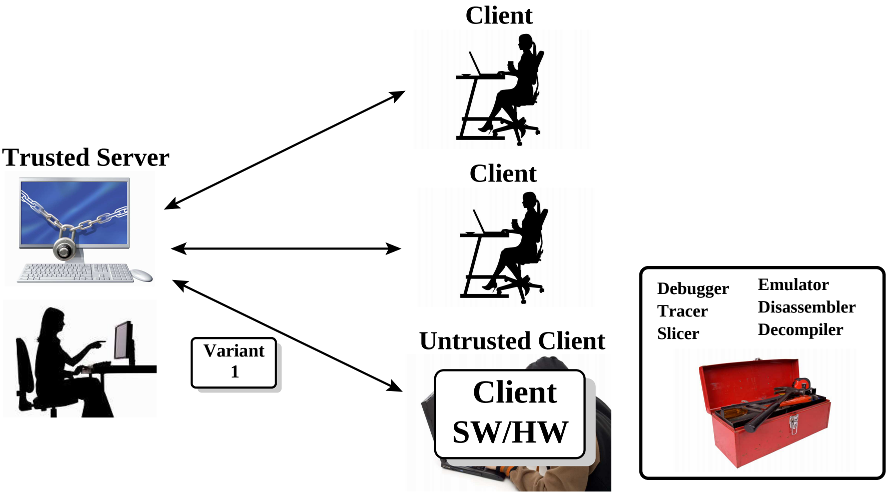
Untrusted Client

Client SW/HW

Debugger
Tracer
Slicer

Emulator
Disassembler
Decompiler

The Remote Man-At-The-End Problem



Client

Client

Trusted Server

Variant 1

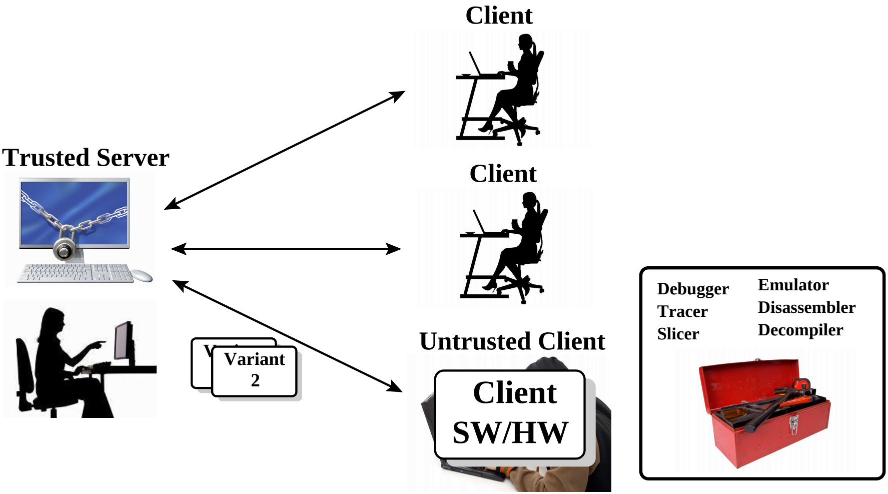
Untrusted Client

Client SW/HW

Debugger
Tracer
Slicer

Emulator
Disassembler
Decompiler

The Remote Man-At-The-End Problem



Trusted Server

Client

Client

Untrusted Client

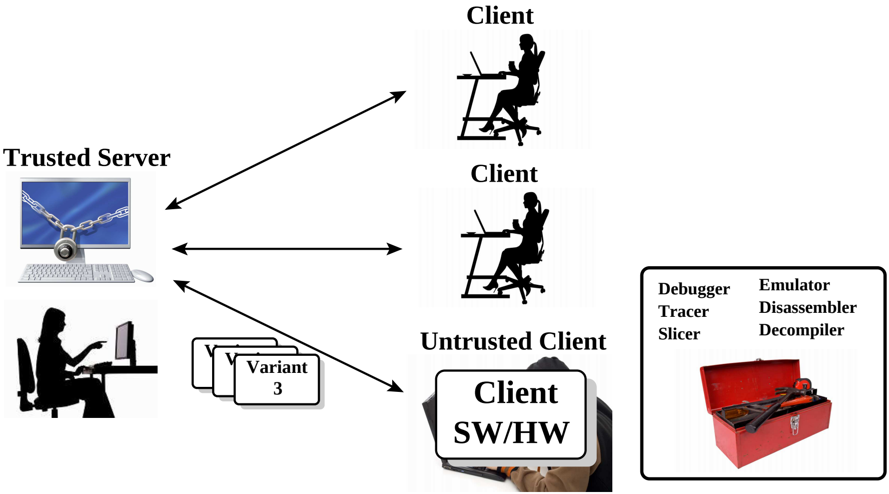
Client SW/HW

Variant 2

Debugger
Tracer
Slicer

Emulator
Disassembler
Decompiler

The Remote Man-At-The-End Problem



Client

Client

Trusted Server

Variant
3

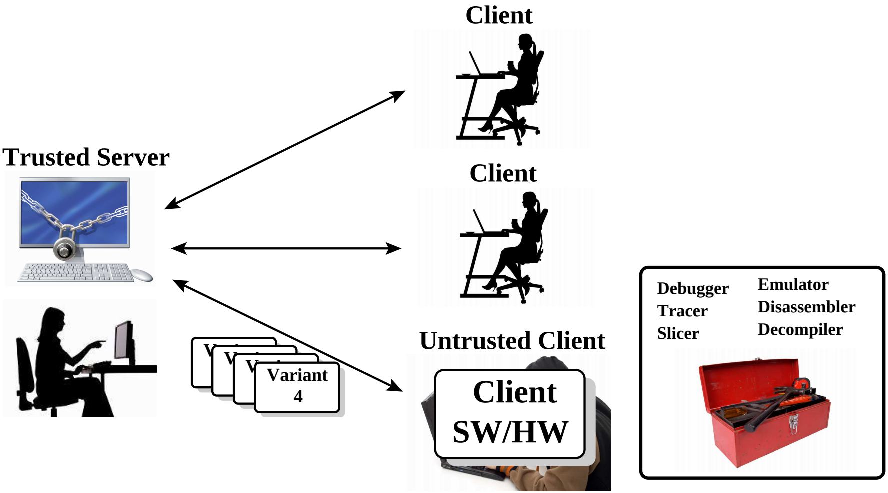
Untrusted Client

Client
SW/HW

Debugger
Tracer
Slicer

Emulator
Disassembler
Decompiler

The Remote Man-At-The-End Problem



Trusted Server

Client

Client

Untrusted Client

Client SW/HW

V

V

V

Variant 4

Debugger
Tracer
Slicer

Emulator
Disassembler
Decompiler

Self−protect
against
tampering!

Debugger
Tracer
Slicer

Emulator
Disassembler
Decompiler

Definition (Man-At-The-End (MATE) Attacks)

MATE attacks occur in any setting where an adversary has physical access to a device and compromises it by inspecting, reverse engineering, or tampering with its hardware or software.

Debugger
Tracer
Slicer

Emulator
Disassembler
Decompiler

Detect remote tampering!

Self−protect against tampering!

Definition (Man-At-The-End (MATE) Attacks)

MATE attacks occur in any setting where an adversary has physical access to a device and compromises it by inspecting, reverse engineering, or tampering with its hardware or software.

Definition (Remote MATE (R-MATE) Attacks)

R-MATE attacks occur in distributed systems where untrusted clients are in frequent communication with trusted servers over a network, and where a malicious user can get an advantage by compromising an untrusted device.

Protection Ideas

Algorithm Ideas

1. Split — move functionality from untrusted to trusted site.

Algorithm Ideas

1. Split — move functionality from untrusted to trusted site.
2. Measure — ask untrusted site “are you running the right code?”

Algorithm Ideas

1. Split — move functionality from untrusted to trusted site.
2. Measure — ask untrusted site “are you running the right code?”
3. Time — make untrusted site compute challenge within given time.

Algorithm Ideas

1. Split — move functionality from untrusted to trusted site.
2. Measure — ask untrusted site “are you running the right code?”
3. Time — make untrusted site compute challenge within given time.
4. Monitor — monitor messages to detect signs of tampering.

Algorithm Ideas. . .

5. Hardware — make untrusted site run tamperproof hardware.

Algorithm Ideas. . .

5. Hardware — make untrusted site run tamperproof hardware.
6. Encrypt — make untrusted site compute in encrypted domain.

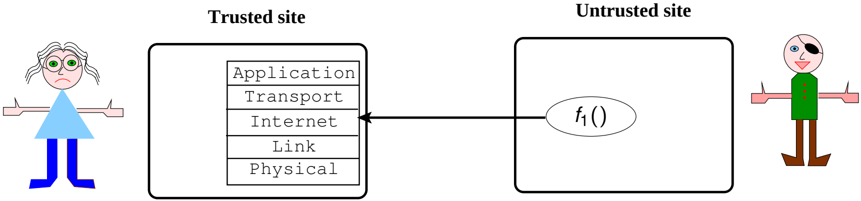
Algorithm Ideas. . .

5. Hardware — make untrusted site run tamperproof hardware.
6. Encrypt — make untrusted site compute in encrypted domain.
7. Update — make untrusted site continuously update its code.

Algorithm Ideas. . .

5. Hardware — make untrusted site run tamperproof hardware.
6. Encrypt — make untrusted site compute in encrypted domain.
7. Update — make untrusted site continuously update its code.
8. Local — obfuscate/tamperproof/watermark/. . . code.

Protocol Monitoring



Trusted site

Application

Transport

Internet

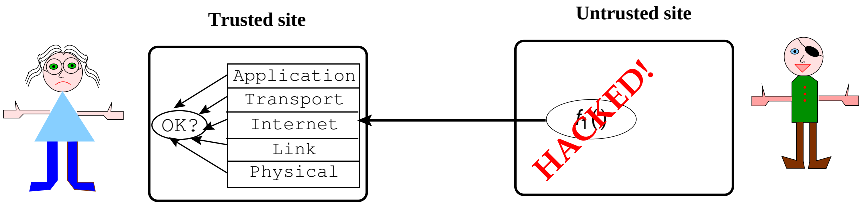
Link

Physical

Untrusted site

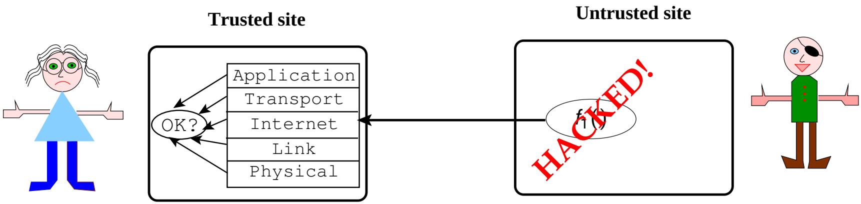
$f_1()$

Protocol Monitoring



- Monitor messages to detect signs of tampering

Protocol Monitoring



Trusted site

Application
Transport
Internet
Link
Physical

OK?

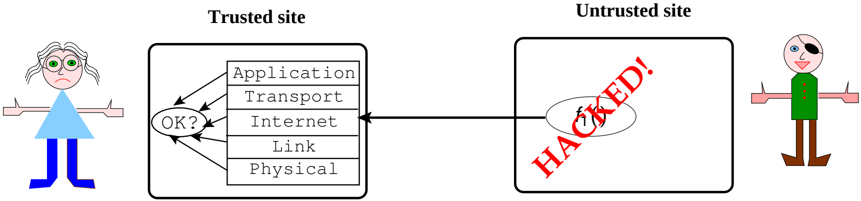
Untrusted site

$f_1()$

HACKED!

- Monitor messages to detect signs of tampering
- Not all tampering will violate protocols!

Protocol Monitoring



Trusted site

Untrusted site

Application

Transport

Internet

Link

Physical

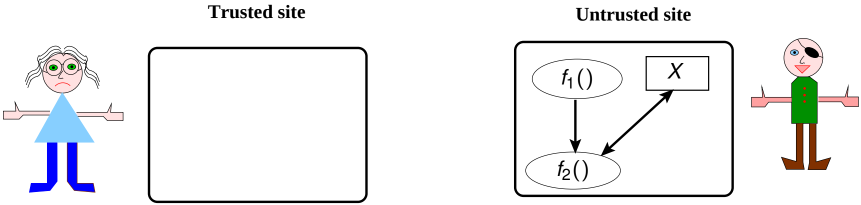
OK?

$f_1()$

HACKED!

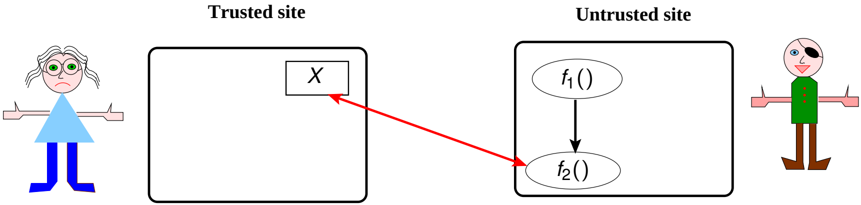
- Monitor messages to detect signs of tampering
- Not all tampering will violate protocols!
- Need to monitor every level of the network stack?

Code Splitting



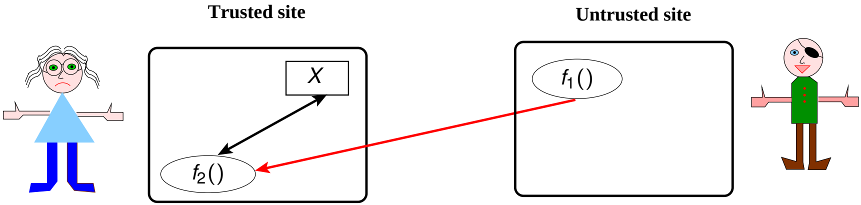
- Move functionality from untrusted to trusted site.

Code Splitting



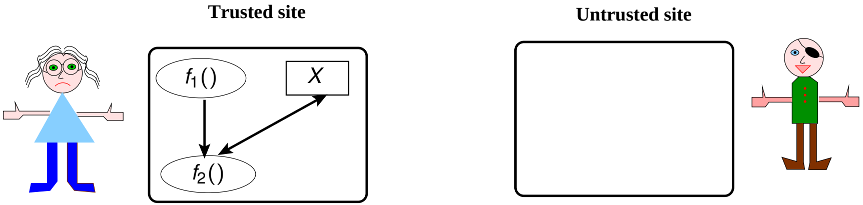
- Move functionality from untrusted to trusted site.
- Increases network traffic, server load.

Code Splitting



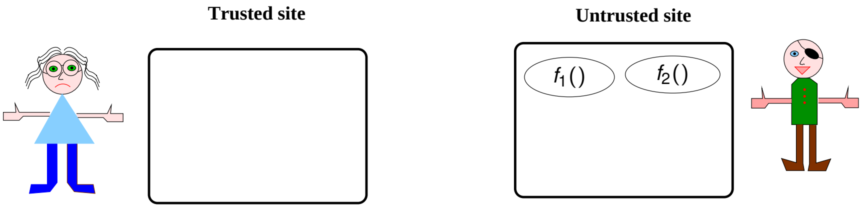
- Move functionality from untrusted to trusted site.
- Increases network traffic, server load.

Code Splitting



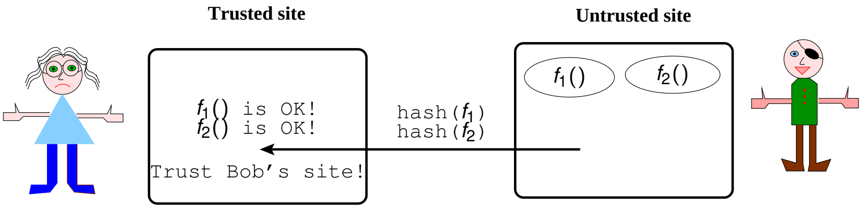
- Move functionality from untrusted to trusted site.
- Increases network traffic, server load.
- Extreme: Run all code server-side.

Trusted Hardware



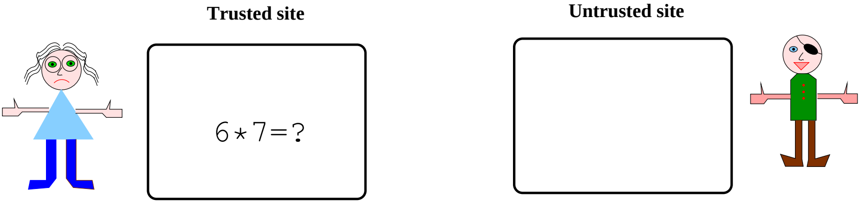
- Bob has a trusted hardware unit.

Trusted Hardware



- Bob has a trusted hardware unit.
- Bob proves that his site contains no untrustworthy software.
- Trusted hardware makes it harder for Bob to cheat.

Encryption



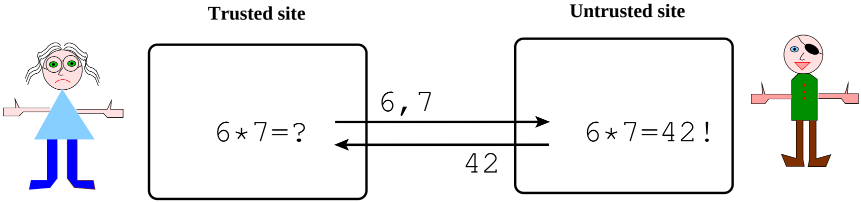
Trusted site

$6 * 7 = ?$

Untrusted site

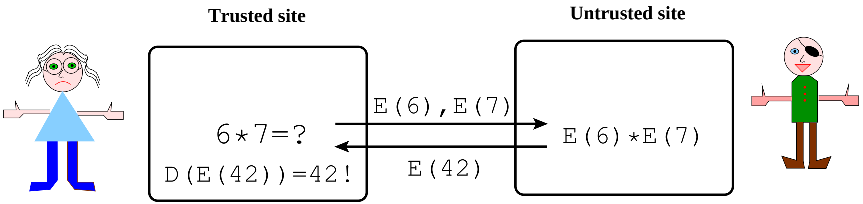
- Alice wants to outsource computation to Bob

Encryption



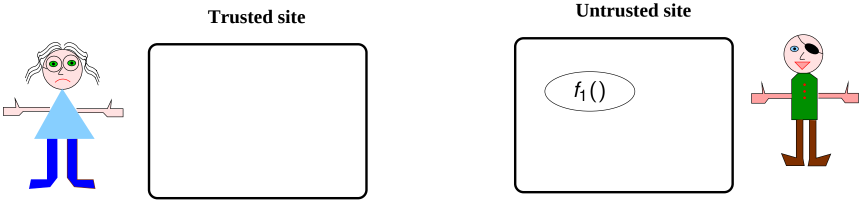
- Alice wants to outsource computation to Bob
- Doesn’t want him to learn her inputs and outputs!

Encryption



- Alice wants to outsource computation to Bob
- Doesn't want him to learn her inputs and outputs!
- Bob performs operations on encrypted data.

Continuous Evolution

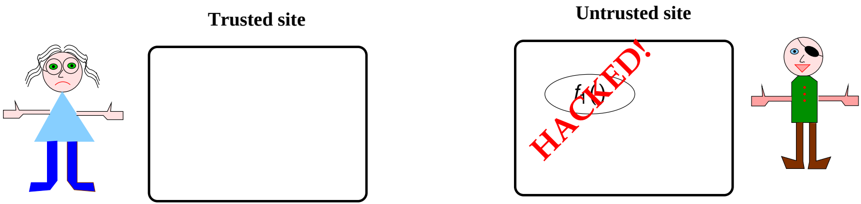


Trusted site

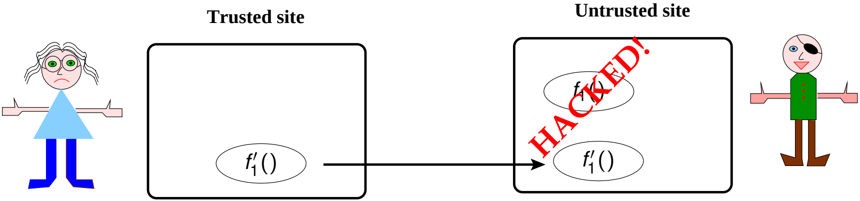
Untrusted site

$f_1()$

Continuous Evolution

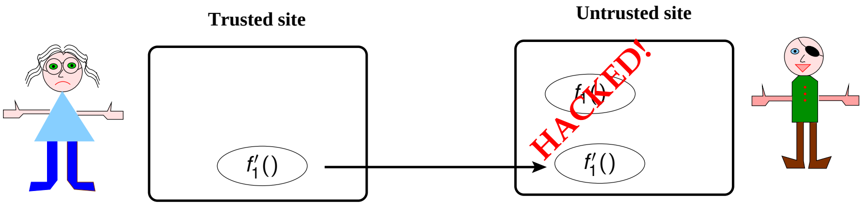


Continuous Evolution



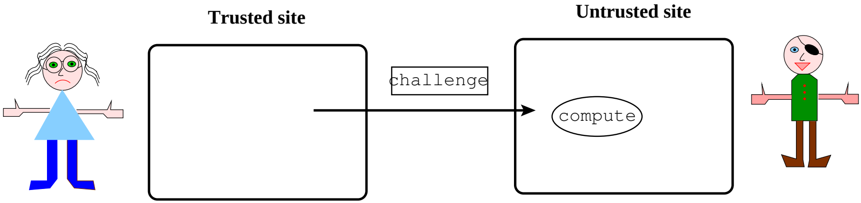
- The server continously updates the client code

Continuous Evolution



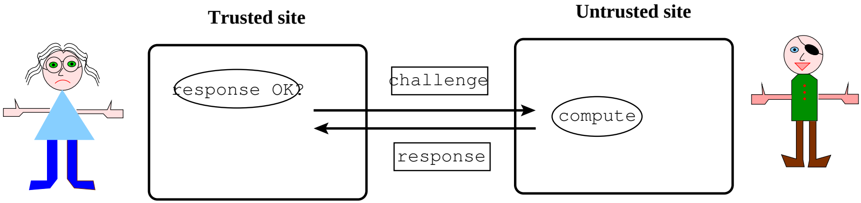
- The server continously updates the client code
- Gives Bob a smaller window to hack!

Challenge Timing



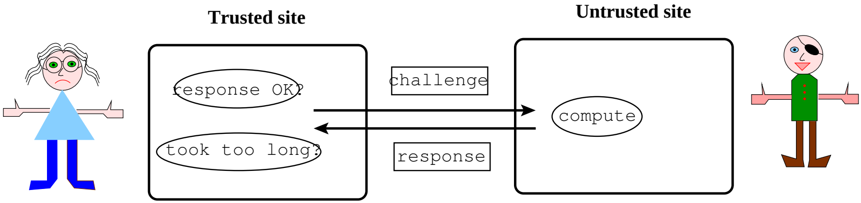
- Alice asks Bob to compute a special function

Challenge Timing



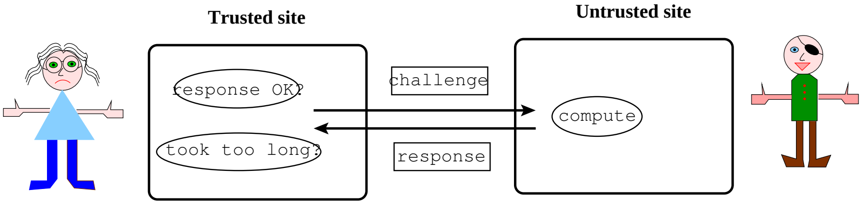
- Alice asks Bob to compute a special function
- Does it return the right result?

Challenge Timing



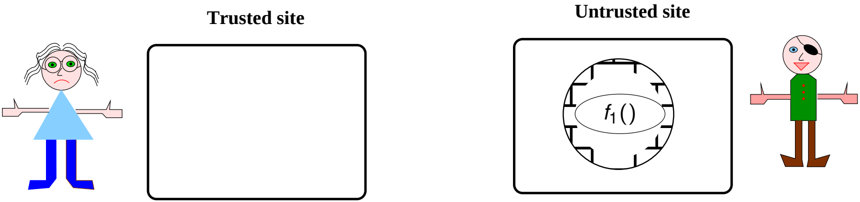
- Alice asks Bob to compute a special function
- Does it return the right result?
- Does Bob return the result fast enough?

Challenge Timing



- Alice asks Bob to compute a special function
- Does it return the right result?
- Does Bob return the result fast enough?
- Accurate timing on the general Internet is hard. . .

Local Defenses



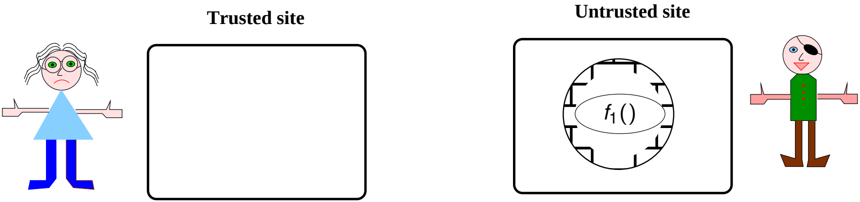
Trusted site

Untrusted site

$f_1()$

- Local defenses don’t involve the trusted site

Local Defenses



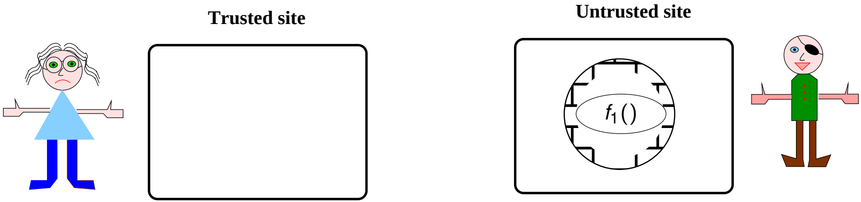
Trusted site

Untrusted site

$f_1()$

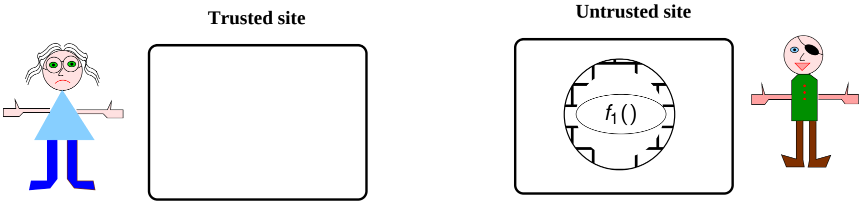
- Local defenses don’t involve the trusted site
- Hash the executable. . .

Local Defenses



- Local defenses don't involve the trusted site
- Hash the executable. . .
- Hash the state. . .

Local Defenses



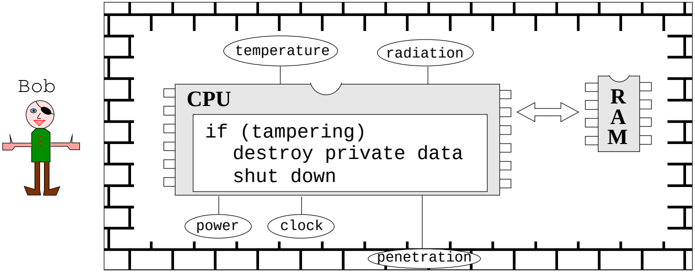
Trusted site

Untrusted site

$f_1()$

- Local defenses don’t involve the trusted site
- Hash the executable. . .
- Hash the state. . .
- Obfuscate. . .

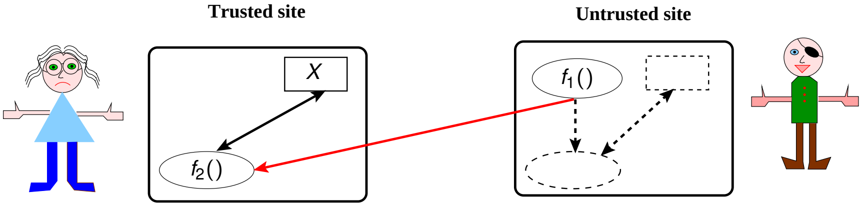
Local Defenses — Hardened Processors



- Hardware can be hardened against attack.
- Consequences for cost, heat, clock-rate, energy-use. . .

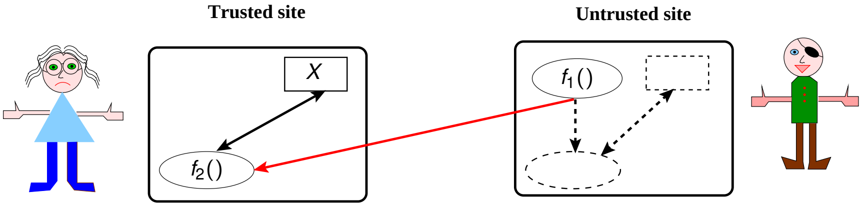
Slicing functions

Move all client code server-side



- High compute load for the server and high latency for the client.

Move some client code server-side



- Intermediate level solution:
  - some computation on the server, some on the client.
  - balance computation, network traffic, tamper-detection.
- Use slicing algorithms.

```
int f(int x,int y){
   int a = 4*x + y;

   int c;
   if (y < 5)
      c = a*x+4;
   else
     c = 2*x+4;

   int sum = 0;
   for(int i=a;i<10;i++)
      sum += i;

   return x*(sum+c);
}
```

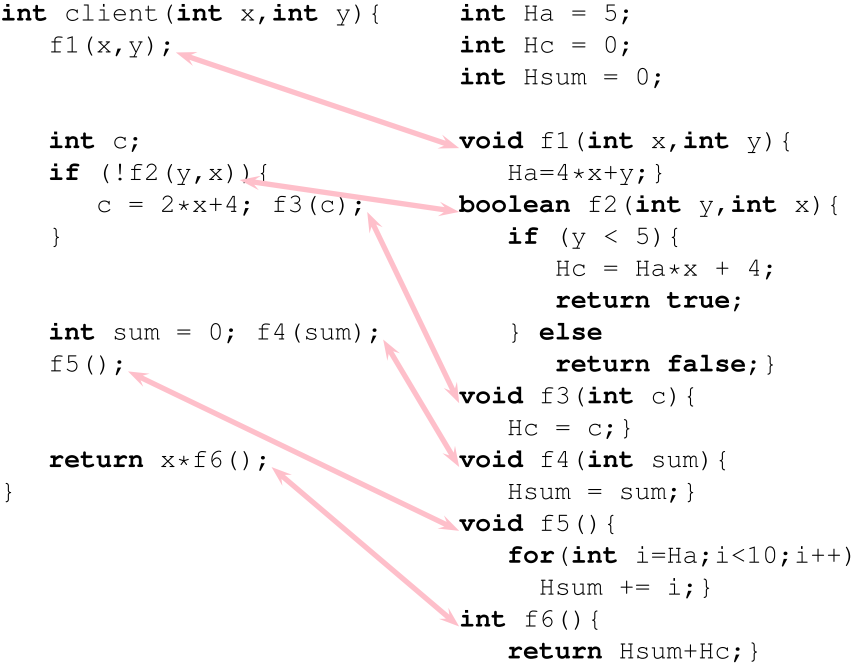
- a is an important variable — hide it on the server!
- Whenever the client needs a — get it from the server!
- Move code that depends on a to the server — better performance!

```
int f(int x,int y){
   int a = 4*x + y;

   int c;
   if (y < 5)
      c = a*x+4;
   else
     c = 2*x+4;

   int sum = 0;
   for(int i=a;i<10;i++)
      sum += i;

   return x*(sum+c);
}
```

- Compute a forward slice from a — move this code to the server!
- Keep unimportant variable c on both the client and the server — better performance!
- Don’t move large data structures — better performance!
- Overhead depends in how much of the program is hidden on the server. On a LAN: 3 to 58%.

```
int client(int x,int y){
   f1(x,y);


   int c;
   if (!f2(y,x)){
      c = 2*x+4; f3(c);
   }


   int sum = 0; f4(sum);
   f5();


   return x*f6();
}
```

```
int Ha = 5;
int Hc = 0;
int Hsum = 0;

void f1(int x,int y){
   Ha=4*x+y;}
boolean f2(int y,int x){
   if (y < 5){
      Hc = Ha*x + 4;
      return true;
   } else
      return false;}
void f3(int c){
   Hc = c;}
void f4(int sum){
   Hsum = sum;}
void f5(){
   for(int i=Ha;i<10;i++)
     Hsum += i;}
int f6(){
   return Hsum+Hc;}
```



```
int client(int x,int y){
   f1(x,y);


   int c;
   if (!f2(y,x)){
      c = 2*x+4; f3(c);
   }


   int sum = 0; f4(sum);
   f5();


   return x*f6();
}
```

```
int Ha = 5;
int Hc = 0;
int Hsum = 0;

void f1(int x,int y){
   Ha=4*x+y;}
boolean f2(int y,int x){
   if (y < 5){
      Hc = Ha*x + 4;
      return true;
   } else
      return false;}
void f3(int c){
   Hc = c;}
void f4(int sum){
   Hsum = sum;}
void f5(){
   for(int i=Ha;i<10;i++)
     Hsum += i;}
int f6(){
   return Hsum+Hc;}
```

Example

- Function `f` is the original one
- You want to hide variable `a`
- Compute a forward slice on `a` (pink).
- You want to protect all the pink code ⇒ put it on the server in functions `Hf1...Hf6`.
- The client accesses the hidden functions by making RPCs.
- `c` is a partially hidden variable. It resides both on the client and the server, but the code that updates it is split between the two.

Performance

- Runtime overhead from 3 to 58%.

Performance

- Runtime overhead from 3 to 58%.
- Depends on the amount of protection that is added:
  1. how much of the program is hidden on the server?
  2. how much extra communication?

Performance

- Runtime overhead from 3 to 58%.
- Depends on the amount of protection that is added:
  1. how much of the program is hidden on the server?
  2. how much extra communication?
- Zhang and Gupta's measurements were done over a local area network!

Performance

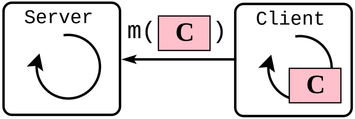
Packet turnaround times:

target site	**# hops**	**ms**
`rorohiko.cs.arizona.edu`	1	0.2
`cse.asu.edu`	10	5
`www.stanford.edu`	12	25
`www.usp.ac.fj`	12	153
`www.eltech.ru`	23	201
`www.tsinghua.edu.cn`	19	209

Verification by timing

Pioneer

- In a *very* restricted environment you can *measure* aspects of the untrusted client to verify that it is running the correct software.



Assumptions

1. The the client's hardware configuration is known;
2. The client-server latency is known;
3. The client can only communicate with the server.

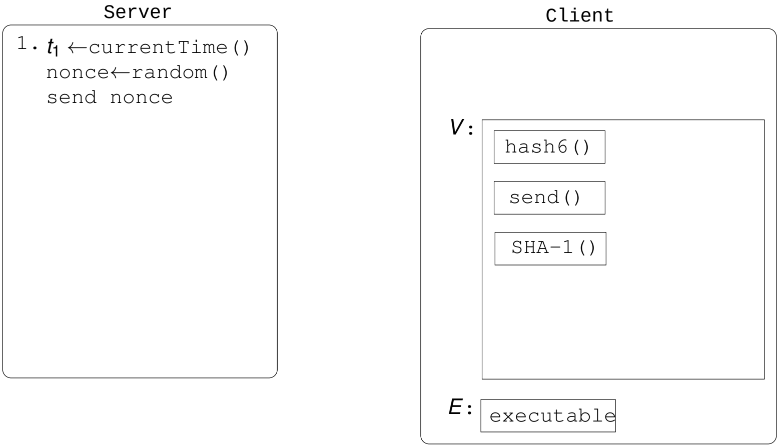
Applications

1. Check cell phone/PDA/smartcard for viruses;
2. Check voting machine code;
3. Check for rootkits on machines on your LAN.

Algorithm

- Basic idea: ask client for a hash of its code.
- If
  1. the hash is the wrong value, or
  2. the computation took too long

  the client has cheated!
- The hash function is constructed such that it can't be computed quicker.



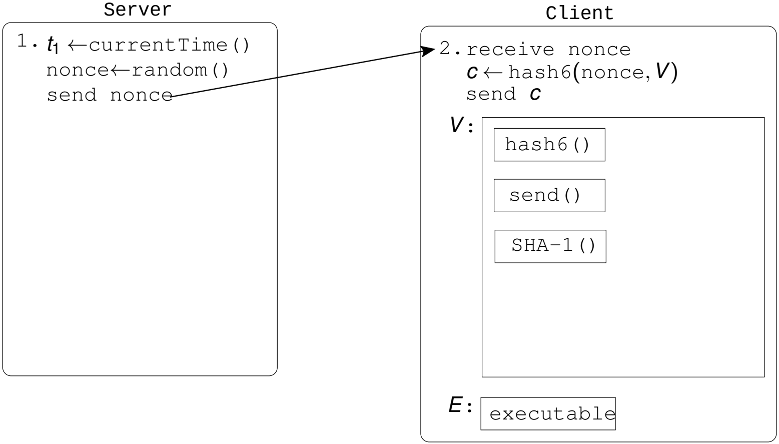
Server

```
1. $t_1$ ←currentTime()
   nonce←random()
   send nonce
```

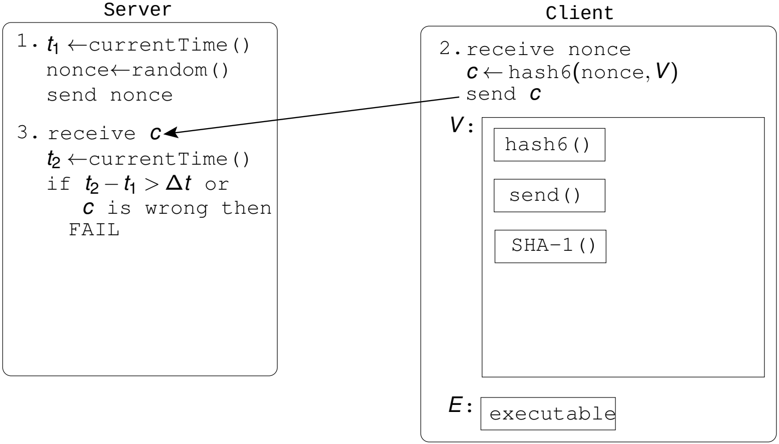
Client

V: hash6() send() SHA-1()

E: executable

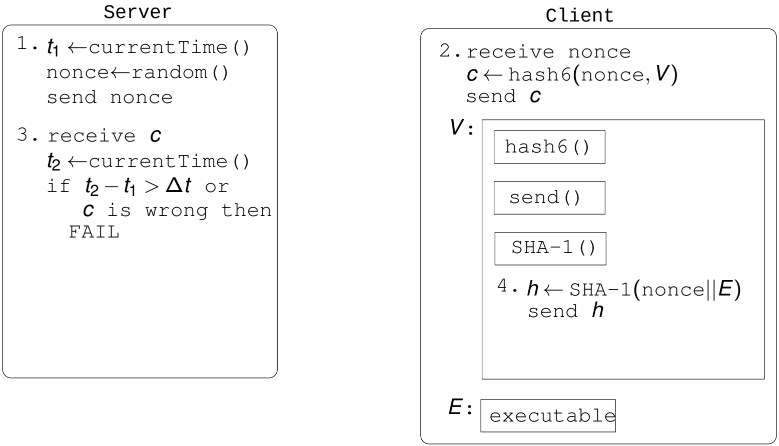


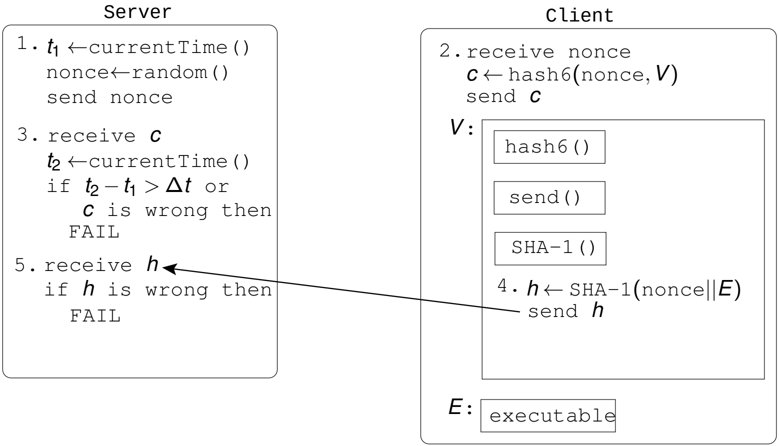
```
1. $t_1$ ←currentTime()
   nonce←random()
   send nonce

2. receive nonce
   $c$ ← hash6(nonce, $V$)
   send $c$
```

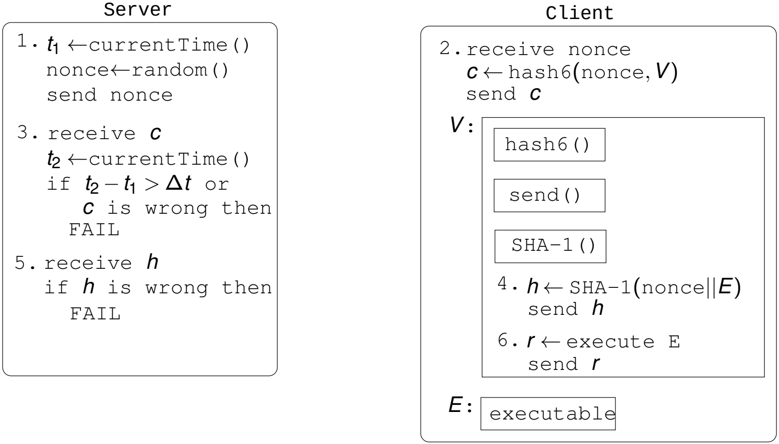


```
1. $t_1 \leftarrow$ currentTime()
   nonce$\leftarrow$random()
   send nonce

3. receive $c$
   $t_2 \leftarrow$ currentTime()
   if $t_2 - t_1 > \Delta t$ or
      $c$ is wrong then
     FAIL
```

```
2. receive nonce
   $c \leftarrow$ hash6(nonce, $V$)
   send $c$
```

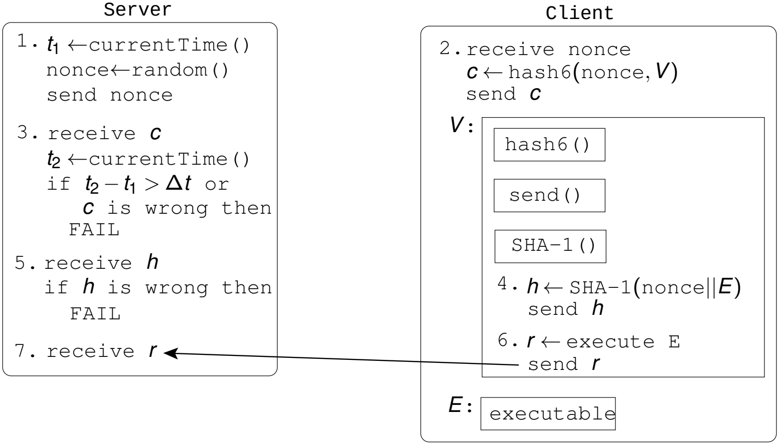

Server
1. $t_1 \leftarrow$currentTime()
nonce←random()
send nonce
3. receive c
$t_2 \leftarrow$currentTime()
if $t_2 - t_1 > \Delta t$ or
c is wrong then
FAIL
Client
2. receive nonce
$c \leftarrow$ hash6(nonce, V)
send c
V:
hash6()
send()
SHA-1()
4. $h \leftarrow$ SHA-1(nonce$||E$)
send h
E:
executable


Server
1. $t_1 \leftarrow$ currentTime()
nonce← random()
send nonce
3. receive c
$t_2 \leftarrow$ currentTime()
if $t_2 - t_1 > \Delta t$ or
c is wrong then
FAIL
5. receive h
if h is wrong then
FAIL
Client
2. receive nonce
$c \leftarrow$ hash6(nonce, V)
send c
V:
hash6()
send()
SHA-1()
4. $h \leftarrow$ SHA-1(nonce$||E$)
send h
E:
executable



```
1. $t_1 \leftarrow$ currentTime()
   nonce $\leftarrow$ random()
   send nonce
3. receive $c$
   $t_2 \leftarrow$ currentTime()
   if $t_2 - t_1 > \Delta t$ or
      $c$ is wrong then
     FAIL
5. receive $h$
   if $h$ is wrong then
     FAIL
```

```
2. receive nonce
   $c \leftarrow$ hash6(nonce, $V$)
   send $c$
$V$:
   hash6()
   send()
   SHA-1()
   4. $h \leftarrow$ SHA-1(nonce$||E$)
      send $h$
   6. $r \leftarrow$ execute E
      send $r$
$E$:
   executable
```


Server
1. $t_1 \leftarrow$currentTime()
nonce$\leftarrow$random()
send nonce
3. receive c
$t_2 \leftarrow$currentTime()
if $t_2 - t_1 > \Delta t$ or
c is wrong then
FAIL
5. receive h
if h is wrong then
FAIL
7. receive r
Client
2. receive nonce
$c \leftarrow$ hash6(nonce, V)
send c
V:
hash6()
send()
SHA-1()
4. $h \leftarrow$ SHA-1(nonce$||E$)
send h
6. $r \leftarrow$ execute E
send r
E:
executable

Algorithm

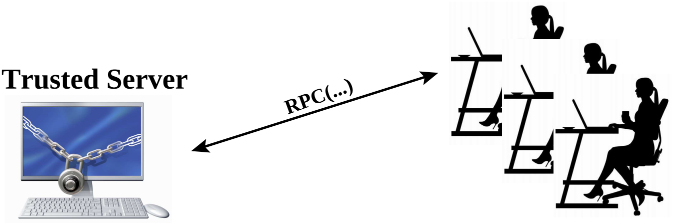
- The hash function must be time optimal, if not
  - the client can use the time he saved to execute his own instructions without the server noticing.

Algorithm

- The hash function must be time optimal, if not
    - the client can use the time he saved to execute his own instructions without the server noticing.
- Others have tried to extend the protocol to general scenarios — highly controversial.

The Tigress **System**

Tigress and the R-MATE Problem

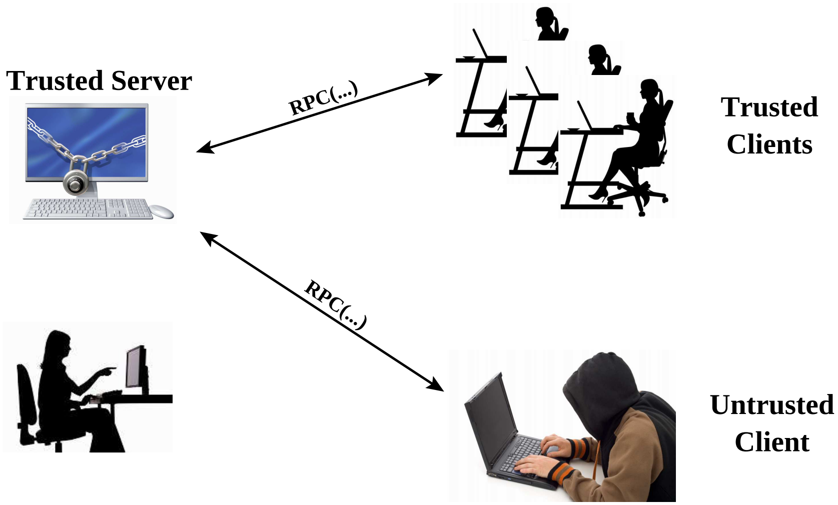


Trusted Server

RPC(...)

Trusted Clients

Tigress and the R-MATE Problem



Trusted Server

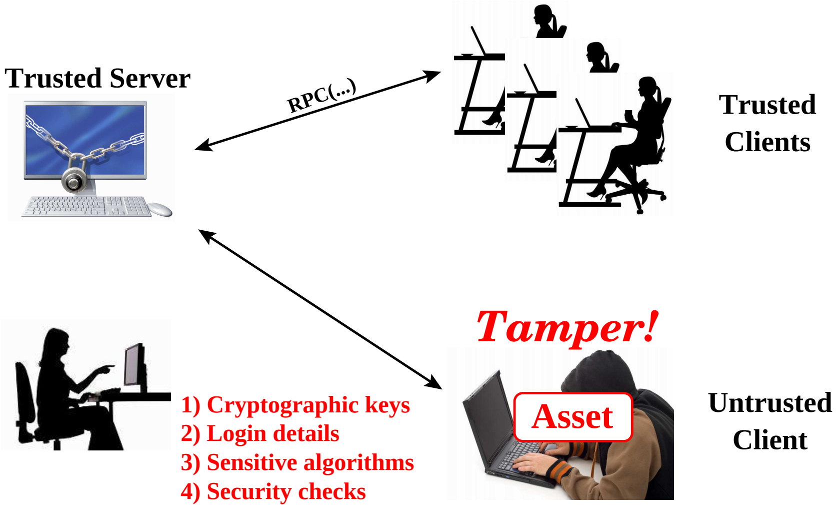
RPC(...)

Trusted Clients

RPC(...)

Untrusted Client

Tigress and the R-MATE Problem



Trusted Server

RPC(...)

Trusted Clients

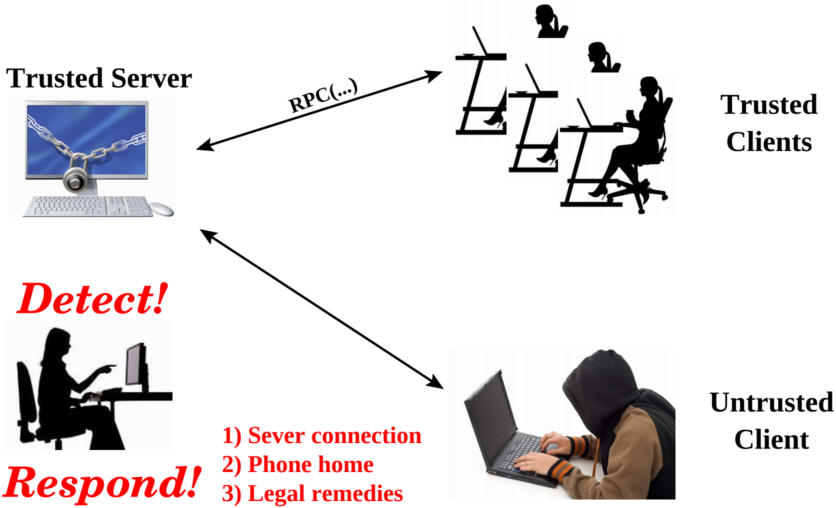
Tamper!

Asset

Untrusted Client

1) Cryptographic keys
2) Login details
3) Sensitive algorithms
4) Security checks

Tigress and the R-MATE Problem



Trusted Server

RPC(...)

Trusted Clients

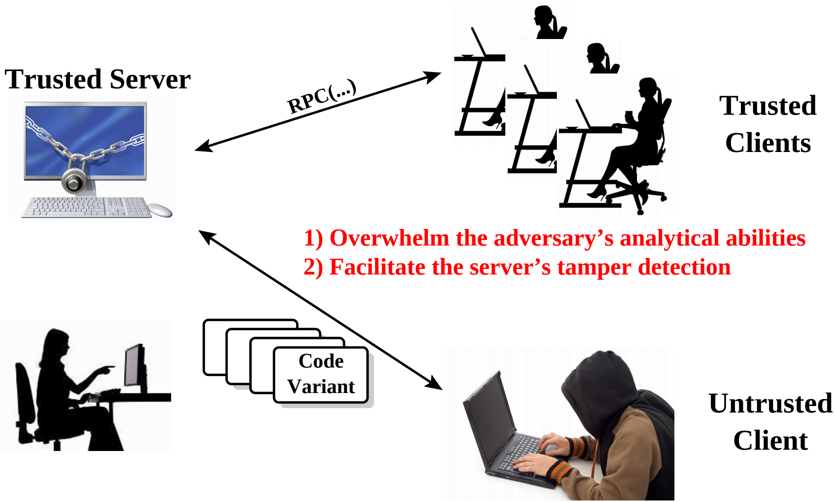
Untrusted Client

Detect!

Respond!

1) Sever connection
2) Phone home
3) Legal remedies

Tigress and the R-MATE Problem



Trusted Server

RPC(...)

Trusted Clients

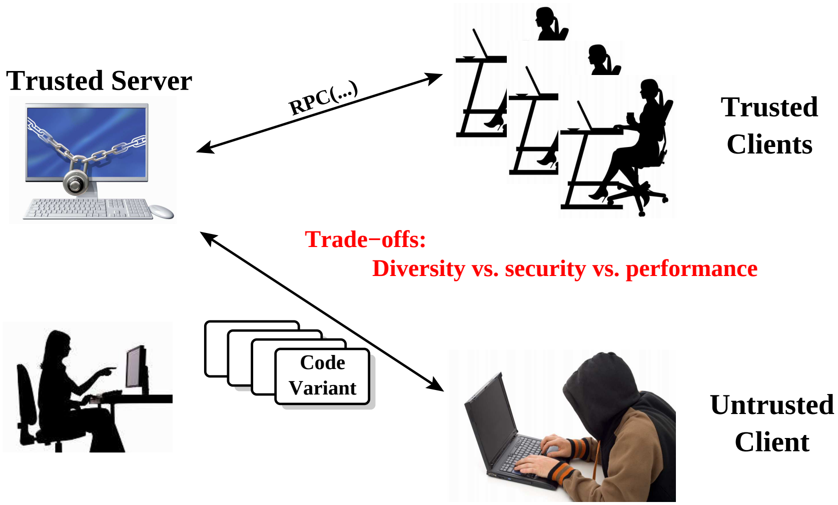
1) Overwhelm the adversary's analytical abilities

2) Facilitate the server's tamper detection

Code Variant

Untrusted Client

Tigress and the R-MATE Problem



Trusted Server

RPC(...)

Trusted Clients

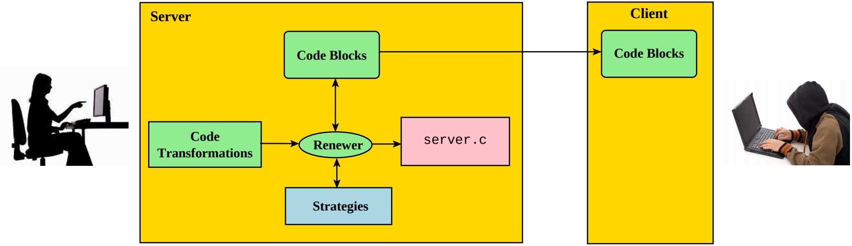
Trade−offs:

Diversity vs. security vs. performance

Code Variant

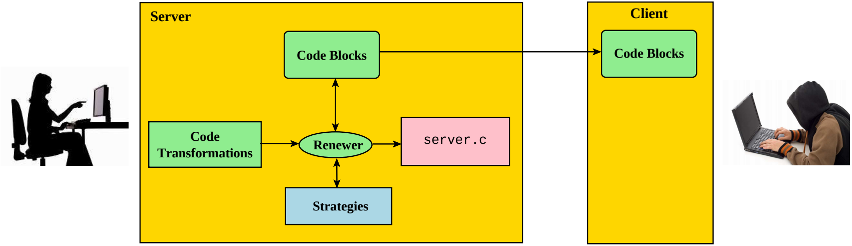
Untrusted Client

The Tigress ***System***



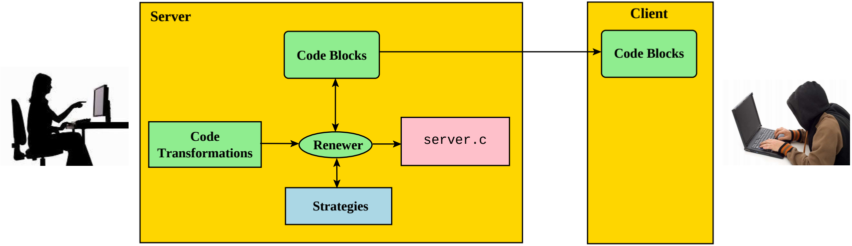
- A fully generalized code diversity system for protecting against R-MATE attacks.

The Tigress ***System***



- A fully generalized code diversity system for protecting against R-MATE attacks.
- The trusted server continuously pushes diverse code variants to the untrusted clients.

The Tigress ***System***



- A fully generalized code diversity system for protecting against R-MATE attacks.
- The trusted server continuously pushes diverse code variants to the untrusted clients.

Attack Model

1. There is no unassailable root-of-trust:
    - the attacker can modify local code/hardware.

Attack Model

1. There is no unassailable root-of-trust:
   - the attacker can modify local code/hardware.
2. The attacker knows the system:
   - primitive code transformations,
   - strategies for combining transformations,
   - the source code of the entire system.

Similar to Kerckhoffs's principles.

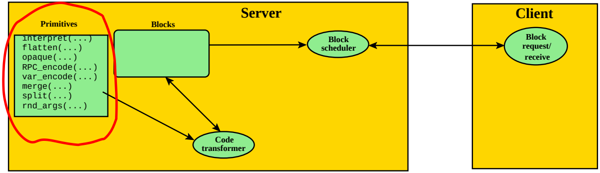
Attack Model

1. There is no unassailable root-of-trust:
   - the attacker can modify local code/hardware.
2. The attacker knows the system:
   - primitive code transformations,
   - strategies for combining transformations,
   - the source code of the entire system.

   Similar to Kerckhoffs's principles.
3. The attacker doesn't know the randomization seed and can't predict the
   - order in which transformations are applied;
   - location in the code where they are applied.

Primitives

Primitive Transformations

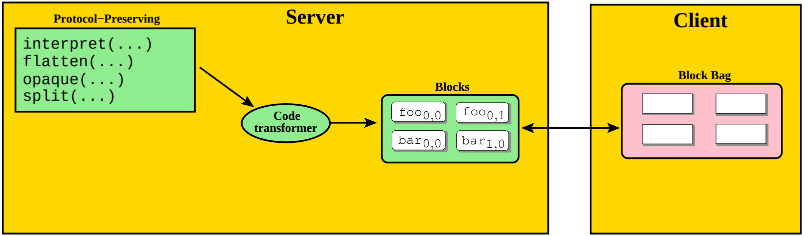


Definition (Primitive)

A primitive is a code transformation that

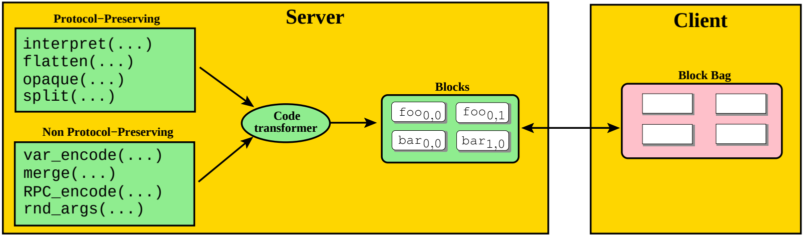
1. adds confusion to the client code, taxing the adversary's analytical abilities (obfuscation);
2. makes modifying client code more difficult (tamperproofing);
3. makes detecting tampering easier (tamper-detect).

Preserving Protocols



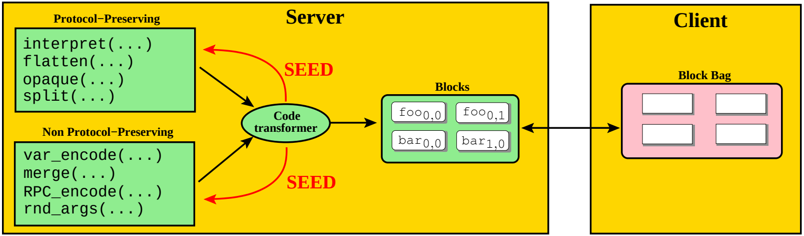
- Protocol-preserving primitives generate confusion and hardening.

Preserving Protocols



- Protocol-preserving primitives generate confusion and hardening.
- Non-protocol-preserving primitives generate incompatible block variants.

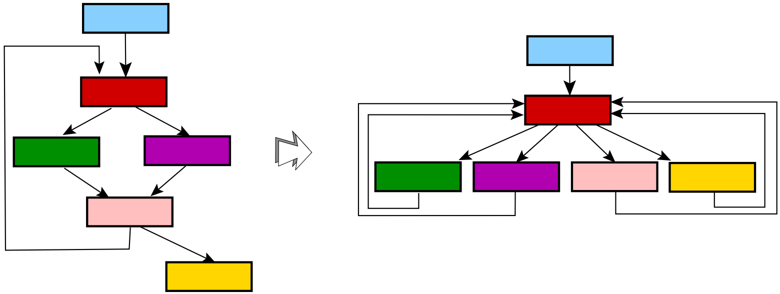
Preserving Protocols



- Protocol-preserving primitives generate confusion and hardening.
- Non-protocol-preserving primitives generate incompatible block variants.
- Randomized primitives generate many unique variants.

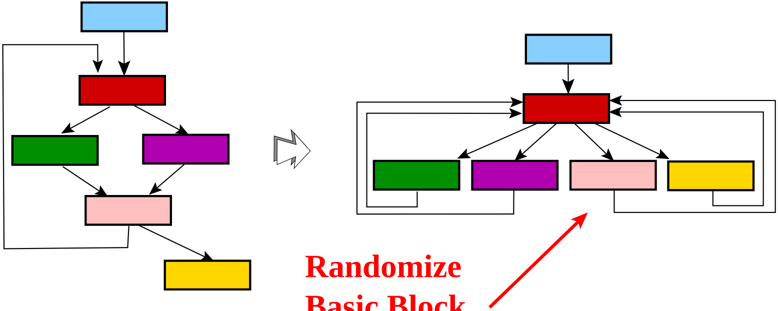
Protocol-Preserving Primitives — Flatten

- **flatten**(f, seed) removes nested control flow.



Protocol-Preserving Primitives — Flatten

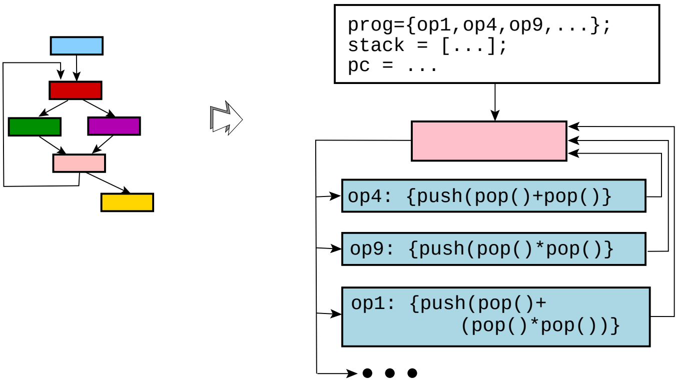
- **flatten**(f, seed) removes nested control flow.



Randomize
Basic Block
Ordering

Protocol-Preserving Primitives — Interpret

- **interpret**(f, seed) turns a function into a specialized interpreter.



```
prog={op1,op4,op9,...};
stack = [...];
pc = ...
```

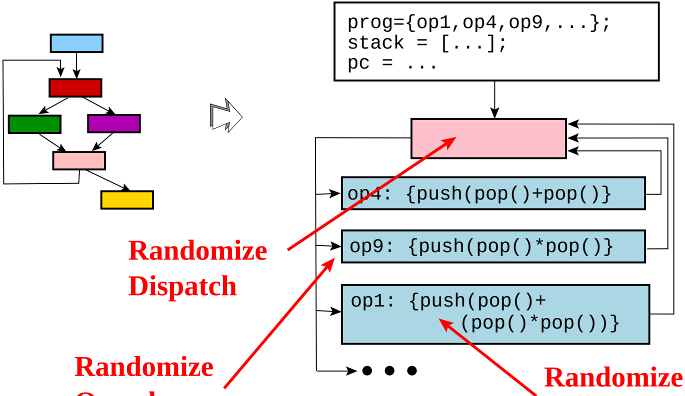
```
op4: {push(pop()+pop()}
```

```
op9: {push(pop()*pop()}
```

```
op1: {push(pop()+
          (pop()*pop())}
```

Protocol-Preserving Primitives — Interpret

- **interpret**(f, seed) turns a function into a specialized interpreter.



```
prog={op1,op4,op9,...};
stack = [...];
pc = ...
```

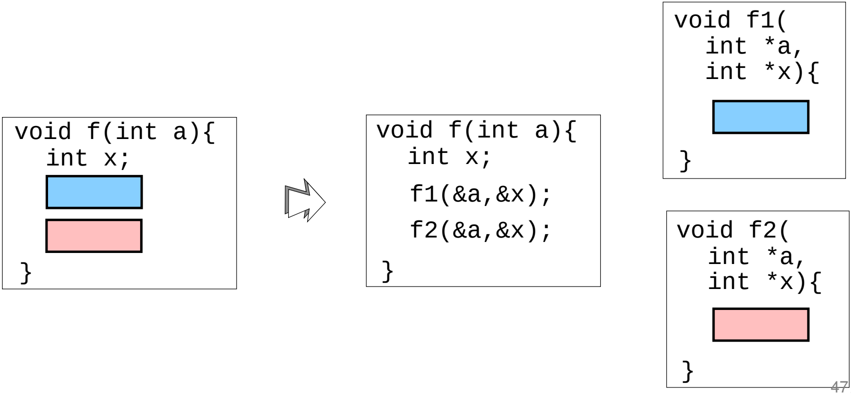
```
op4: {push(pop()+pop()}
```

```
op9: {push(pop()*pop()}
```

```
op1: {push(pop()+
         (pop()*pop())}
```

Protocol-Preserving Primitives — Split

- **split**(f, seed) converts a function *f* into two functions called from *f*:



```
void f(int a){
  int x;
  🟦
  🟥
}
```

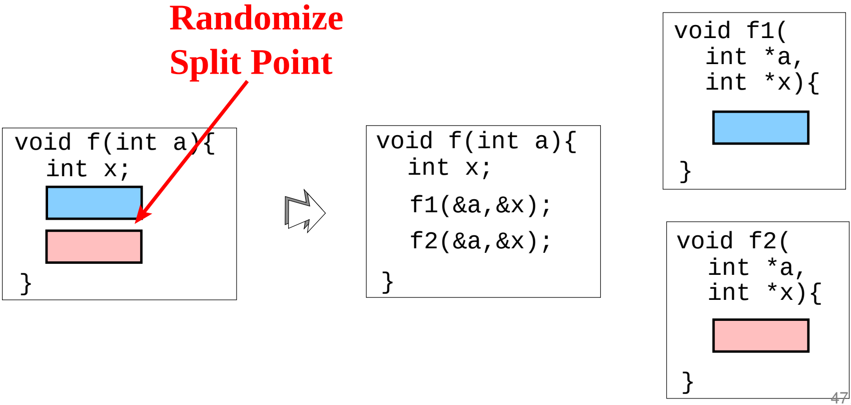
```
void f(int a){
  int x;
  f1(&a,&x);
  f2(&a,&x);
}
```

```
void f1(
  int *a,
  int *x){
  🟦
}
```

```
void f2(
  int *a,
  int *x){
  🟥
}
```

Protocol-Preserving Primitives — Split

- **split**(f, seed) converts a function *f* into two functions called from *f*:



Randomize
Split Point

```
void f(int a){
  int x;
}
```

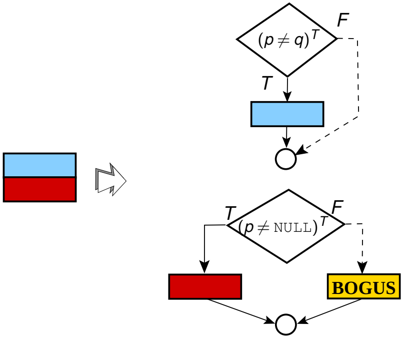
```
void f(int a){
  int x;
  f1(&a,&x);
  f2(&a,&x);
}
```

```
void f1(
  int *a,
  int *x){
}
```

```
void f2(
  int *a,
  int *x){
}
```

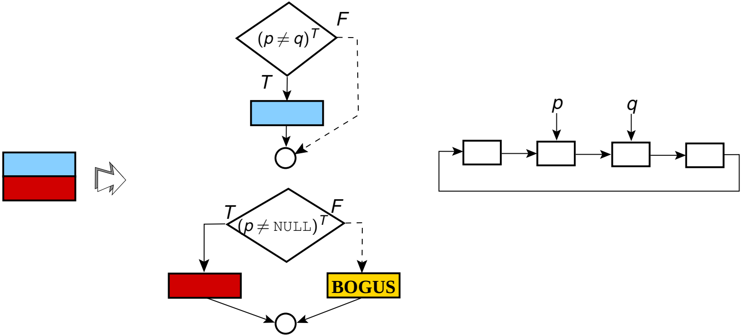
Protocol-Preserving Primitives — Opaque

- **opaque**(f, seed) inserts non-functional code protected by an opaque predicate.



Protocol-Preserving Primitives — Opaque

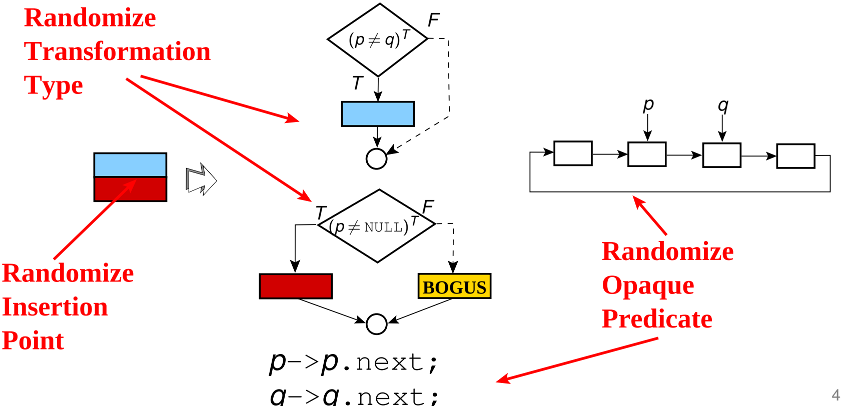
- **opaque**(f, seed) inserts non-functional code protected by an opaque predicate.



```
p->p.next;
q->q.next;
```

Protocol-Preserving Primitives — Opaque

- **opaque**(f, seed) inserts non-functional code protected by an opaque predicate.



Randomize Transformation Type

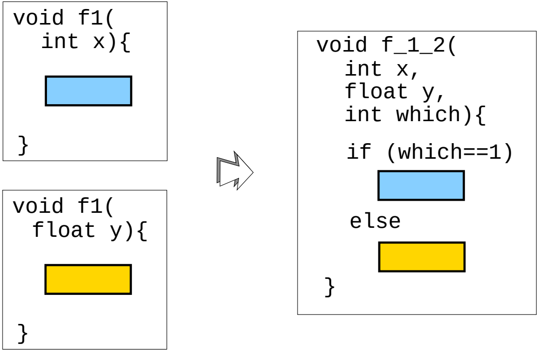
Randomize Insertion Point

Randomize Opaque Predicate

```
p->p.next;
q->q.next;
```

Non-Protocol-Preserving Primitives — Merge

- **merge**$(f_1, f_2, \mathrm{seed})$ combines functions $f_1(\mathrm{args}_1)$ and $f_2(\mathrm{args}_2)$ into $f_{1,2}(\mathrm{args}_1 || \mathrm{args}_2, \mathrm{sel})$.



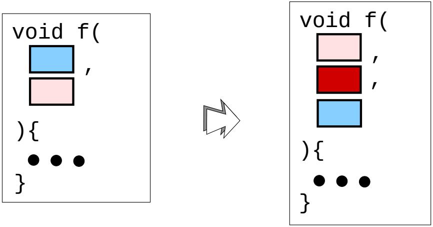
```
void f1(
  int x){

}
```

```
void f1(
 float y){

}
```

```
void f_1_2(
  int x,
  float y,
  int which){
  if (which==1)

  else

}
```

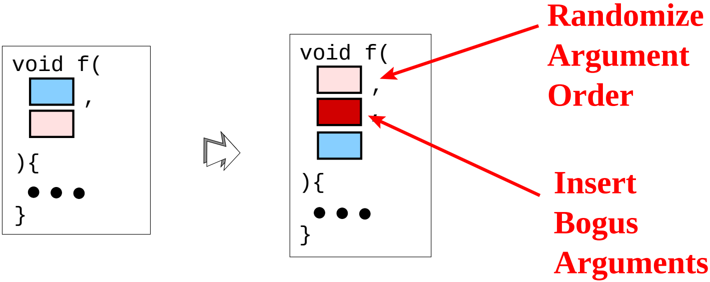
Non-Protocol-Preserving Primitives — rnd_args

- **rnd_args**(f, seed) randomly reorders *f*'s formal parameters and adds extra, bogus, formals.



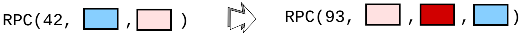
Non-Protocol-Preserving Primitives — rnd_args

- **rnd_args**(f, seed) randomly reorders *f*'s formal parameters and adds extra, bogus, formals.



Non-Protocol-Preserving Primitives — RPC_encode

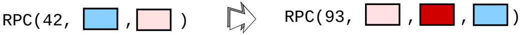
- **RPC_encode**(n, seed) assigns a new random encoding of the *n*:th remote procedure call $\mathtt{RPC}(n, \mathrm{args})$.



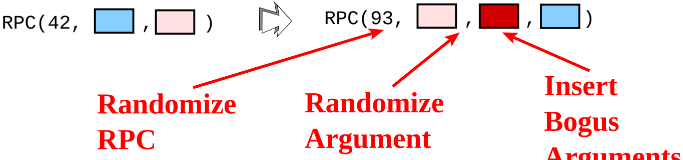
- If the adversary ignores block updates, he may execute an invalid block containing an RPC with an obsolete encoding.
- This alerts the server of the tampering

Non-Protocol-Preserving Primitives — RPC_encode

- **RPC_encode**(n, seed) assigns a new random encoding of the *n*:th remote procedure call $\mathtt{RPC}(n, \mathrm{args})$.



RPC(42, ,) RPC(93, , ,)



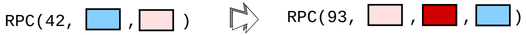
RPC(42, ,) RPC(93, , ,)

Randomize RPC

Randomize Argument

Insert Bogus Arguments

RPC_encode. . .

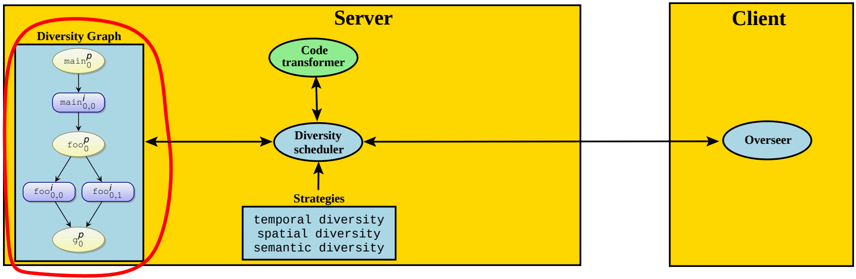


RPC(42, 🟦, ⬜) ⇨ RPC(93, ⬜, 🟥, 🟦)

- If the adversary ignores block updates, he may execute an invalid block containing an RPC with an obsolete encoding.
- This alerts the server of the tampering.

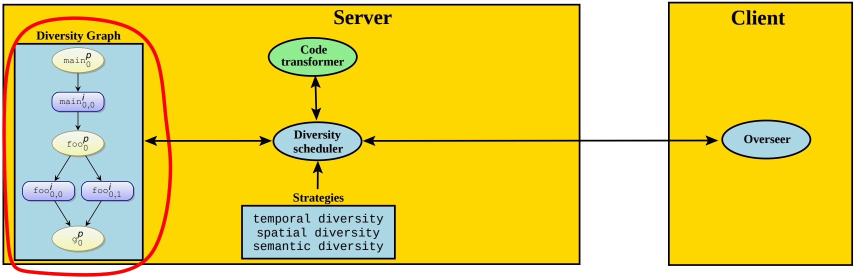
Mechanisms — Strategies

System Overview — Diversity Graph



- The diversity graph represents the complex dependencies between blocks and protocols.

System Overview — Diversity Graph



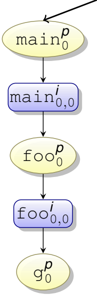
- How does a transformation applied to one block force updates to other blocks?
- Initially, similar to a call graph.

Diversity Graph — Forward Update Cycle

```
int g;
void foo(){
   g++;
}
int main(){
   foo();
}
```

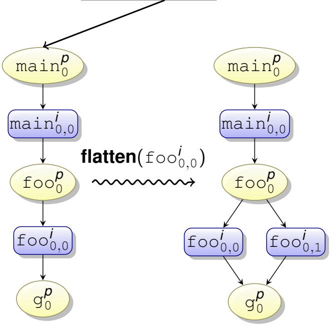
Diversity Graph — Forward Update Cycle

```
int g;
void foo(){
   g++;
}
int main(){
   foo();
}
```



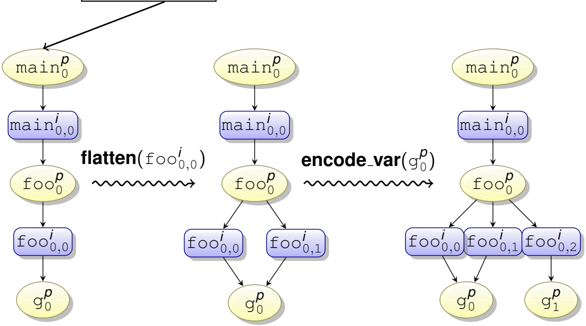
Diversity Graph — Forward Update Cycle

```
int g;
void foo(){
   g++;
}
int main(){
   foo();
}
```



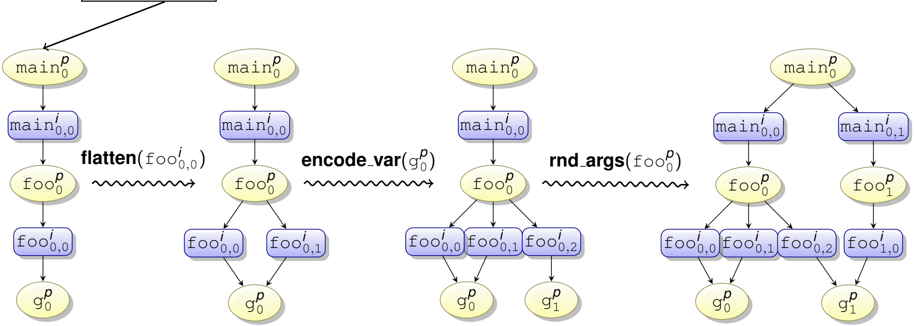
Diversity Graph — Forward Update Cycle

```
int g;
void foo(){
   g++;
}
int main(){
   foo();
}
```

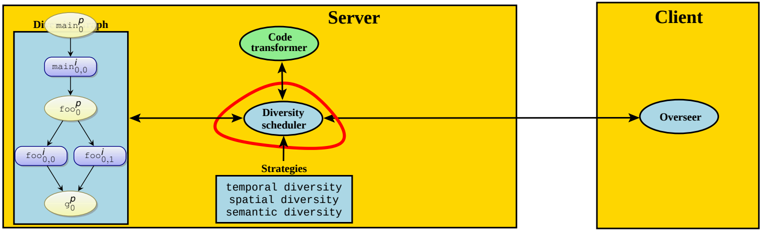


Diversity Graph — Forward Update Cycle

```
int g;
void foo(){
   g++;
}
int main(){
   foo();
}
```



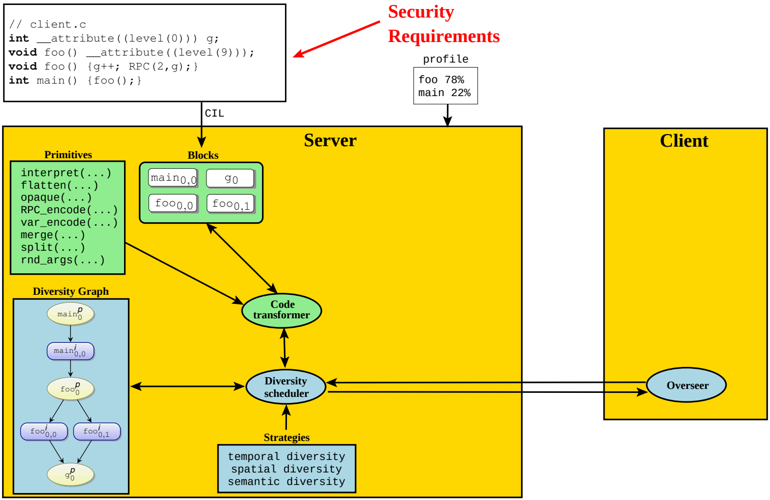
Strategies



temporal diversity
spatial diversity
semantic diversity

- Temporal Diversity: program is continuously renewed.
- Spatial Diversity: *defense-in-depth*, multiple layers of primitives.
- Semantic Diversity: *software aging*, variants are not interchangeable.

Scheduler Operation



```
// client.c
int __attribute((level(0))) g;
void foo() __attribute((level(9)));
void foo() {g++; RPC(2,g);}
int main() {foo();}
```

Security Requirements

profile

```
foo 78%
main 22%
```

CIL

Server

Primitives

```
interpret(...)
flatten(...)
opaque(...)
RPC_encode(...)
var_encode(...)
merge(...)
split(...)
rnd_args(...)
```

Blocks

$\texttt{main}_{0,0}$ $\texttt{g}_0$ $\texttt{foo}_{0,0}$ $\texttt{foo}_{0,1}$

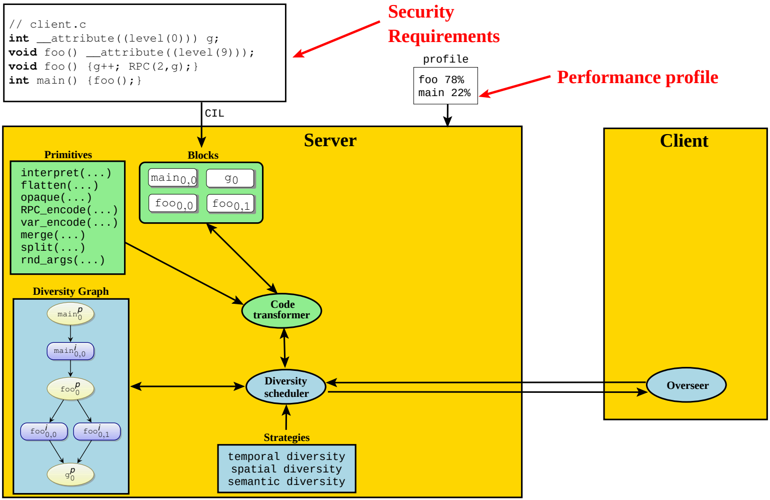
Diversity Graph

Strategies

```
temporal diversity
spatial diversity
semantic diversity
```

Client

Scheduler Operation



```
// client.c
int __attribute((level(0))) g;
void foo() __attribute((level(9)));
void foo() {g++; RPC(2,g);}
int main() {foo();}
```

Security Requirements

profile

```
foo 78%
main 22%
```

Performance profile

CIL

Server

Primitives

```
interpret(...)
flatten(...)
opaque(...)
RPC_encode(...)
var_encode(...)
merge(...)
split(...)
rnd_args(...)
```

Blocks

$\mathtt{main}_{0,0}$ $\mathtt{g}_0$ $\mathtt{foo}_{0,0}$ $\mathtt{foo}_{0,1}$

Diversity Graph

$\mathtt{main}^p_0$ $\mathtt{main}^i_{0,0}$ $\mathtt{foo}^p_0$ $\mathtt{foo}^i_{0,0}$ $\mathtt{foo}^i_{0,1}$ $\mathtt{g}^p_0$

Code transformer

Diversity scheduler

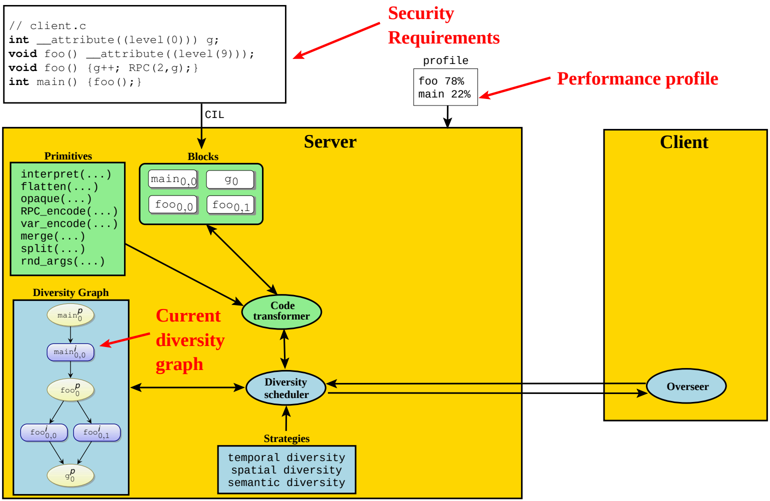
Strategies

```
temporal diversity
spatial diversity
semantic diversity
```

Client

Overseer

Scheduler Operation



```
// client.c
int __attribute((level(0))) g;
void foo() __attribute((level(9)));
void foo() {g++; RPC(2,g);}
int main() {foo();}
```

profile

```
foo 78%
main 22%
```

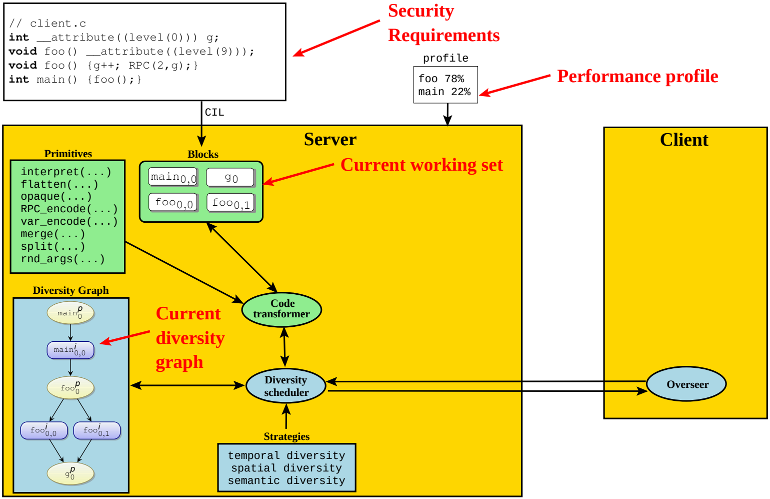
Primitives

```
interpret(...)
flatten(...)
opaque(...)
RPC_encode(...)
var_encode(...)
merge(...)
split(...)
rnd_args(...)
```

Strategies

```
temporal diversity
spatial diversity
semantic diversity
```

Scheduler Operation



```
// client.c
int __attribute((level(0))) g;
void foo() __attribute((level(9)));
void foo() {g++; RPC(2,g);}
int main() {foo();}
```

Security Requirements

profile

```
foo 78%
main 22%
```

Performance profile

Current working set

Current diversity graph

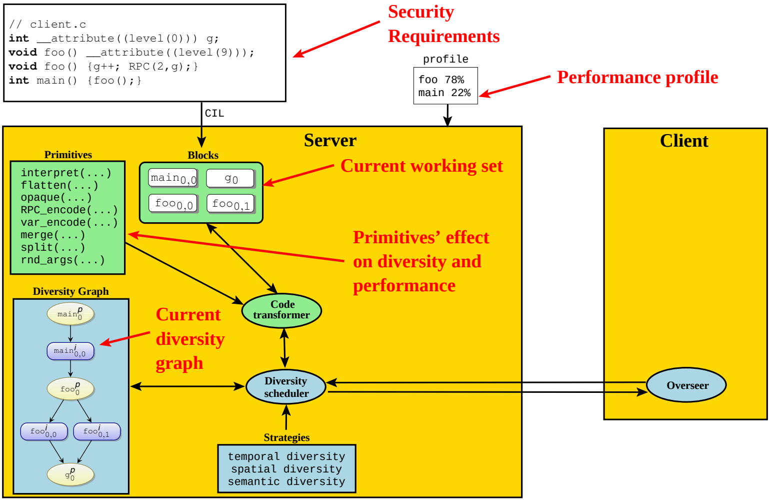
Primitives

```
interpret(...)
flatten(...)
opaque(...)
RPC_encode(...)
var_encode(...)
merge(...)
split(...)
rnd_args(...)
```

Strategies

```
temporal diversity
spatial diversity
semantic diversity
```

Scheduler Operation



```
// client.c
int __attribute((level(0))) g;
void foo() __attribute((level(9)));
void foo() {g++; RPC(2,g);}
int main() {foo();}
```

Security Requirements

profile

```
foo 78%
main 22%
```

Performance profile

Current working set

Primitives’ effect on diversity and performance

Current diversity graph

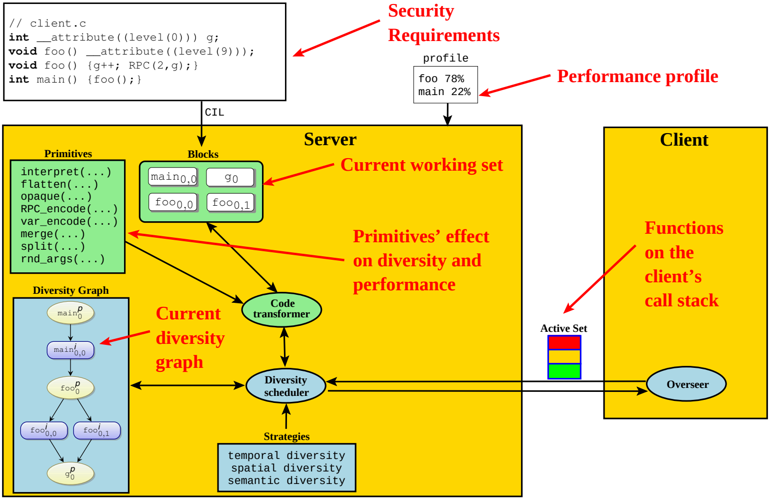
Primitives

```
interpret(...)
flatten(...)
opaque(...)
RPC_encode(...)
var_encode(...)
merge(...)
split(...)
rnd_args(...)
```

Strategies

```
temporal diversity
spatial diversity
semantic diversity
```

Scheduler Operation



```
// client.c
int __attribute((level(0))) g;
void foo() __attribute((level(9)));
void foo() {g++; RPC(2,g);}
int main() {foo();}
```

Security Requirements

profile

```
foo 78%
main 22%
```

Performance profile

CIL

Server

Primitives

```
interpret(...)
flatten(...)
opaque(...)
RPC_encode(...)
var_encode(...)
merge(...)
split(...)
rnd_args(...)
```

Blocks

$\texttt{main}_{0,0}$ $\texttt{g}_{0}$ $\texttt{foo}_{0,0}$ $\texttt{foo}_{0,1}$

Current working set

Primitives' effect on diversity and performance

Diversity Graph

$\texttt{main}^{p}_{0}$ $\texttt{main}^{i}_{0,0}$ $\texttt{foo}^{p}_{0}$ $\texttt{foo}^{i}_{0,0}$ $\texttt{foo}^{i}_{0,1}$ $\texttt{g}^{p}_{0}$

Current diversity graph

Code transformer

Diversity scheduler

Strategies

```
temporal diversity
spatial diversity
semantic diversity
```

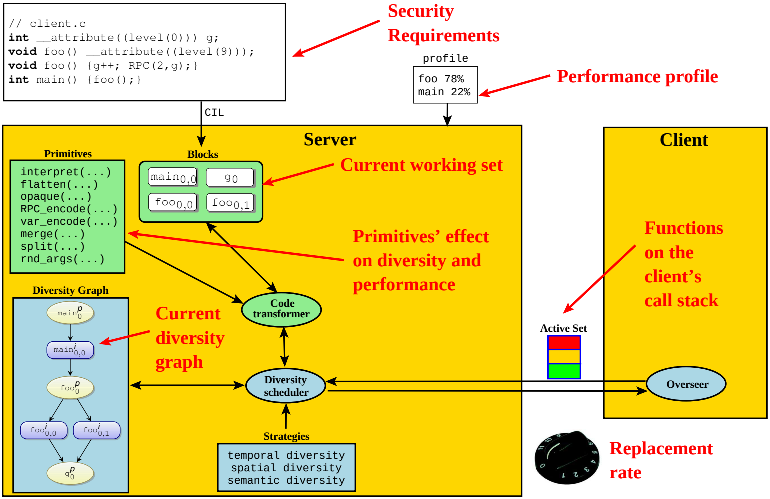
Active Set

Client

Functions on the client's call stack

Overseer

Scheduler Operation



```
// client.c
int __attribute((level(0))) g;
void foo() __attribute((level(9)));
void foo() {g++; RPC(2,g);}
int main() {foo();}
```

profile

```
foo 78%
main 22%
```

Security Requirements

Performance profile

Current working set

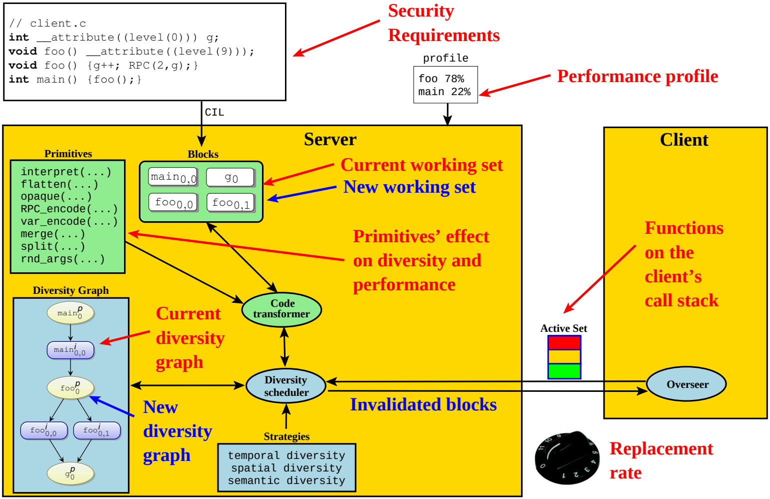
Primitives’ effect on diversity and performance

Current diversity graph

Functions on the client’s call stack

Replacement rate

Scheduler Operation



```
// client.c
int __attribute((level(0))) g;
void foo() __attribute((level(9)));
void foo() {g++; RPC(2,g);}
int main() {foo();}
```

profile

```
foo 78%
main 22%
```

Security Requirements

Performance profile

Current working set

New working set

Primitives' effect on diversity and performance

Current diversity graph

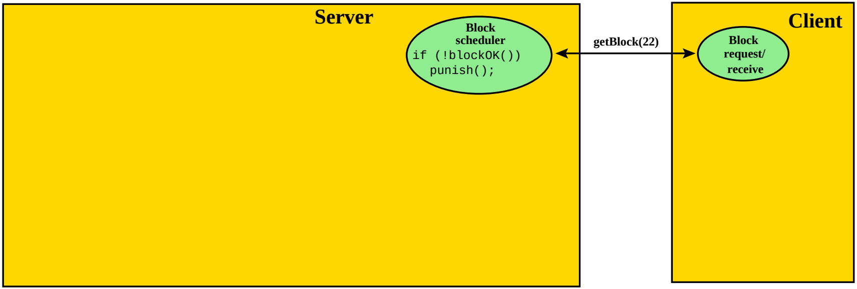
New diversity graph

Invalidated blocks

Functions on the client's call stack

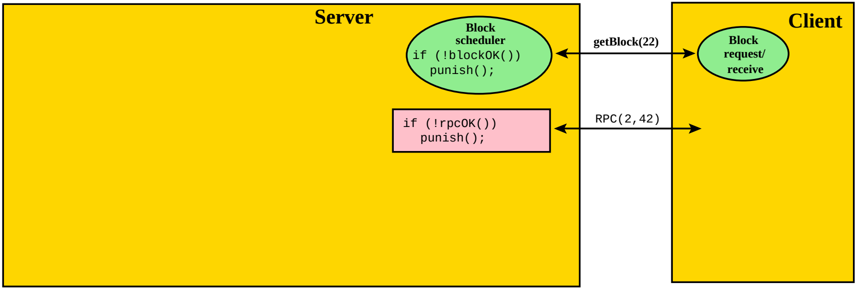
Replacement rate

Crime and Punishment



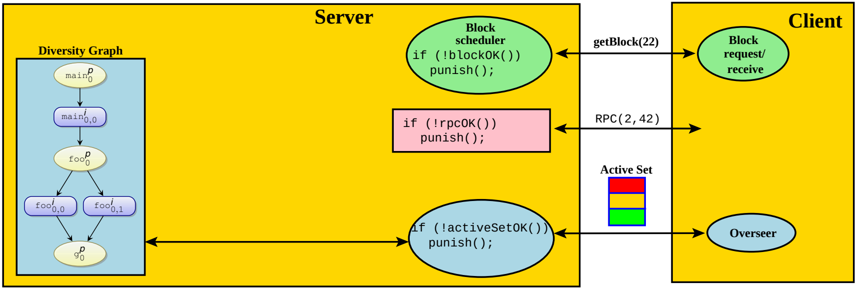
1. Is the requested block part of the current block working set?

Crime and Punishment



1. Is the requested block part of the current block working set?
2. Valid RPC number? Valid RPC argument types?

Crime and Punishment



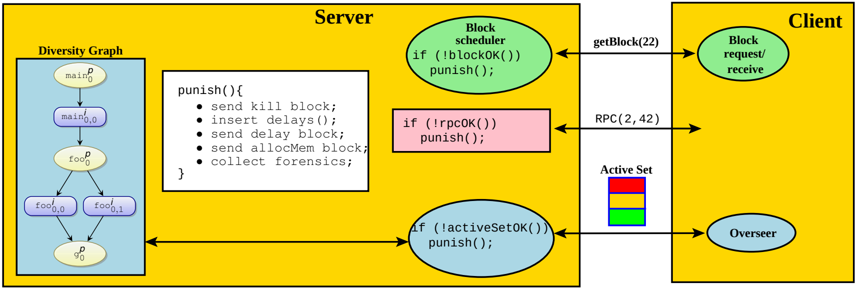
```
if (!blockOK())
  punish();
```

```
if (!rpcOK())
  punish();
```

```
if (!activeSetOK())
  punish();
```

1. Is the requested block part of the current block working set?
2. Valid RPC number? Valid RPC argument types?

Crime and Punishment



```
punish(){
   • send kill block;
   • insert delays();
   • send delay block;
   • send allocMem block;
   • collect forensics;
}
```

```
if (!blockOK())
  punish();
```

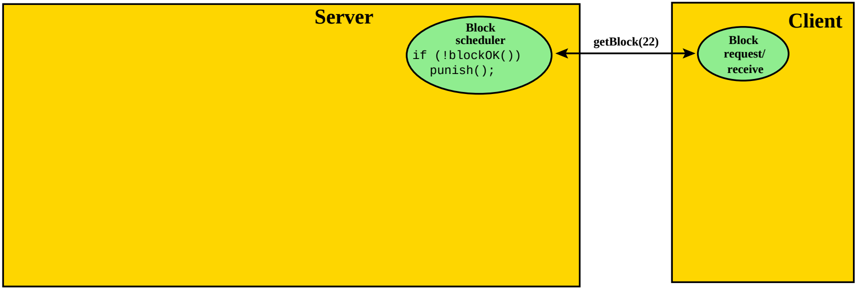
```
if (!rpcOK())
  punish();
```

```
if (!activeSetOK())
  punish();
```

1. Is the requested block part of the current block working set?
2. Valid RPC number? Valid RPC argument types?

Security Evaluation

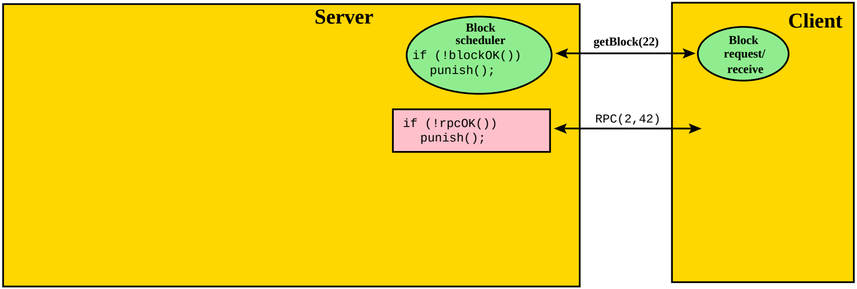
Crime and Punishment



```
if (!blockOK())
   punish();
```

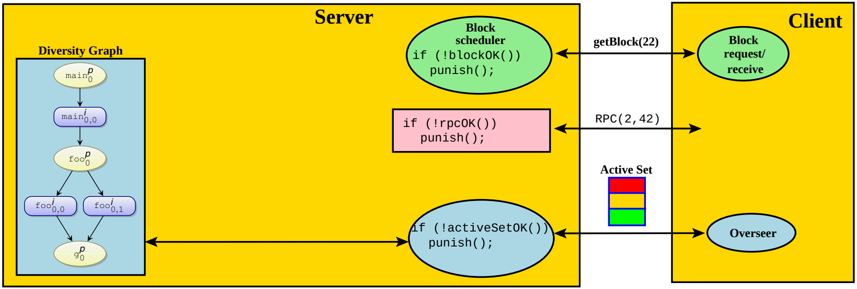
1. Is the requested block part of the current block working set?

Crime and Punishment



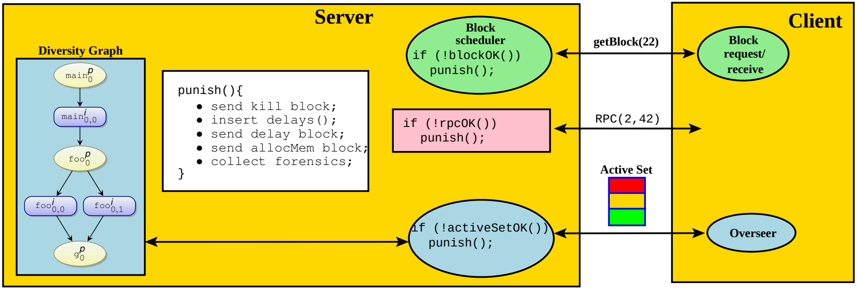
1. Is the requested block part of the current block working set?
2. Valid RPC number? Valid RPC argument types?

Crime and Punishment



1. Is the requested block part of the current block working set?
2. Valid RPC number? Valid RPC argument types?

Crime and Punishment



```
punish(){
   • send kill block;
   • insert delays();
   • send delay block;
   • send allocMem block;
   • collect forensics;
}
```

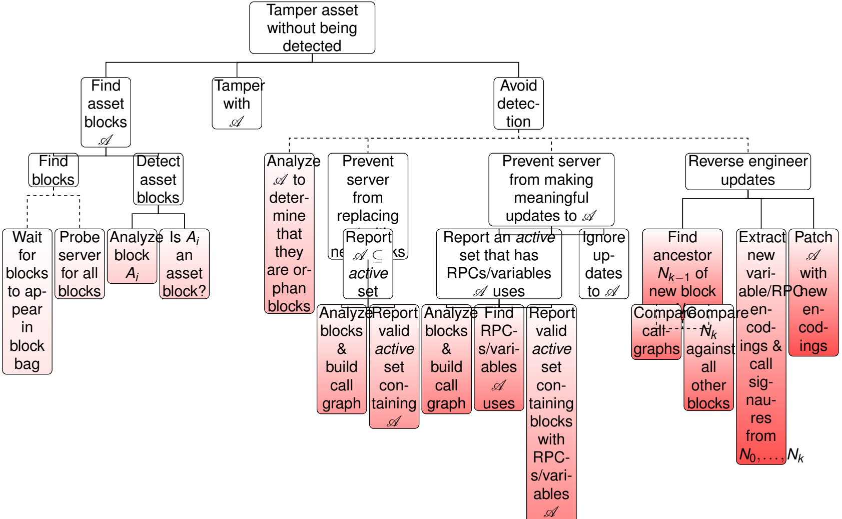
```
if (!blockOK())
   punish();
```

```
if (!rpcOK())
   punish();
```

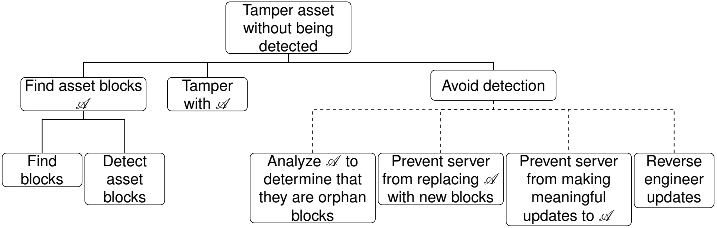
```
if (!activeSetOK())
   punish();
```

1. Is the requested block part of the current block working set?
2. Valid RPC number? Valid RPC argument types?

Enumeration of the Attack Space


Tamper asset without being detected
Find asset blocks $\mathscr{A}$
Tamper with $\mathscr{A}$
Avoid detection
Find blocks
Detect asset blocks
Wait for blocks to appear in block bag
Probe server for all blocks
Analyze block A_i
Is A_i an asset block?
Analyze $\mathscr{A}$ to determine that they are orphan blocks
Prevent server from replacing ne ks
Report $\mathscr{A} \subseteq$ *active* set
Analyze blocks & build call graph
Report valid *active* set containing $\mathscr{A}$
Prevent server from making meaningful updates to $\mathscr{A}$
Report an *active* set that has RPCs/variables $\mathscr{A}$ uses
Ignore updates to $\mathscr{A}$
Analyze blocks & build call graph
Find RPCs/variables $\mathscr{A}$ uses
Report valid *active* set containing blocks with RPCs/variables $\mathscr{A}$
Reverse engineer updates
Find ancestor N_{k-1} of new block
Compare call-graphs
Compare N_k against all other blocks
Extract new variable/RPC encodings & call signaures from $N_0, \ldots, N_k$
Patch $\mathscr{A}$ with new encodings

Enumeration of the Attack Space



Tamper asset without being detected

Find asset blocks $\mathscr{A}$

Tamper with $\mathscr{A}$

Avoid detection

Find blocks

Detect asset blocks

Analyze $\mathscr{A}$ to determine that they are orphan blocks

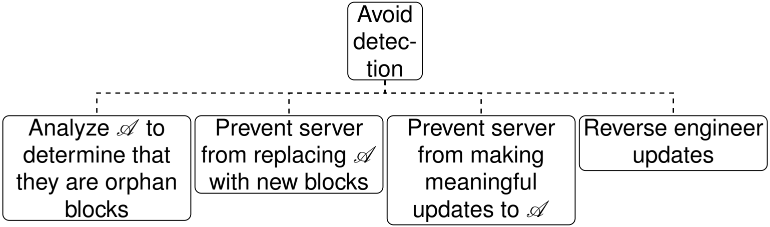
Prevent server from replacing $\mathscr{A}$ with new blocks

Prevent server from making meaningful updates to $\mathscr{A}$

Reverse engineer updates

- Root represents the *asset* in the client code (security check, code that updates a global variable, the integrity of a control-flow path, global data, . . .).
- Attack steps:
  1. find the *asset blocks*
  2. tamper with these blocks

The Attack Space — Avoiding Detection



Avoid detection

Analyze $\mathscr{A}$ to determine that they are orphan blocks

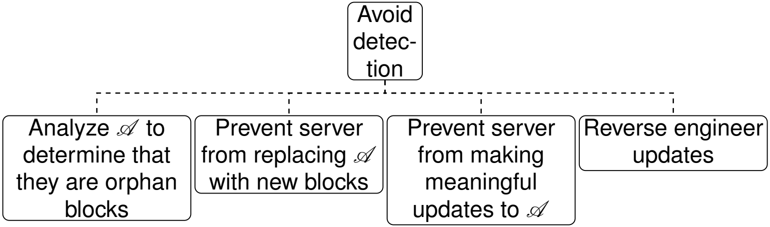
Prevent server from replacing $\mathscr{A}$ with new blocks

Prevent server from making meaningful updates to $\mathscr{A}$

Reverse engineer updates

1. Orphan blocks (no calls, RPCs): modify at will!

The Attack Space — Avoiding Detection



Avoid detection

Analyze $\mathscr{A}$ to determine that they are orphan blocks

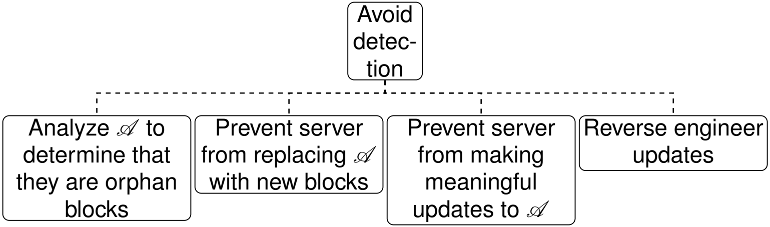
Prevent server from replacing $\mathscr{A}$ with new blocks

Prevent server from making meaningful updates to $\mathscr{A}$

Reverse engineer updates

1. Orphan blocks (no calls, RPCs): modify at will!
2. Trick the server that asset blocks are all active: server can't update!

The Attack Space — Avoiding Detection



Avoid detection

Analyze $\mathscr{A}$ to determine that they are orphan blocks

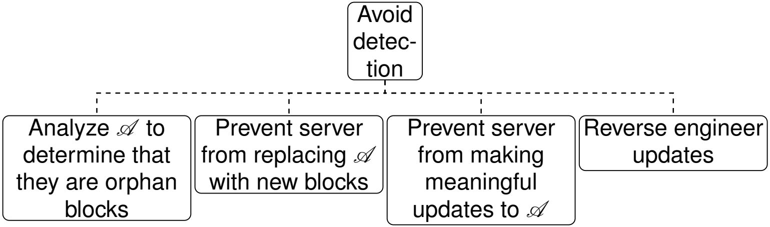
Prevent server from replacing $\mathscr{A}$ with new blocks

Prevent server from making meaningful updates to $\mathscr{A}$

Reverse engineer updates

1. Orphan blocks (no calls, RPCs): modify at will!
2. Trick the server that asset blocks are all active: server can't update!
3. Trick the server to only make trivial changes to asset blocks: ignore updates!

The Attack Space — Avoiding Detection



Avoid detection

Analyze $\mathscr{A}$ to determine that they are orphan blocks

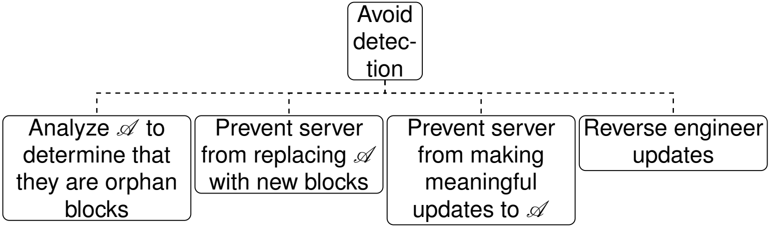
Prevent server from replacing $\mathscr{A}$ with new blocks

Prevent server from making meaningful updates to $\mathscr{A}$

Reverse engineer updates

1. Orphan blocks (no calls, RPCs): modify at will!
2. Trick the server that asset blocks are all active: server can't update!
3. Trick the server to only make trivial changes to asset blocks: ignore updates!
4. Reverse engineer/patch new variants on

The Attack Space — Countermeasures



Avoid detection

Analyze $\mathscr{A}$ to determine that they are orphan blocks

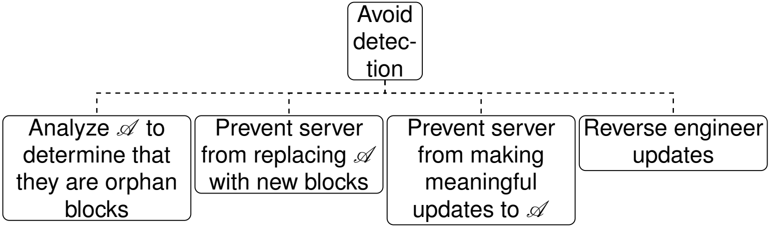
Prevent server from replacing $\mathscr{A}$ with new blocks

Prevent server from making meaningful updates to $\mathscr{A}$

Reverse engineer updates

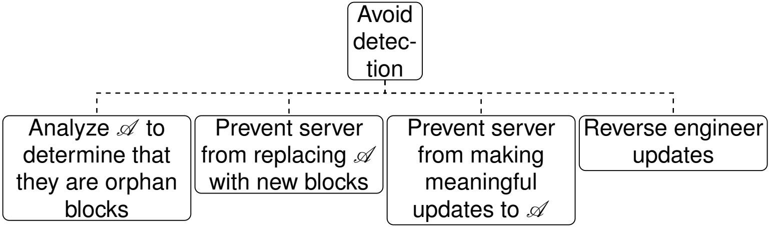
1. Slow down reverse engineering using protocol-preserving primitives.

The Attack Space — Countermeasures


Avoid detection
Analyze $\mathscr{A}$ to determine that they are orphan blocks
Prevent server from replacing $\mathscr{A}$ with new blocks
Prevent server from making meaningful updates to $\mathscr{A}$
Reverse engineer updates

1. Slow down reverse engineering using protocol-preserving primitives.
2. Use **opaque** to connect orphan blocks to the rest of the program.

The Attack Space — Countermeasures



Avoid detection

Analyze $\mathscr{A}$ to determine that they are orphan blocks

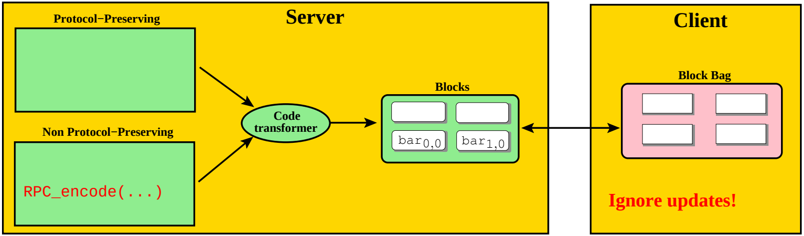
Prevent server from replacing $\mathscr{A}$ with new blocks

Prevent server from making meaningful updates to $\mathscr{A}$

Reverse engineer updates

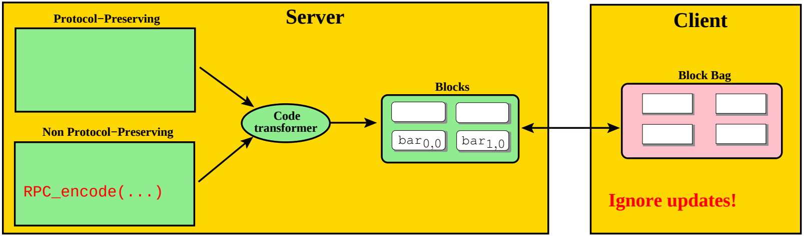
1. Slow down reverse engineering using protocol-preserving primitives.
2. Use **opaque** to connect orphan blocks to the rest of the program.
3. Use **opaque** primitive to insert calls to non-existing functions. If the adversary reports an *active* set containing such a

Security Evaluation — Empirical Test #1



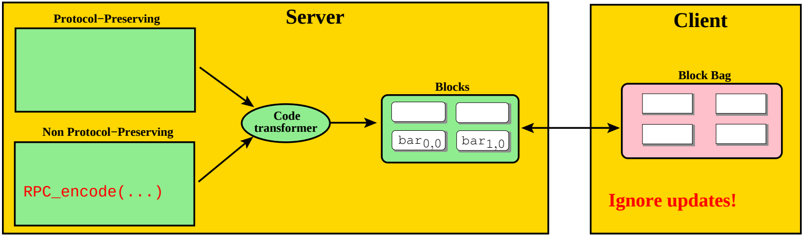
- Attack: Ignore block updates!

Security Evaluation — Empirical Test #1



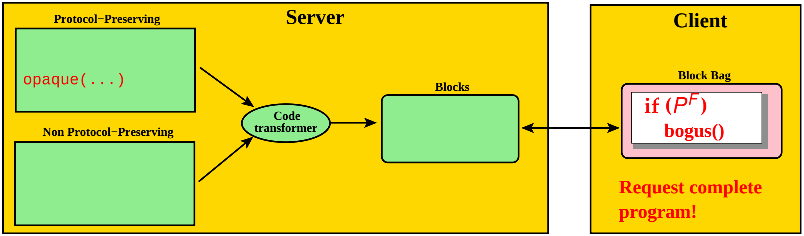
- Attack: Ignore block updates!
- Simulated Test: Turn off client updates.

Security Evaluation — Empirical Test #1



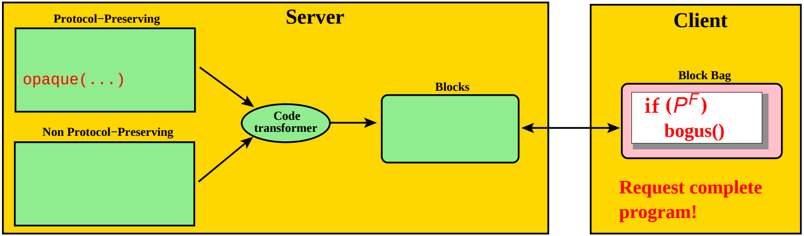
- Attack: Ignore block updates!
- Simulated Test: Turn off client updates.
- Result: RPCs are frequent in our test program, the server reliably detected the malicious behavior shortly after the first **RPC_encode** update.

Security Evaluation — Empirical Test #2



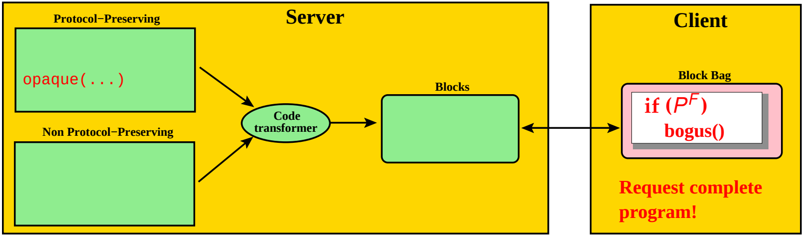
- Attack: Build a snapshot of the entire program, in order to analyze it off-line!

Security Evaluation — Empirical Test #2



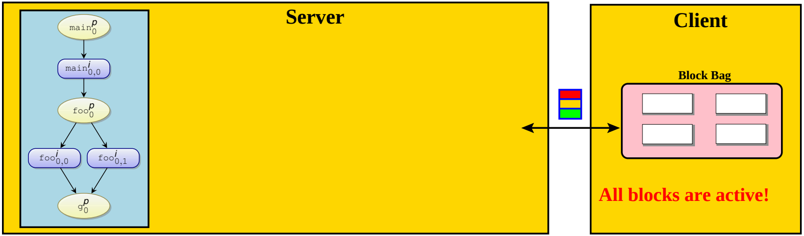
- Attack: Build a snapshot of the entire program, in order to analyze it off-line!
- Simulated Test: Client disassembles its blocks, requests referenced blocks.

Security Evaluation — Empirical Test #2



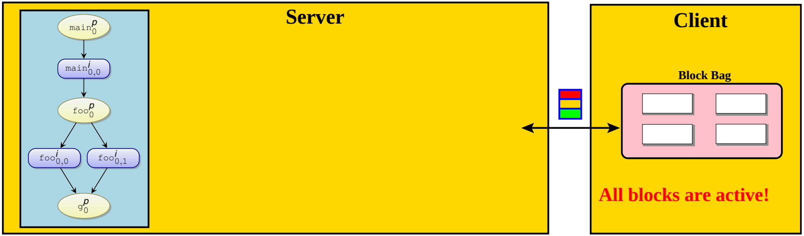
- Attack: Build a snapshot of the entire program, in order to analyze it off-line!
- Simulated Test: Client disassembles its blocks, requests referenced blocks.
- Result: The malicious client quickly requested nonexistent blocks.

Security Evaluation — Empirical Test #3



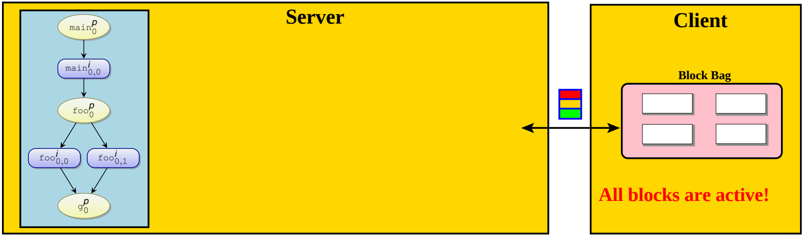
- Attack: Prevent the server from updating blocks!

Security Evaluation — Empirical Test #3



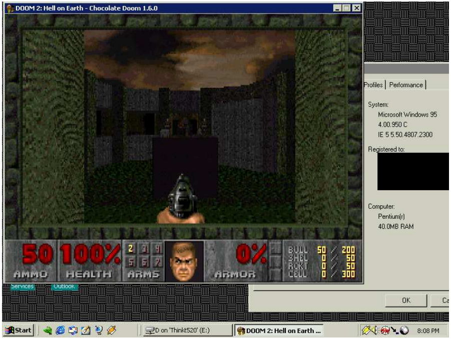
- Attack: Prevent the server from updating blocks!
- Simulated Test: Client reports the entire contents of the block bag as the *active* set.

Security Evaluation — Empirical Test #3



- Attack: Prevent the server from updating blocks!
- Simulated Test: Client reports the entire contents of the block bag as the *active* set.
- Result: Using the program call graph the server reliably identified the malicious

Security Evaluation — Empirical Test #4



DOOM 2: Hell on Earth - Chocolate Doom 1.6.0

System:
Microsoft Windows 95
4.00.950 C
IE 5 5.50.4807.2300

Registered to:
[redacted]

Computer:
Pentium(r)
40.0MB RAM

- We’re porting ChocolateDoom to Tigress.
- Capture-the-Flag exercises!
- To appear. . .

Discussion

Summary

- A system for detecting tampering of clients running on untrusted nodes in a distributed system.

Summary

- A system for detecting tampering of clients running on untrusted nodes in a distributed system.
- Assume the adversary has complete knowledge of our system
  - no security-through-obscurity

Summary — Security

- Protocol-preserving primitives:
  - Gives attacker limited time-window for analysis/tampering.

Summary — Security

- Protocol-preserving primitives:
  - Gives attacker limited time-window for analysis/tampering.
- Non-protocol-preserving primitives:
  - Harder to tamper without modifying expected behavior ⇒ easier tamper-detection.

Summary — Security

- Protocol-preserving primitives:
  - Gives attacker limited time-window for analysis/tampering.
- Non-protocol-preserving primitives:
  - Harder to tamper without modifying expected behavior ⇒ easier tamper-detection.
- Security:
  - Function of the frequency of code updates and the complexity and variability generated by primitives.

Summary — Performance

- Highly tunable:
    - Control which parts of the program to transform, which transformations to apply, update frequency.

Summary — Performance

- Highly tunable:
  - Control which parts of the program to transform, which transformations to apply, update frequency.
- Performance overhead:
  - Infrastructure: 4% to 23%.
  - Update delay: 2 to 3 seconds (protocol-preserving primitives), 7 to 24 seconds (non-protocol-preserving primitives).

Discussion

- Optimize differently for different scenarios:
  - Client performance
  - Server performance
  - Network latency/bandwidth
  - Client energy use
  - Time-to-crack

Discussion

- Optimize differently for different scenarios:
  - Client performance
  - Server performance
  - Network latency/bandwidth
  - Client energy use
  - Time-to-crack
- What about different network topologies?
  - client-server
  - 1 server + *n* untrusted clients running same code?
  - 1 server + *n* untrusted clients running different code?
  - 1 server + *n* trusted clients + *m* untrusted clients?