Software Tampering

What is tamper proofing?

✓Ensure that programs executes as intended, even in the presence of an adversary who tries to disrupt, monitor, or change the execution

✓A tamper-proofing algorithm:

- makes tampering difficult (obfuscation)
- *detects* when tampering has occurred
- *responds* to the attack

Goal

✓Prevent an adversary from removing license checking code from your program

```
if (expired()) {
   printf("pay me!")
   abort()
}
```

```
if (false) {
   printf("pay me!")
   abort()
}
```

```
if (expired()) {



}
```

✓Prevent an adversary from adding missing functionality (print, save)

Attack Scenario, Problem setting

- ✓Examples of how software developers try to ensure a financial gain:
  - Adobe's free PDF reader lets you fill in a form and print it but not save it
  - Some evaluation product don't allow you to print
  - Games don't provide you with an infinite supply of ammunition
  - Evaluation copies stop working after a certain period of time
  - Voice-over-IP-clients charge you money to make phone calls
  - DRM media players and TV-set-top boxes charge you to watch movies or listen to music

Attack Scenario: How

- ✓Modify P's executable files prior to execution
- ✓Force a modified operating system to be loaded
- ✓Modify the dynamic linker
- ✓Replace the dynamic libraries
- ✓Run P under emulation
- ✓Modify P while running under a debugger

Check Scenario: What

✓Ensure that P's executable files are healthy and that the environment in which it is running isn't hostile in any way

- Unadulterated hardware and operating system
- Unmodified P's code
- Not running under emulation
- Not being modified by a debugger
- The right dynamic libraries have been loaded

Tamper-proofing techniques

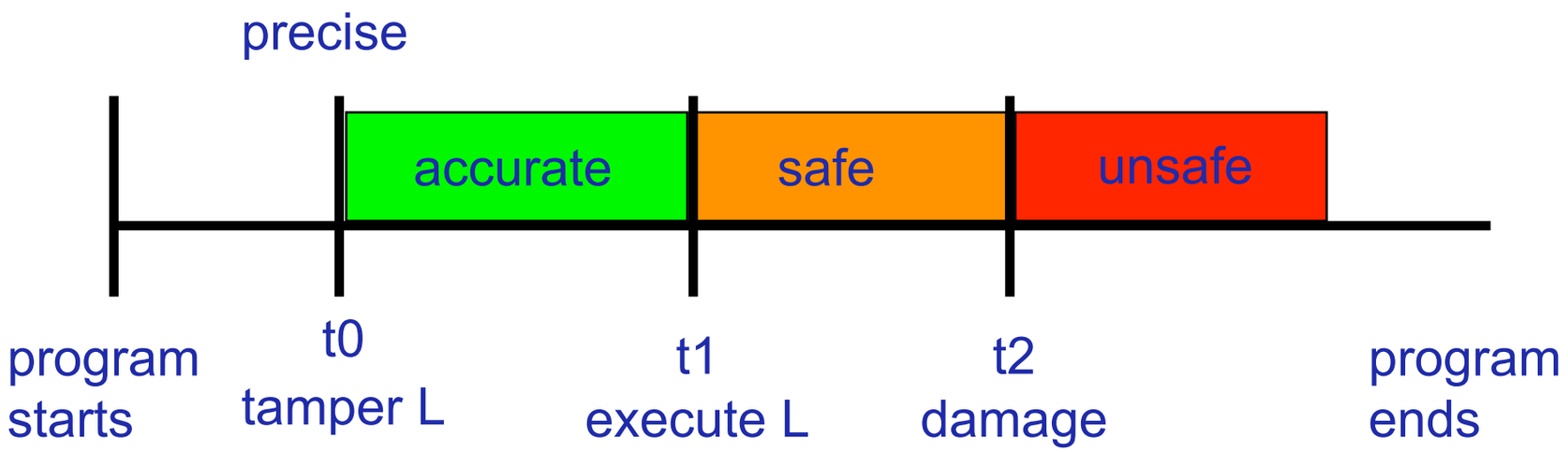
- ✓Consists of two functions
  - ✓CHECK: that monitors the health of the system by testing a set of invariants and returning **true** if nothing suspicious is found
  - ✓RESPOND: queries CHECK to see if P is running as expected, and if it's not, issues a *tamper response* (e.g. terminating the program)

Checking for tampering

✓CHECK tests a set of invariants

- Code checking: check that P's code hashes to a known value
- Result checking: test that the result of computation is correct. Checking the validity of the computation result is often computationally cheaper than performing the computation itself
- Environment checking: the hardest thing for a program to check is the validity of its execution environment.
  - Running under an emulator
  - Operating system
  - …

Precision



- ✓ Precise: the checker detects the attack immediately after the modification has taken place
- ✓ Accurate: the checker detects the attacks before the modified code is executed
- ✓ Safe: the checker waits until after the modified code has been executed, but before the first interaction event (before damage)
- ✓ Unsafe: the checker identifies the attack after the damage has taken place

Respond Scenario: How

1. Terminate the program.
2. Restore the program to its correct state, by patching the tampered code.
3. Deliberately return incorrect results, maybe deteriorate slowly over time.
4. Degrade the performance of the program.
5. Report the attack for example by “phoning home”.
6. Punish the attacker by destroying the program or objects in its environment:
   - *DisplayEater* deletes your home directory.
   - Destroy the computer by repeatedly flashing the bootloader flash memory.

Introspection

Checking by Introspection

Augment the program with functions that compute a hash over a code region to compare to an expected value

```
.........
start = start_address;
end   = end_address;
h = 0;
while (start < end) {
   h = h ⊕ *start;
   start++;
}
if (h != expected_value)
   abort();
goto *h;
.........
```

Attacks to these techniques are based on pattern matching:

- static: search for suspicious patterns in the code, like the initialization section (easy when the loop bounds are not obfuscated)
- dynamic: reads into the code segment (unusual in typical programs)

Checking by Introspection

- Checkers compute a hash over a region and compare to the expected value.
- Checkers check the code and check each other as well!
- Build up a network of code regions: blocks of user code, checkers, and responders.
- When a tampered function is found repair it!
- Skype uses a similar technique.

```
int main (int argc, char *argv[]) {


    A();
}

int A() {

    B();
}

int B() {
    ...
}
```

```
uint32 B_COPY[]={0x83e58955,0xaeb820ec,0xc7080486,...};

int main (int argc, char *argv[]) {


    A();
}

int A() {
    B_hash = hash(B);
    if (B_hash != 0x4f4205a5)
        memcpy(B,B_COPY);
    B();
}

int B() {
    ...
}
```

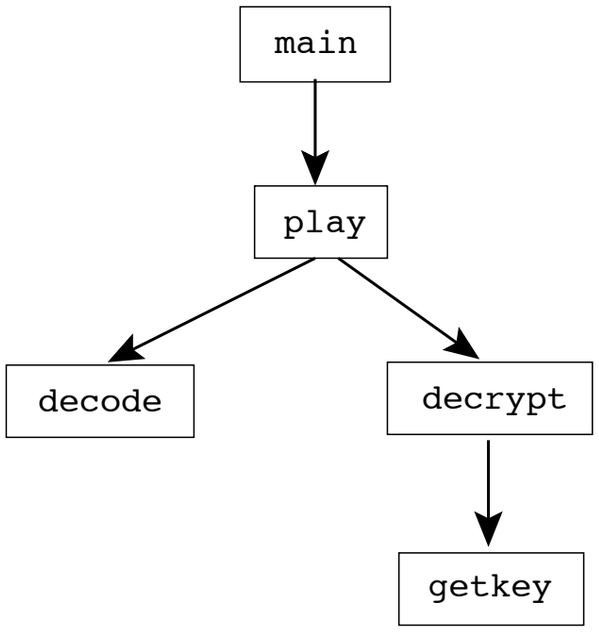
Protecting B

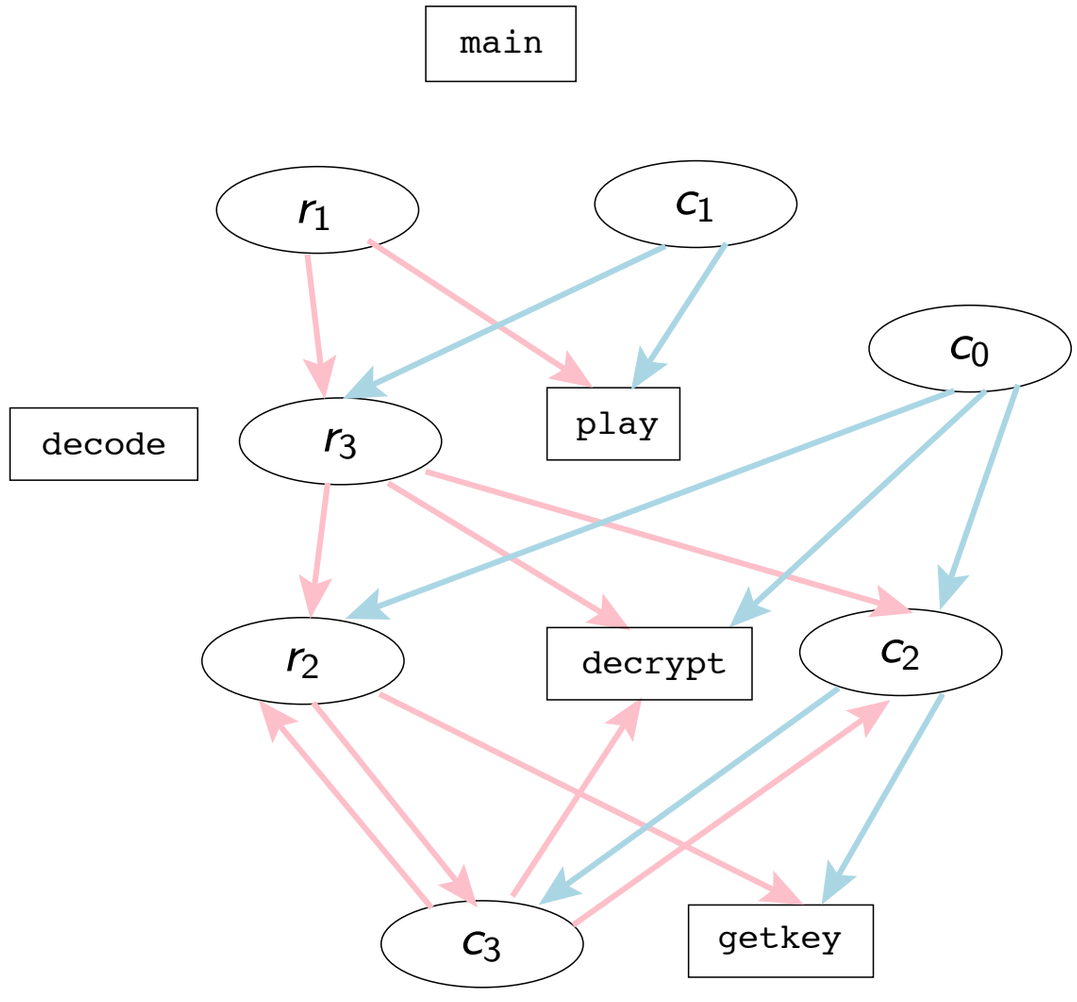
```
uint32 A_COPY[] ={0x83e58955,0x72b820ec,0xc7080486,...};
uint32 B_COPY[]={0x83e58955,0xaeb820ec,0xc7080486,...};

int main (int argc, char *argv[]) {
    A_hash = hash(A);
    if (A_hash != 0x105AB23F)
        memcpy(A,A_COPY);
    A();
}

int A() {
    B_hash = hash(B);
    if (B_hash != 0x4f4205a5)
        memcpy(B,B_COPY);
    B();
}

int B() {
    ...
}
```

Protecting A and B

Idea: build a graph of protect/act on P

The input to the protection tool is the guard graph. It shows the relationship between regions to be protected, and the checkers and responders that check them.





Protecting software code by guards. Chang and Atallah DRM Workshop 2001

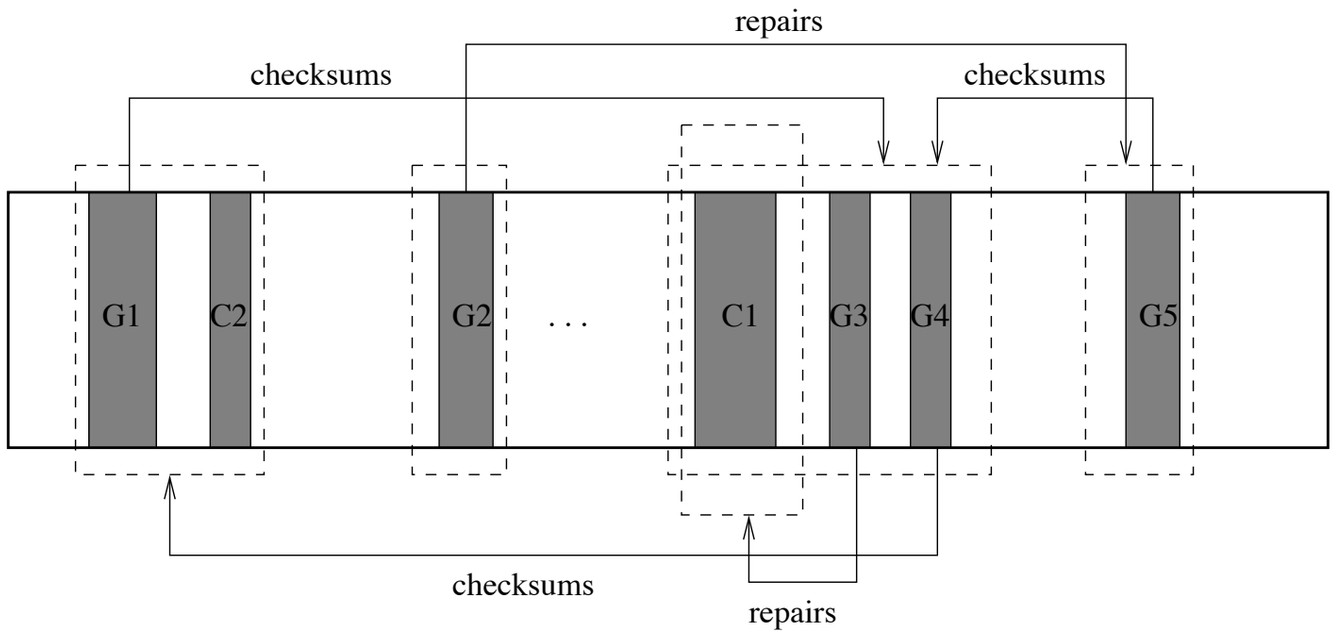
Program: C1;C2

Tamperproof(P,G):

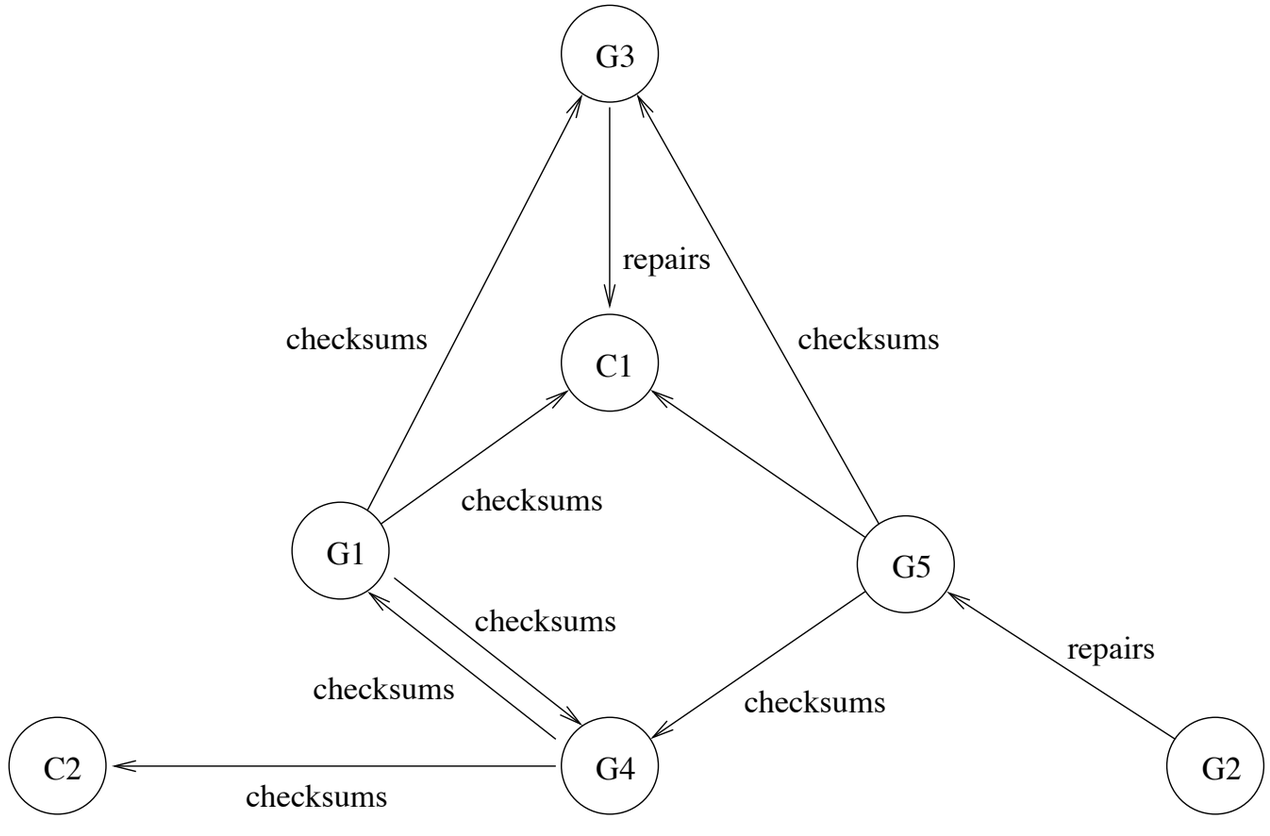
1. Let P = C1;...;Cn
2. Let G = (V,E) with

   nodes: code/guard

   edges: guard ⟶ node
3. insert respond in P such that they dominante the checked region
4. insert check in P such that at least one check dominates every respond

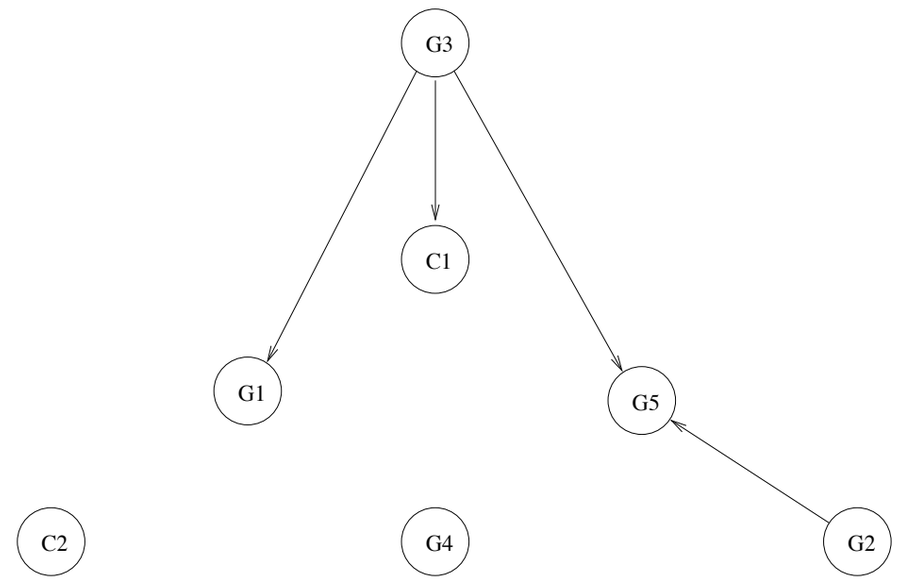


(a) Memory layout of the guarded program

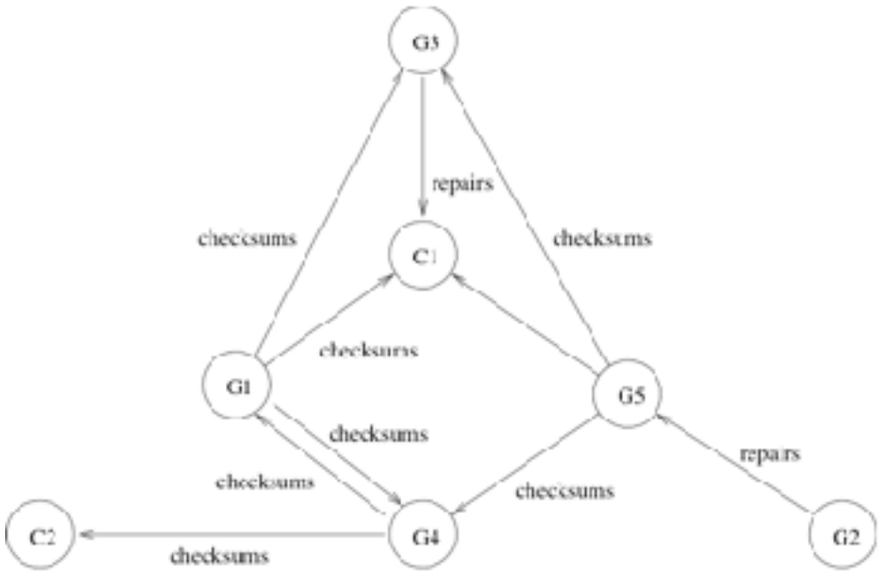


(b) The corresponding guard graph

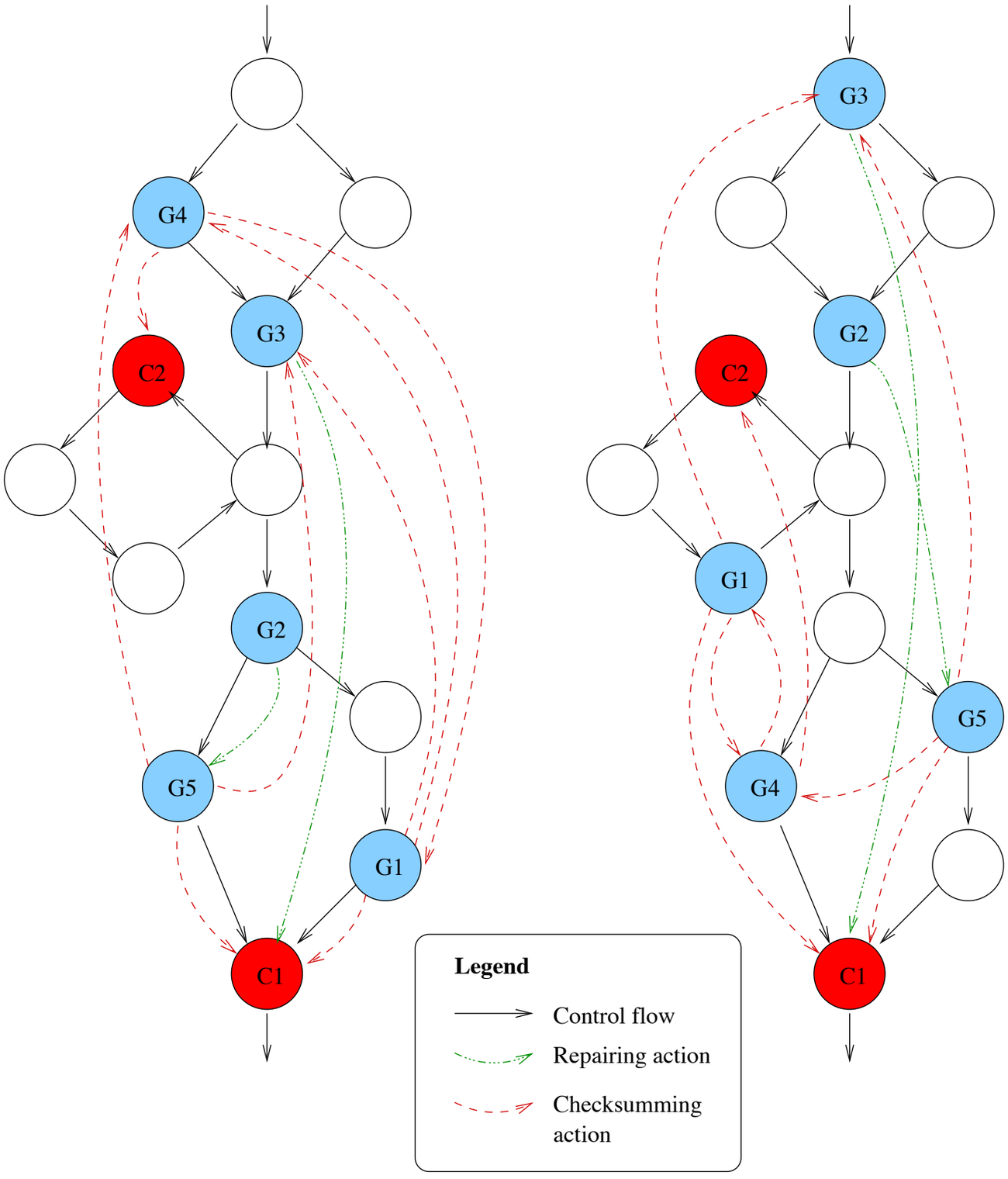
Modified CFG for program: C1; C2



(a) Partial execution ordering of the guards



(b) The corresponding guard graph



(b) Two possible placements of the guards in a CFG

Generating Hash Functions

In order to prevent pattern matching and collusive attacks it is important to have a set of different hash functions

- Hash functions that are “cryptographically secure” are large and slow and have static and dynamic features that the attacker can exploit

- In the tamper proofing setting we are interested in the following aspects of hash functions:
  - size (we may include many of them in the program)
  - speed (we may execute them frequently)
  - stealth (resistant to pattern-matching)

Generating Hash Functions

Need diversification = obfuscation

```
typedef uint32* addr_t;
uint32 hash1 (addr_t addr, int words) {
  uint32 h = *addr;
  int i;
  for (i = 1; i < words; i++) {
     addr++;
     h ^= *addr;
  }
  return h;
}
```

Simple hash XOR-based

```
typedef uint32* addr_t;
uint32 hash2 (addr_t start, addr_t end) {
  uint32 h = *start;
  while (1) {
     start++;
     if (start >= end) return h;
     h ^= *start;
  }
}
```

Simple hash variant

Generating Hash Functions

Need diversification = obfuscation

```
typedef uint32* addr_t;

uint32 hash3 (addr_t start, addr_t end, int step) {
  uint32 h = *start;
  while (1) {
     start += step;
     if (start >= end) return h;
     h ^= *start;
  }
}
```

...adding a step through the code

```
typedef uint32* addr_t;

uint32 hash4 (addr_t start, addr_t end, uint32 end) {
  addr_t t = (addr_t)((uint32)start + (unint32)end + rnd;
  uint32 h = 0;
  do {
     h += *((addr_t)(-(uint32)end - (uint32)rnd + (unint32)t));
     t++;
   } while (t < (addr_t)((uint32)end + (uint32)end + (uint32)end;
   return h;
}
```

...adding and subtracting random values rnd

Generating Hash Functions

- ✓Family of hash functions used by Skype!
- ✓To prevent the address of the region to be checked to appear in the code, the initialization section is obfuscated and the addresses are dynamically computed
- ✓Select randomly op into {add, sub, xor, etc..}
- ✓Generate a number of hash by modifying op
- ✓The Skype protocol was broken by looking for a signature of the address computation!

```
uint32 hash7() {
  addr_t addr;
  addr = (addr_t)((uint32)addr ^ (uint32)addr;
  addr = (addr_t)((uint32)addr + (uint32)addr;
  uint32 hash = 0x320E83 ^ 0x1C4C4;
  int bound = hash + 0xFFCC5AFD;

  do {
     uint32 data = *((addr_t)((uint32)addr + 0x10;
     goto b1;
       asm volatile(".byte 0x19");
     b1:
     hash = hash op data;
     addr -= 1;
     bound--;
  } while (bound != 0);
  goto b2;
       asm volatile(".byte 0x73");
  b2:
  goto b3;
       asm volatile(".word 0xC8528417,0xD8FBBD1,0xA36CFB2F");
       asm volatile(".word 0xE8D6E4B7,0xC0B8797A");
       asm volatile(".byte 0x61,0xBD");
  b3:
  hash -= 0x4C49F346;
  return hash
}
```

Hiding Hash Values

```
h = hash(start,end)
if (h != 0xca7babe5) abort()
```

not stealthy!

```
start:  0xab01cd02
        0x11001100
slot:   0x????????
        0xca7ca7ca
end:    0xabcdefab
```

```
h = hash(start,end);
if (h) abort();
```

- ✓ A possible solution is to *hide the expected value literals*.The idea is to construct a hash function such that, unless the code has been hacked, always hashes to zero.
- ✓ To this end we can use an *empty slot* within the region to protect, and later give to this slot a value that makes the region hash to zero.

Hiding Hash Values

Tamperproof (P,n):

1. Insert n checks: `if (hash(start,end)) RESPOND()` randomly in P
2. Randomize the basic blocks
3. Insert m ≥ n corrector slots: c1,...,cm
4. Compute n overlapping regions 1_1,...,I_n: I_i = [$start_i$; end_i] and associate I_i with c_i
5. Associate checks with regions and set c_i such that hash(I_i)=0 (and a fingerpint?)

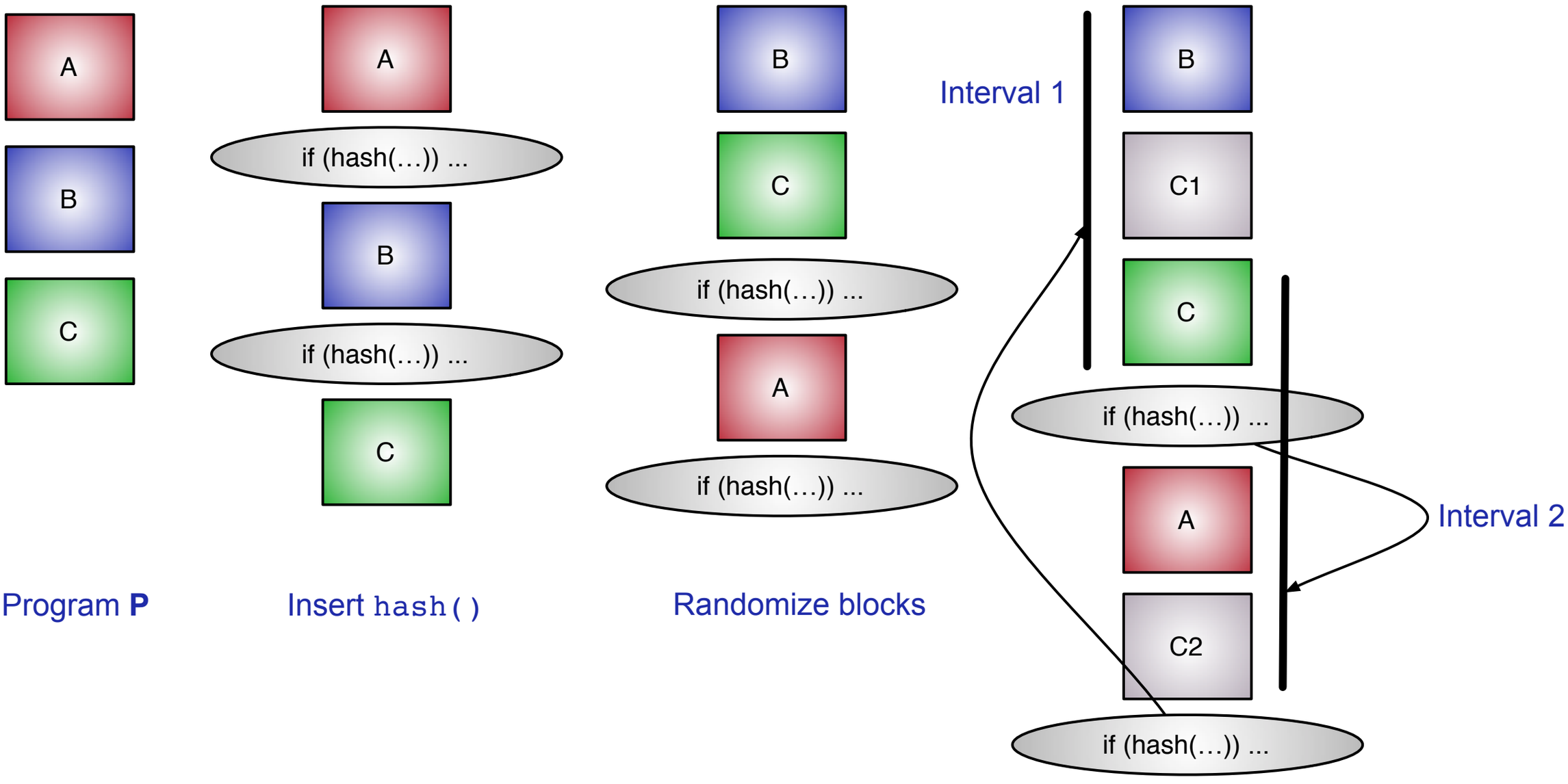
Technologies

intertrust®

Patent: 20050210275, Jan 2003

Dynamic self-checking techniques for improved tamper resistance. Horne et al. DRM 2001

Hiding Hash Values



Hiding Hash Values: computing corrector slots

$$x = [x_1, \ldots, x_n]$$

x_k Corrector slot

$$hash(x) = \sum_{i=1}^{n} C^{n-i+1} x_i$$

Odd constant

$$z = \sum_{i \neq k}^{n} C^{n-i+1} x_i \longrightarrow C^{n-k+1} x_k + z = 0 \mod 2^{32}$$

Variable!

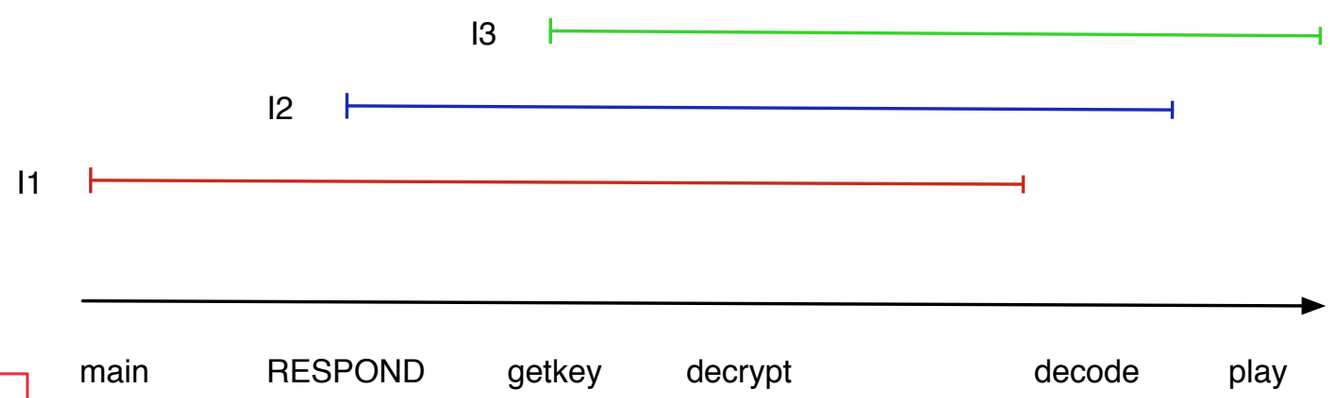
Register size

Theorem: *ax = b mod n* is solvable if *d* |*b* where *d = gcd(a,n) = ax'+ny'*
There are *d* solutions:

$$\begin{aligned} x_0 &= x'(b/d) \mod n \\ x_i &= x_0 + i(n/d) \qquad \text{where } i = 1, \ldots, d-1 \end{aligned}$$

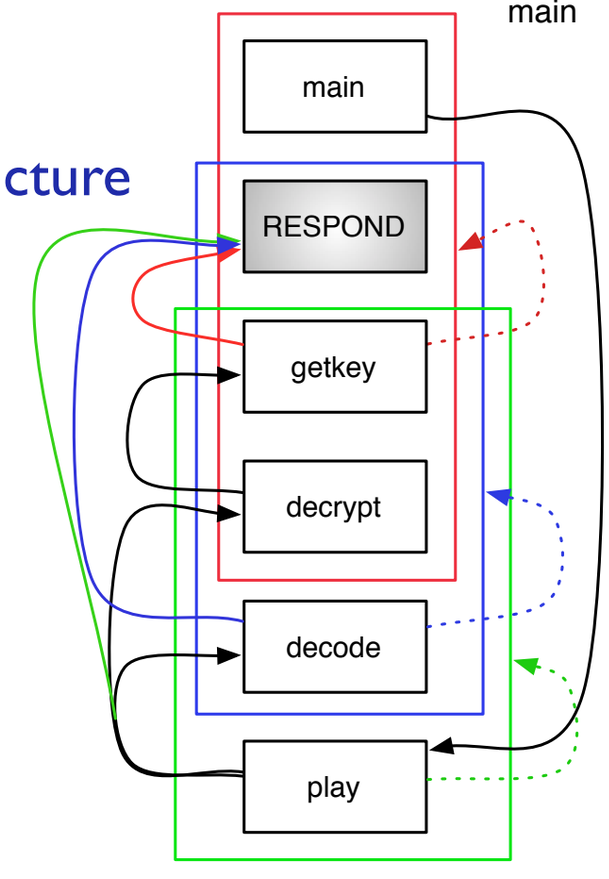
Overlap factor **n**: most (~80%) bytes of the code are checked by at least **n** checkers

Overlap factor 6 is considered optimal!

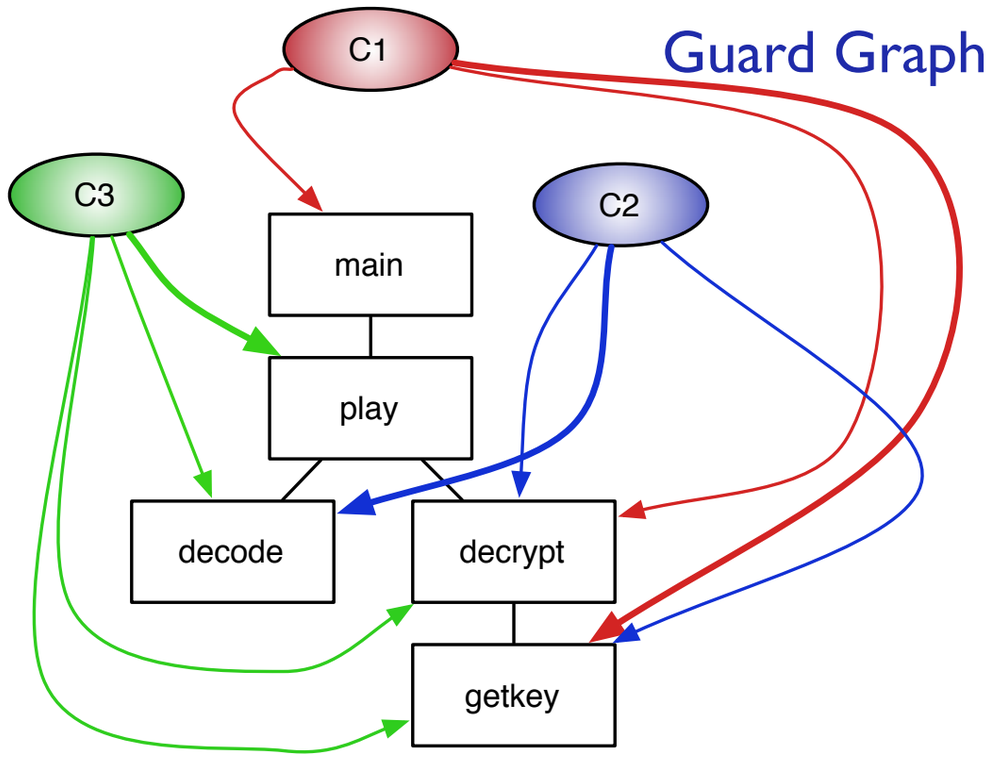


Here overlap factor 2

Interval structure



Guard Graph



Checking by Introspection

- How can we be sure that the attacker won't tamper with the hash computation itself?
  1. build up a network of checkers and responders
  2. repair code that has been tampered with
  3. hide the hash values
- Can we attack all introspection algorithms?

Attacking self-hashing algorithms

✓How to attack introspections algorithms?

- Analyze the code to locate the *checkers*, or
- Analyze the code to locate the *responders*, then
- Remove or disable them without destroying the rest of the program

✓Attack can just as well be external to the program

✓*The attacker may modify the execution environment*

Attacking self-hashing algorithms

✓Processors treat *code* and *data* differently

✓TLBs (Translation Lookaside Buffers) and caches are *split in separate parts for code and data*

✓In the hash-based algorithm code is accessed

- as code (when it's being executed)
- as data (when it's being hashed)

➡ sometimes a function will be read into the I-cache and sometimes into the D-cache

➡ the hash-based algorithms assume that the function is the same regardless of how it is read

Attacking self-hashing algorithms

✓Attack: modify the OS such that

- *redirect reads* of the code to the original, *unmodified program*, in this way hash values will be computed as expected
- *redirect execution* of the code to the *modified program*, in this way the modified program will be executed

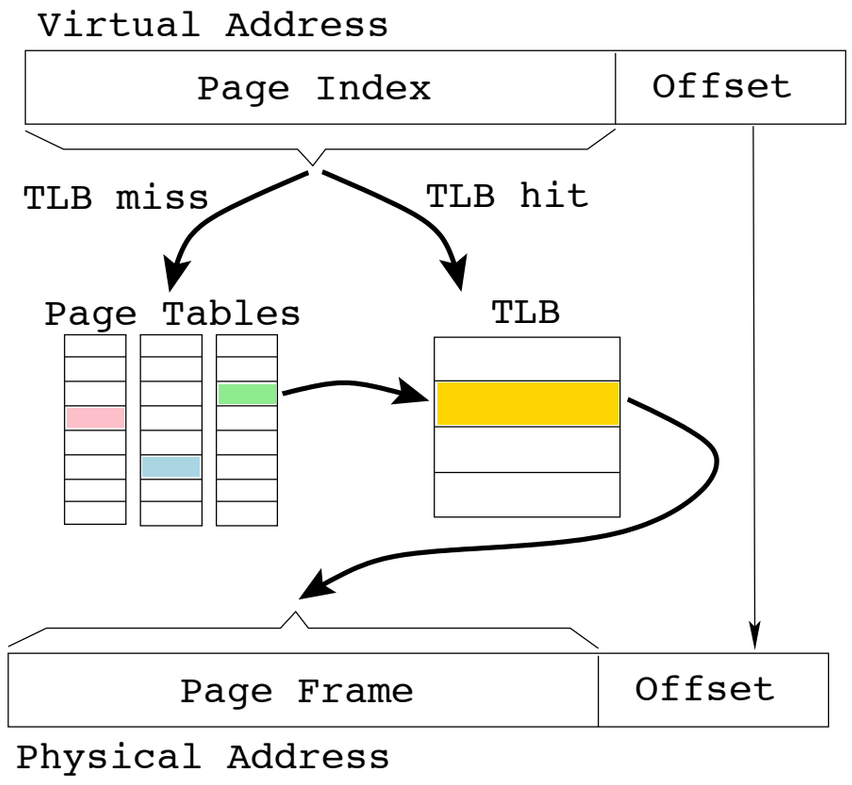
Attacking self-hashing algorithms

✓ATTACK(P, K):

- Copy program P to P_{orig}
- Modify P as desired to a hacked version P’
- Modify the operating system kernel K such that data reads are directed to P_{orig} and instruction reads to P’

Attacking self-hashing algorithms

✓Typical memory management system



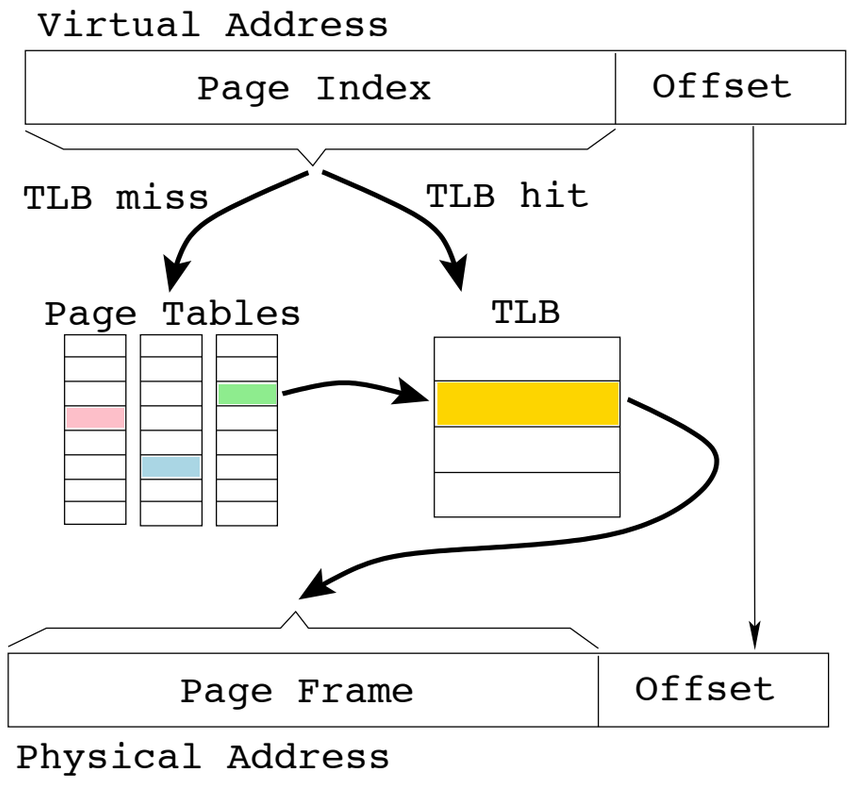
✓Each process operates within its own virtual address space

✓The OS or CPU maintains a set of page tables that map the virtual addresses seen by the process to physical addresses of the underlying HW

✓To avoid to look up every memory access in the page tables, the TLB (translation look aside buffer) chaches recent lookups

Attacking self-hashing algorithms

✓Typical memory management system

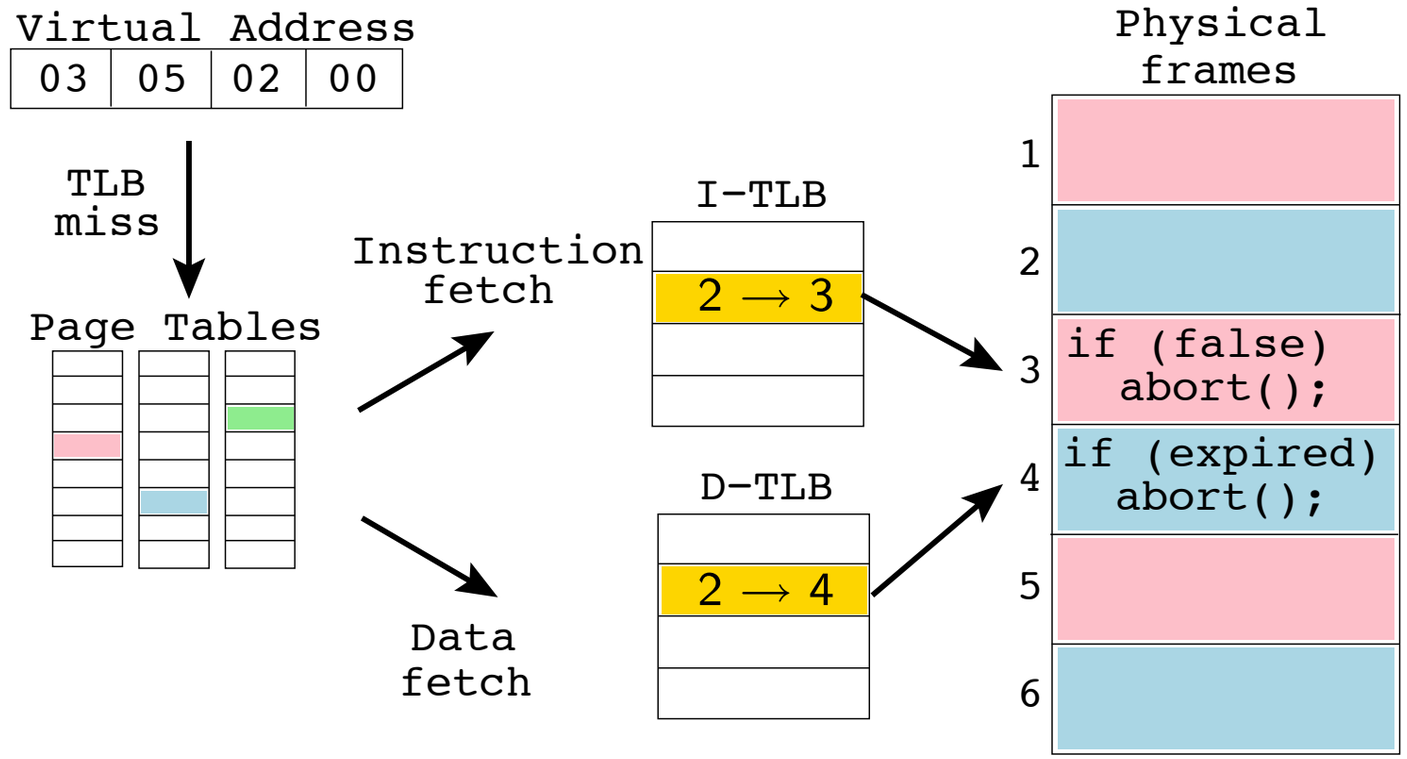


✓On a TLB miss walk the page tables (slow) and update the TLB with the new virtual-to-physical address mapping

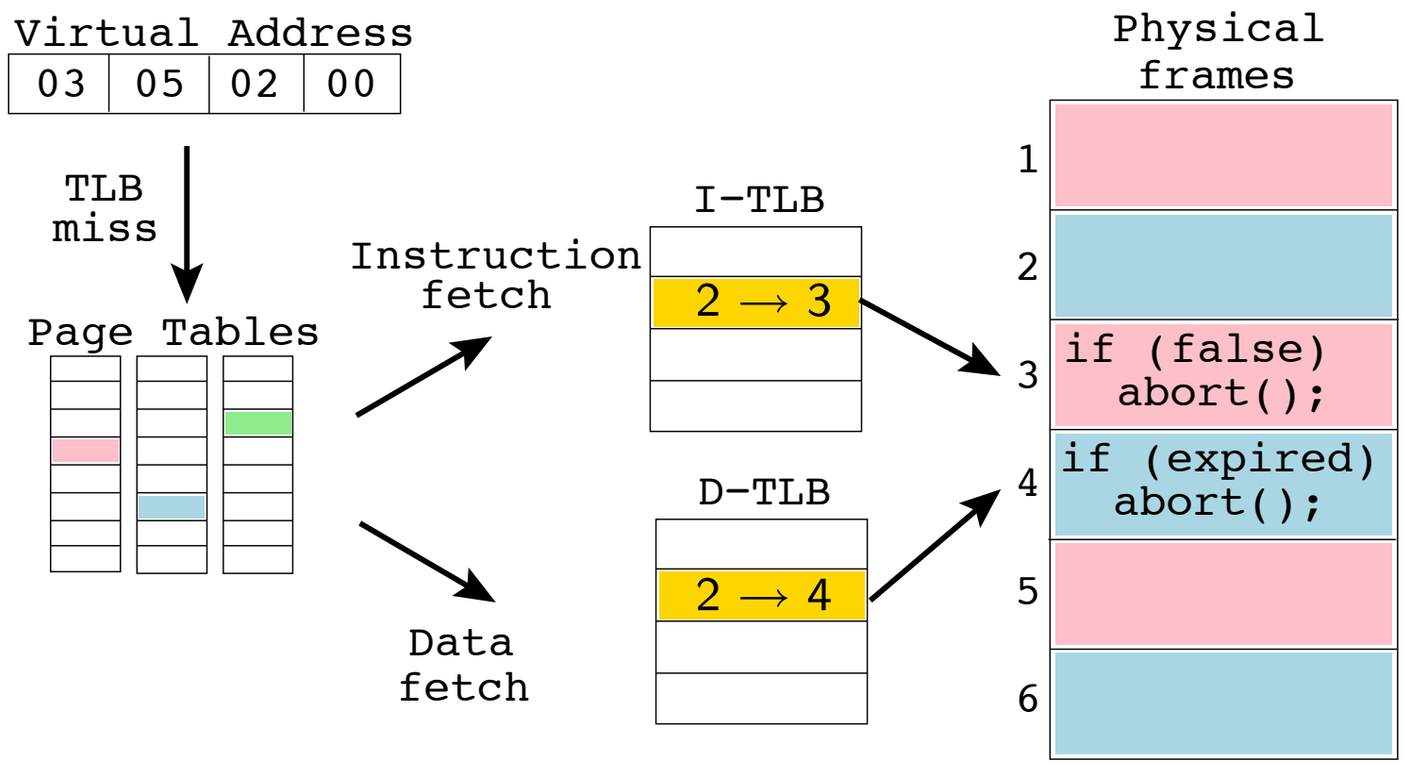
✓On the UltraSparc, the hardware gives the OS control on a TLB miss by *throwing one of two exceptions depending on whether the miss was caused by a data or an instruction fetch*

Attacking self-hashing algorithms

- ✓Copy P to P_{orig} and modify P however you like
- ✓Arrange the physical memory such that frame i comes from the hacked P and frame i + 1 is the corresponding original frame from P_{orig}
- ✓Modify the kernel: if a page table lookup yields a v → p virtual to physical address mapping, I-TLB is updated with v → p and D-TLB with v → p + 1

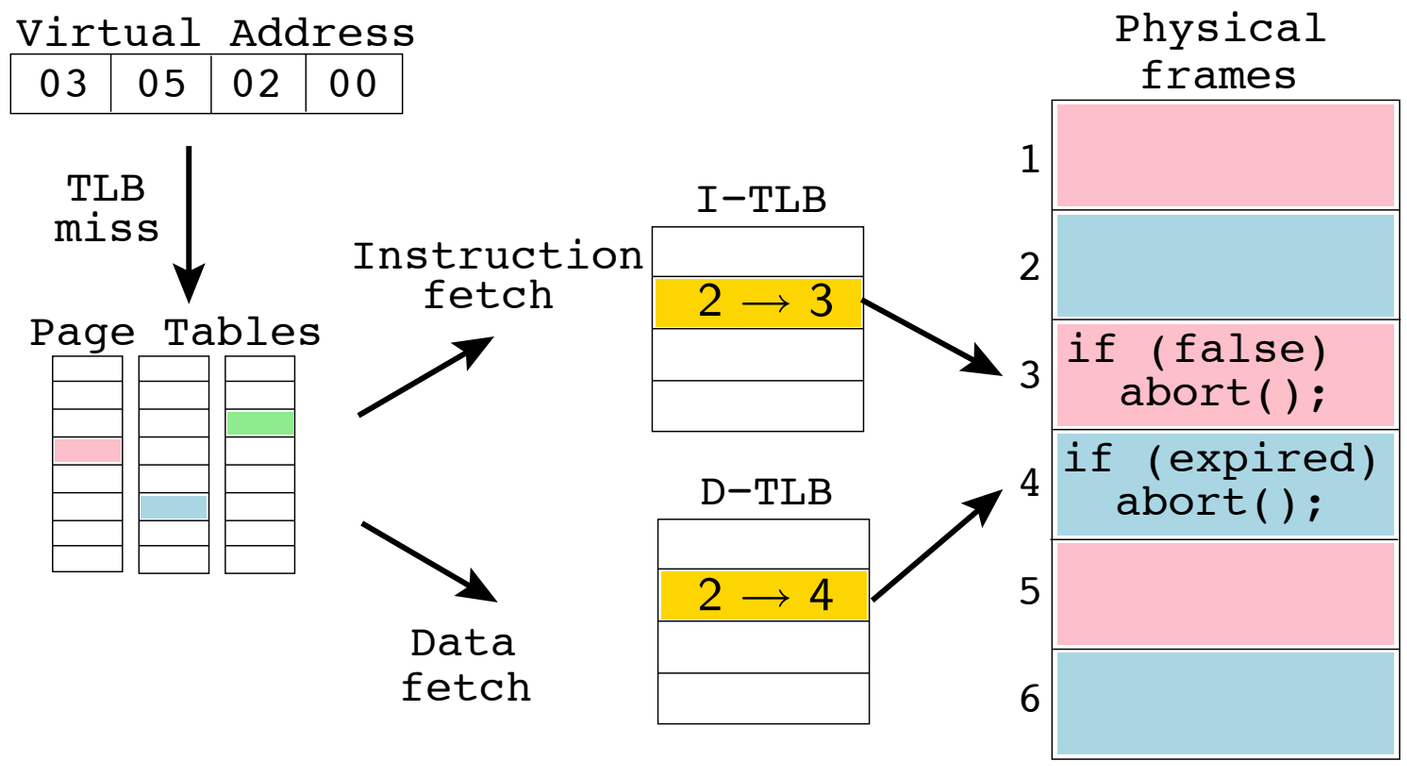


Attacking self-hashing algorithms



- ✓The attacker has modified the program to bypass a license-expired check
- ✓The original program pages are in blue
- ✓The modified program pages are in pink

Attacking self-hashing algorithms



✓The program tries to read its own code in order to execute it

➡ the processor throws an I-TLB miss exception, the OS updates the I-TLB to refer to the modified page

✓The program tries to read its own code to hash it

➡ the processor throws an D-TLB miss exception, the OS updates the D-TLB to refer to the original, unmodified page

State Inspection

Problems in Introspection

- ✓Introspection performs unusual operations: *read their own code*
- ✓The only check is on the code (syntax)
- ✓Easy to catch (non stealthy)

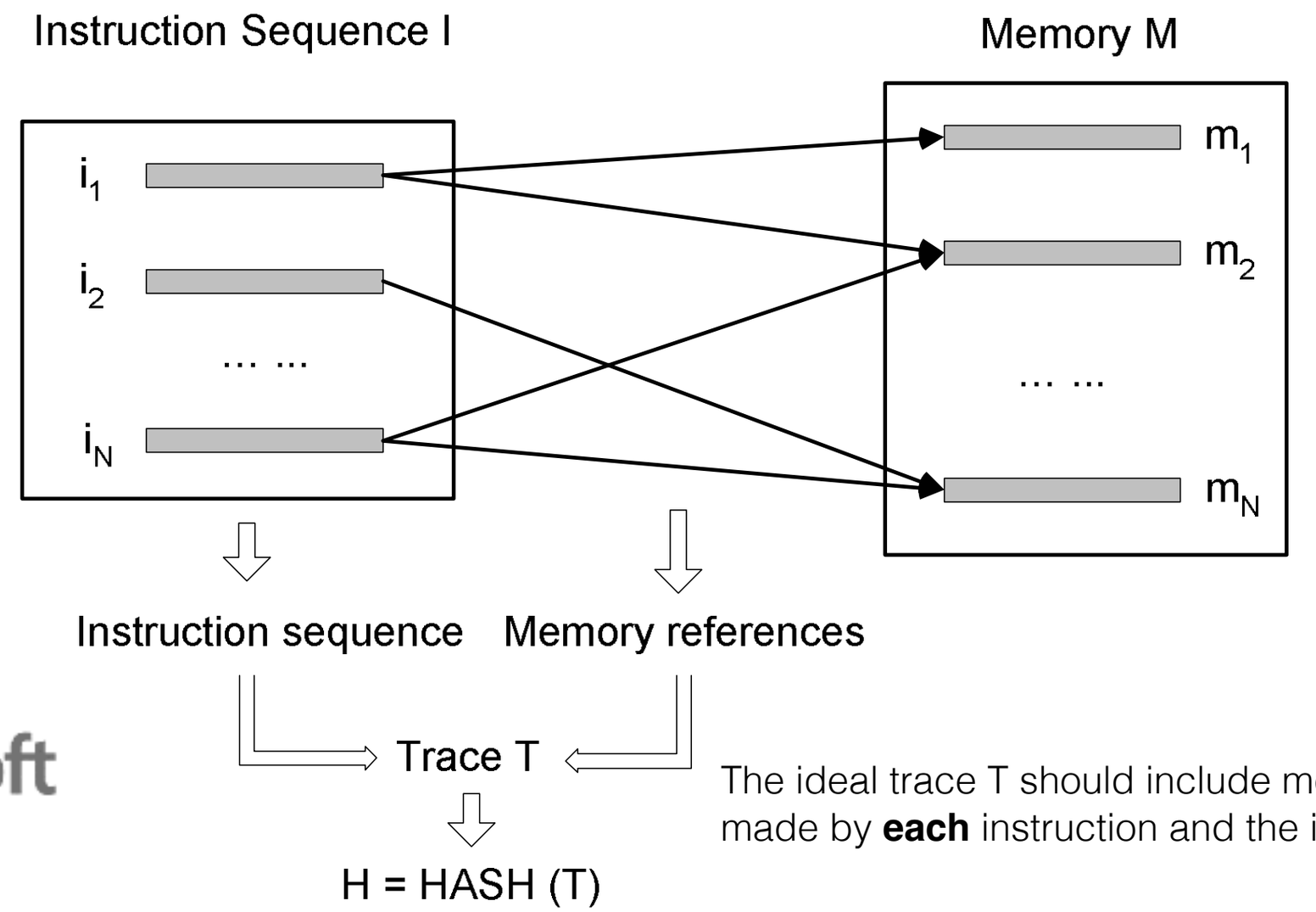
Idea: Make state inspection!

Is the code correct?  Is the program in a good state?

Possible Solutions

- ✓Adding assertion checks (Proof Carrying Code). Not easy: the current state of the program depends on all previous states, it is difficult to come up with non-trivial invariants to add and check
- ✓Call sensible functions on challenge data (like *hash*):
  - automatically generating challenge data that exercises important aspects of a function is not easy
  - adding this data in the program in a stealthy way is not easy
  - ...be careful on side effects of challenge-based computations! (memory consumption)
- ✓**Oblivious hashing:** hash over computed variables and the outcome of control flow predicates

Oblivious Hashing



Microsoft

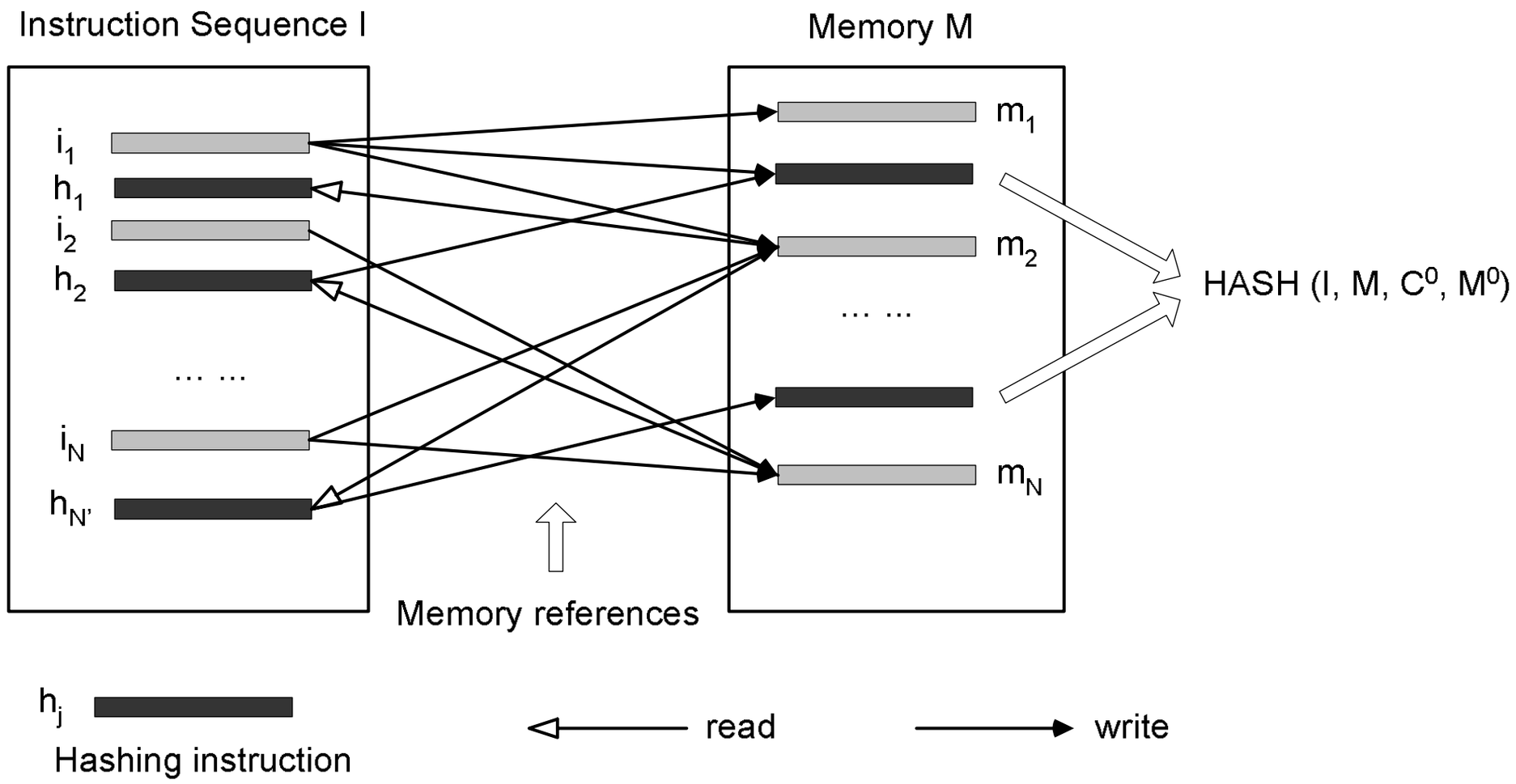
The ideal trace T should include memory references made by **each** instruction and the instruction itself.

Attack model: The attacker changes the instruction sequence and memory content during runtime in order to produce a correct hash value for a given set of inputs (which can be exorbitantly large)!

Oblivious hashing: a stealthy software integrity verification primitive. Cheng et al. Information Hiding 2002

Oblivious Hashing

Implementation: by *code injection*, (e.g., in profilers and dynamic bounds checkers)! Inject into the host code "monitor" instructions that capture each step of the computation and compute the oblivious-hashing value as the computation proceeds



a = exp **is transformed to** **a = (t = exp, HASH(t), t)**

if (exp) {…} **is transformed to** **if ((t=exp, HASH(t), t)) {…}**

Practical Integrity Protection with Oblivious Hashing

Mohsen Ahmadvand, Anahit Hayrapetyan, Sebastian Banescu, and Alexander Pretschner
Technical University of Munich
firstname.lastname@cs.tum.edu

- OH is an appealing protection measure but it is unable to protect nondeterministic program traces, i.e., memory accesses with values dependent on user-input or environment variables. This is mainly because hashing such traces results in different hashes for different program executions.
- Only 13% of program instructions evaluated are deterministic.
- Employ the proposed protection scheme on a dataset of 29 real-world applications to evaluate the security as well as the induced overheads

ACSAC 2018

Practical Oblivious Hashing

- ✓OH checks the execution effects in memory, as opposed to checking the code itself
- ✓It is therefore *harder to identify* (stealthy) *or trick* (resilient) OH in contrast to SC (Self-Checksumming schemes)
- ✓However, OH is *unable to protect nondeterministic program traces*, namely memory accesses with values dependent on user input or environment variables.
- ✓Indeed, hashing such traces results in *different hashes* for different program executions

Nondeterminism

✓To face nondeterminism we have to:

- Automatically extract deterministic program fragments (0,5% median on 29 applications)
- **Protect nondeterministic fragments**

✓ Nondeterministic data: data that is provided by the environment or user and that can vary across different executions

✓There are two types of nondeterminism:

- *Data dependency* which capture instructions where at least one of their operands depends on nondeterministic data
- *Control flow dependences* which includes instructions in branches with condition that depends on nondeterministic data

Dependence Graph

- ✓Use *user-input dependency detection* to identify nondeterministic instructions:
  - Dependence graphs capture dependency relations (both data and control-flow) among program instructions. Nodes are program instructions and edges indicate data and control-flow dependencies among them.
  - Given the dependence graphs, identifying instructions that depend on nondeterministic data is solvable by a *graph reachability analysis*.

Nondeterministic Instructions

- ✓The outcome of the analysis indicates the instructions that are data and control-flow dependent on inputs
  - **III** input-independent data and control-flow instruction
  - **DII** data-independent instruction
  - **DDI** data-dependent instruction
  - **CFDI** control-flow dependent instruction
- ✓**III** instructions can directly be protected using standard OH
- ✓While hashing memory references of both **DDI** and **CFDI** yields different hashes for different inputs

Nondeterministic instructions

Listing 1: Fictional electricity meter application with varying rates

```
1  enum period {Peak, OffPeak, Normal};
2  float computeUsage(float *kwMinute, int size , enum period rate ){
3     float usage = 0;                                  // III
4     for ( int i =0; i < size ; i++){                  //DDI
5        float rating = 1.0;                            //DII|CFDI
6        if ( rate == Peak){                            //DDI
7           rating = 2.0;                               //DII|CFDI
8        } else if ( rate == OffPeak){                  //DDI
9           rating = 0.5;                               //DII|CFDI
10       }
11          usage += kwMinute[i]*rating ;               //DDI
12    }
13    return usage;                                     //DDI
14 }
15  ...
16 meterUsageCycle(float *kwMinute, int size , enum period rate ){
17    kwHour = 0;                  // III
18    if ( isHoliday () ){         //DDI
19          enum period normalRate = Normal;            //DII|CFDI
20          kwHour = computeUsage(kwMinute, size, normalRate);//DDI|CFDI
21    } else {
22          kwHour = computeUsage(kwMinute, size, rate);      //DDI|CFDI
23    }
24     ...
25 }
```

Inconsistent Hash Values

- ✓Hashing both **DDIs** and **CFDIs** may lead to inconsistent hash values.
- ✓The incorporation of data-dependent variables into hashes yields unverifiable hashes.
- ✓Similarly, covering instructions in input-dependent branches by OH may lead to two types of inconsistent hashes:
  - the expected hash cannot be precomputed because nondeterministic values lead to the execution of different branches,
  - the expected hash cannot be computed as a result of taking some branches for an indefinite number of times depending on nondeterministic values.

Short Range Oblivious Hashing SROH

- ✓Short Range Oblivious Hashing SROH: is a technique comprised of a path-specific set of hash variables and a tailored verification.
- ✓SROH can protect **DIIs** residing in nondeterministic branches
- ✓IDEA: realize OH using distinct hash variables for every **ordered sequence of basic blocks (OSBB)** that are strictly executed in the identified order, in every execution. That is, for a given block in the sequence, all the preceding blocks are necessarily executed before reaching the block.
- ✓Another requirement is that hashes, in such sequences, are verified in one of the blocks within the sequence—conveniently, the last block.
- ✓SROH addresses the two aforementioned issues:
  - hashes are computed in the scope of identified sequences and, therefore, not taking a branch does not affect the expected hashes,
  - the verification of hashes takes place within the identified sequences, and thus the number of times a branch is taken does not corrupt hashes.

Ordered sequences of basic blocks OSBB

- ✓Dominator tree relationships can be utilized to identify OSBB
- ✓*Input-independent loops*, the looping conditions that do not rely on any nondeterministic input, do not require any specific action.
- ✓The number of iterations of an *input-dependent loop* is determined by (nondeterministic) inputs. We ensure hash consistencies for such loops by abstracting their (indefinite) number of iterations. That is, we only consider their (loop) bodies; and, subsequently, loop condition, iteration and end blocks are excluded from OSBB computations. Moreover, we require every OSBB to end immediately before an input-dependent loop begins.

Data Dependent Protector DDP

- ✓**III** are protected with OH
- ✓**DII** and **CFDI** are protected by SROH
- ✓We propose to use a complementary protection technique, called *data dependent protector (DDP)*, for **DDI** instructions that OH/SROH cannot protect.
- ✓Unlike OH, such an integrity protection technique must be unaffected by nondeterministic input.
- ✓Since SC protection monitors program code, not the execution, it is not affected by any input data. Therefore, we can use it to build the DDP layer.

Intertwined protection

- ✓The idea is to have OH/SROH protect SC guards, which can also be seen as the implicit protection of nondeterministic segments by OH.
- ✓The protection incorporates all (invariant) attributes of guards (i.e. offset, size, and expected hash) into OH/SROH hashes.
- ✓This works because self-checksumming guards do not depend on inputs: they compute hashes over certain process memory blocks and compare them with expected values, which remain unchanged, independent of the program inputs

- ✓Performance: Although the application of resilient protection on CPU-intensive programs yields a high overhead, we believe these overheads are acceptable for non-CPU-intensive program

Remote Tamper-proofing

Client-Server

- ✓The program to protect runs remotely on an untrusted client.

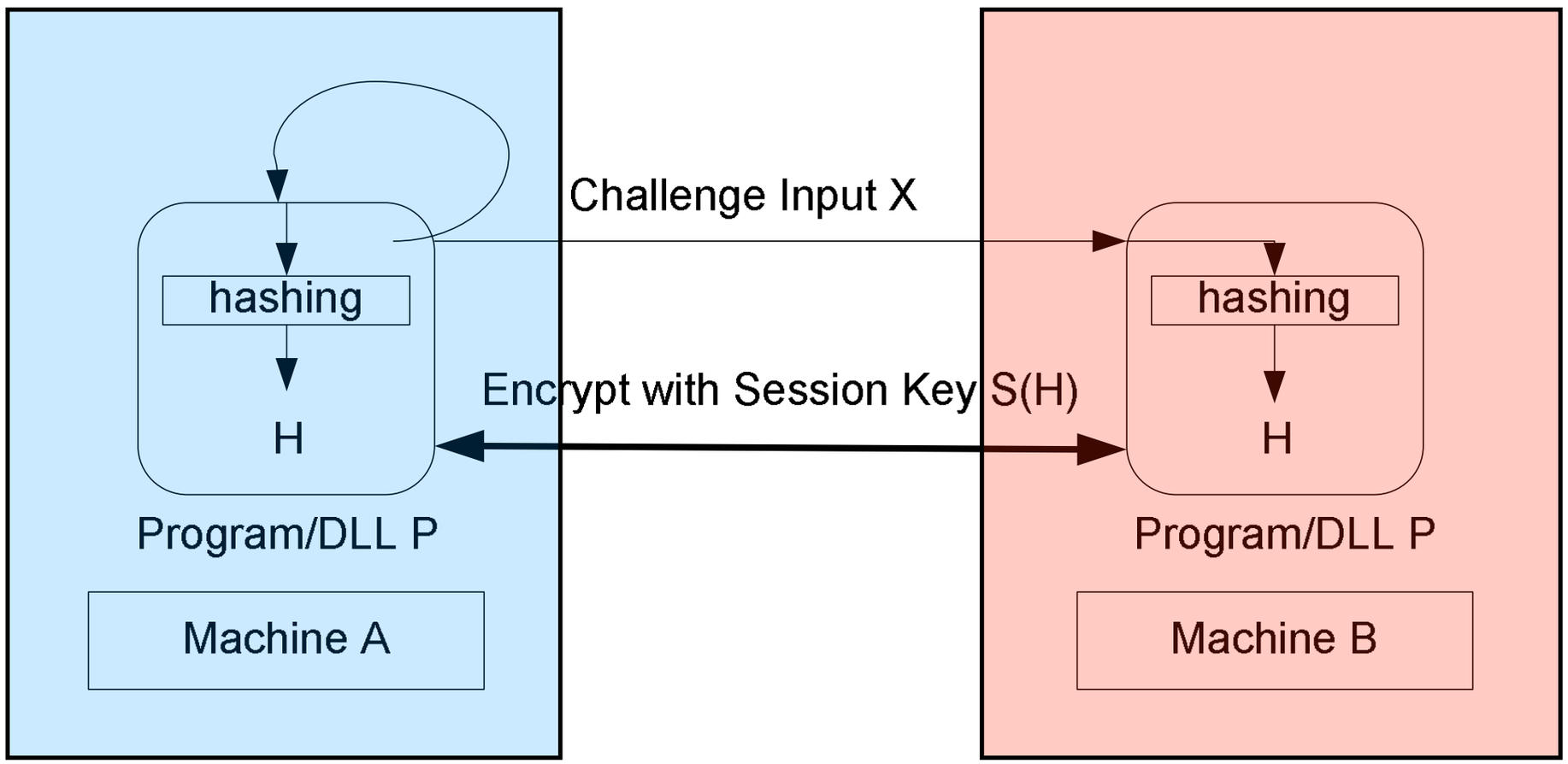
- ✓To protect the client code we can *move it to the server* (SW as a service)
  - high computational load for the server and high latency for the client

- ✓Intermediate level solution
  - some computation is done server-side
  - some computation is done client-side
  - balance computation, network traffic and protection

- ✓Client is in constant communication with the server: *exploit these communications* for checking the integrity of the code running on the client
  - **the client can lie**

Oblivious hashing in remote execution

- ✓Use oblivious hashing
- ✓Get remotely challenge values for hashing
- ✓Compute as usual the hashing



Client vs Server via Code Slicing

- ✓Algorithm developed to *prevent piracy*, namely to prevent the attacker from getting a working copy of the application
- ✓IDEA: split software modules into open and hidden components:
  - *open components*: installed and executed on *un-trusted machines*
  - *hidden components*: installed and executed on *trusted machines*
- ✓Open components can be stolen but they are incomplete (they only provide a subset of the application functionality)
- ✓Prevent tampering with the code on the server
- ✓*Resilience*: Deriving the hidden components by observing the code of the open components and their run-time interactions with the hidden components requires a *great deal of effort*
- ✓*Cost*: Communication between open and hidden components

Hiding Program Slices for Software Security. X. Zhang and R. Gupta. CGO 2003

Application Scenario

- ✓ *Untrusted User*: prevent authorized user from transferring the software to unauthorized machines. In this case hidden components are executed on the server and open components on the user device.

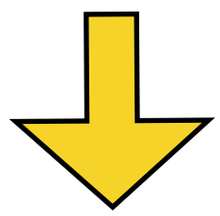
- ✓ *Untrusted Server*: Users carry mobile devices that host applications that may be executed on a remote server. The sever is considered untrusted and we want to prevent piracy of the applications. In this case hidden components are executed on the user device and open components on the server.

Client vs Server via Code Slicing

CLIENT

```
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);
}
```

- ✓Function **f** originally executed on the untrusted client that we want to divide between client and server
- ✓**a** is an important variable that we want to hide on the server
- ✓whenever the client needs **a** he/she can get it from the server
- ✓move code that depends on **a** to the server for better performances



- ✓Program Slicing

Hiding Program Slices for Software Security. X. Zhang and R. Gupta. CGO 2003

Client vs Server via Code Slicing

CLIENT

```
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** and move the selected code to the server
- ✓Keep unimportant variable **c** on both the client and the server for better performance
- ✓Don't move large data structures for better performances
- ✓Overhead depends on how much of the program is hidden on the server. On a LAN 3 to 58%

Hiding Program Slices for Software Security. X. Zhang and R. Gupta. CGO 2003

ORIGINAL CLIENT

```
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);
}
```

CLIENT

```
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();
}
```

SERVER

```
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;}
```

How does it work?

- ✓Function **f** is the original one
- ✓You want to hide a variable **`a`**
- ✓Compute a forward slice on **`a`** (pink)
- ✓You want to protect all the pink code and to do so you put it on the server in functions **`f1`**…**`f6`**
- ✓The client accesses the hidden functions by making RPCs (remote procedure call)
- ✓**`c`** is a partially hidden variable. It resides both on the client and on the server, but the code that updates it is split between the two

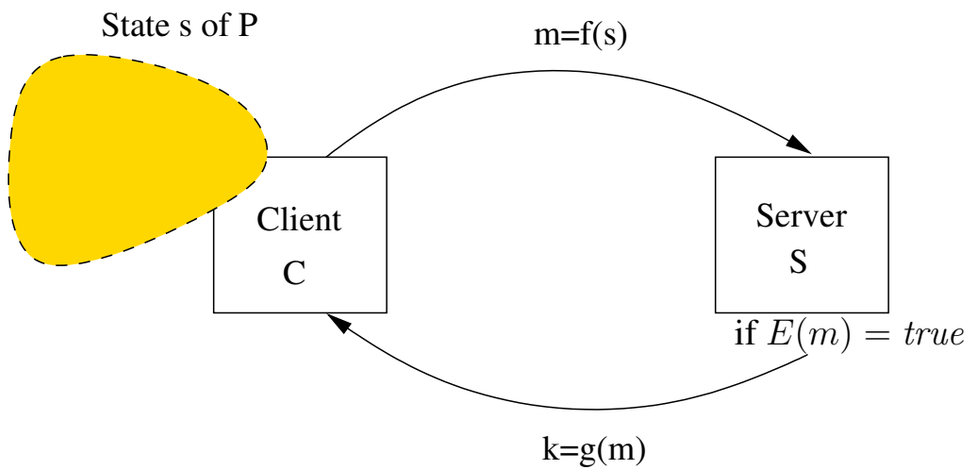
How does it work?

- ✓Runtime overhead from 3 to 58%
- ✓Depends on the amount of protection that is added
  - how much of the program is hidden on the server?
  - how much extra communications?
- ✓Zhang and Gupta's measurements were done over a local area network, in real scenarios server and clients are far way and the network latency makes this protection difficult to apply

Barrier slicing for remote SW protection

- ✓Focus on network applications: server and clients continuously communicate
- ✓The sever is willing to deliver its services only to clients that are in a valid state
- ✓The application **P** running on the client requires services delivered by **S**
- ✓Message **m** depends on the state **s** of the application **P**
- ✓Message **k** depends on the previous message **m**
- ✓In real scenario we have a *sequence of communication acts*

communication act



Barrier slicing for remote software protection, Ceccato et al. SCAM 2007

Improving Performances with Barrier Slicing

- ✓Application **P** is in a valid state **s** at communication act **C[s]** $\xrightarrow{m}$ **S** if **A(s) = true** where **A** is an assertion
- ✓**S** can analyze the integrity of **P** by analyzing the message **m** it sends
- ✓**S** trusts **P** at communication act **C[s]** $\xrightarrow{m}$ **S** if **E(m) = *true***
- ✓The protection mechanism is **not sound** if **E(m) = *true*** and **A(s) = *false***
- ✓The protection mechanism is **not complete** if **E(m) = *false*** and **A(s) = *true***
- ✓ Ideal situation when **E(m)** $\Leftrightarrow$ **A(m)**

Barrier slicing for remote software protection, Ceccato et al. SCAM 2007

Improving Performances with Barrier Slicing

- ✓ *Barrier Slicing* is computed on a code where some special statements are marked as barriers
- ✓ Barrier Slicing can be computed by stopping the computation of the transitive closure of the program dependencies whenever a barrier is reached.
- ✓ Given the PDG (N,E) the backward barrier slice with criteria $C \subseteq N$ and barrier $B \subseteq N$

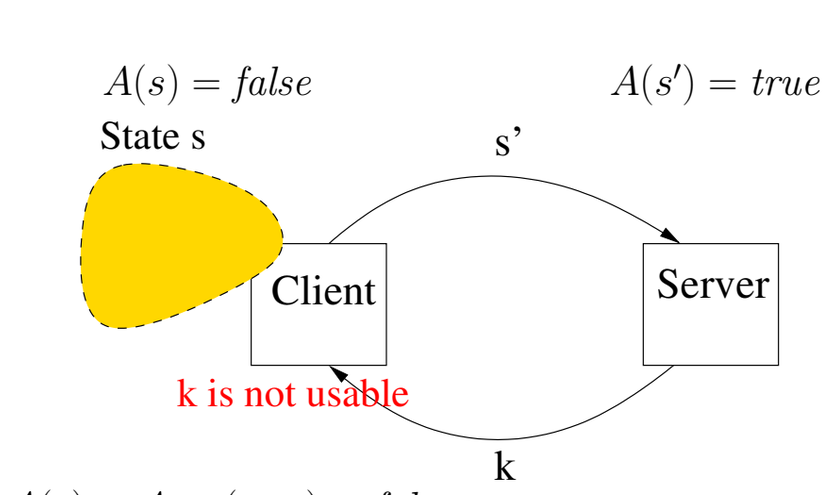
$$Slice_{\sharp}(C,B) = \left\{ m \in N \;\middle|\; \begin{array}{l} p \in m \longrightarrow^{*} n \;\wedge\; n \in C \;\wedge \\ p\,\langle n_1 ... n_l \rangle : \\ \forall 1 \leq i \leq l : \; n_i \notin B \end{array} \right\}$$

Barrier slicing for remote software protection, Ceccato et al. SCAM 2007

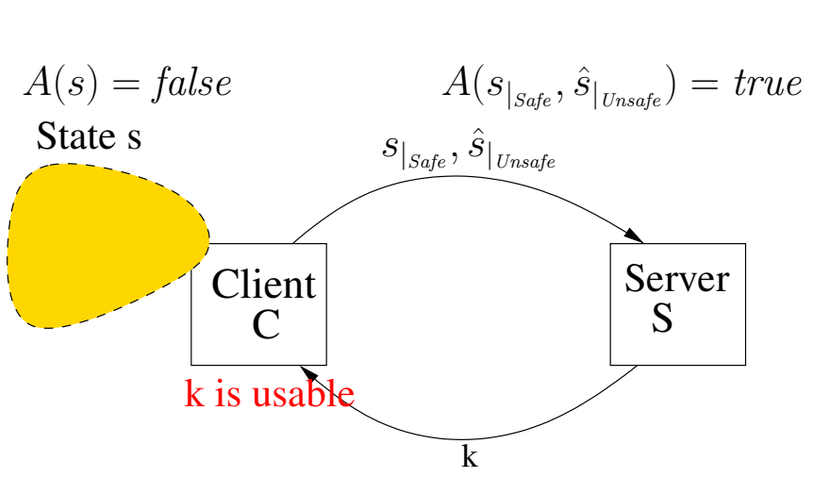
Improving Performances with Barrier Slicing

✓Let $Safe \subseteq Var$ be the subset of variables that determine the usability of message **k**

$$A(s) = A_{Safe}(s_{|_{Safe}}) \wedge A_{Unsafe}(s_{|_{Unsafe}})$$



(a) $Unsafe = \varnothing$



(b) $Unsafe \neq \varnothing$

* Solution **m = s** and **E = A**

* the previous solution does not work

Barrier slicing for remote software protection, Ceccato et al. SCAM 2007

Improving Performances with Barrier Slicing

- ✓ Barrier slicing can be used to develop a protection scheme that works when $Unsafe \neq \varnothing$
- ✓ The idea is to move the portion of P that maintains the variables in *Unsafe* to the server in order to prevent the attacker from tampering with them
- ✓ To limit the portion of code that has to be moved on the server we use barrier slicing
- ✓ The instruction marked as barrier are the ones that define the variables that belong to $Safe \subseteq Var$

Barrier slicing for remote software protection, Ceccato et al. SCAM 2007

Orthogonal Replacement

Orthogonal Replacement

- ✓IDEA: *periodically replace the client code with new orthogonal client code* such that:
  - the combination with the server is functionally equivalent to the original client-server application
  - successive client applications are *orthogonal* to previous ones: *the attacker cannot take advantage from previous attempts*
  - orthogonality is achieved through code obfuscation and code splitting
- ✓*the time that the attacker has for reverse engineer the application is limited*

Remote software protection by orthogonal client replacement. Ceccato et al. SAC 2009

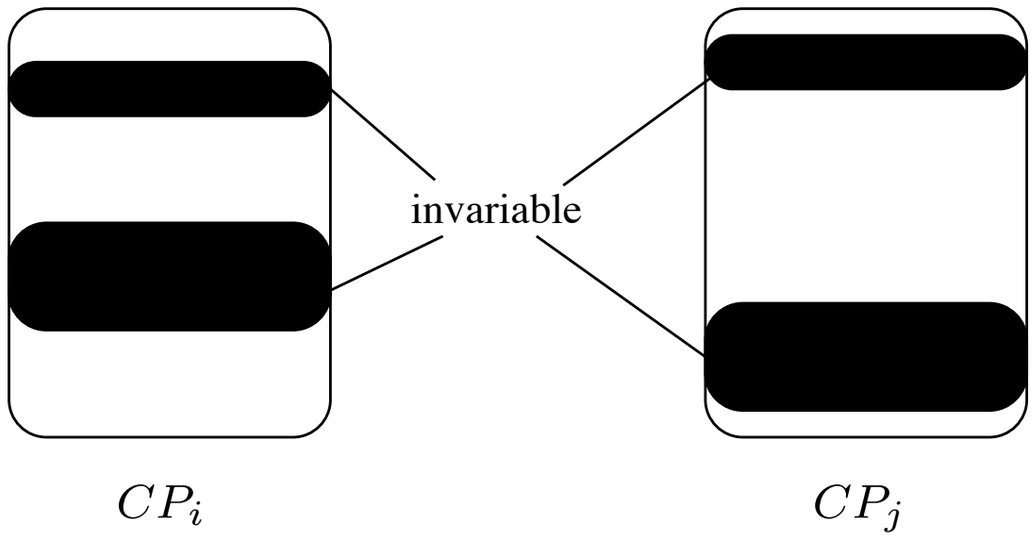
Orthogonal Replacement

- ✓Identify the *critical part* CP of the code, namely the portion of code that it is important to protect
- ✓Idea: *keep on substituting the critical part* of the client with new versions that are orthogonal to the previous ones. Ideally, orthogonality ensures that an attacker cannot use the knowledge gained from the (static and dynamic) analysis of any previous client version to tamper with the current code of the client's critical part.
- ✓*Code obfuscation*: is used to generate variants of CP: $CP_1 \ldots CP_n$
- ✓*Code splitting*: the invariant part shared by CP_n and the previous versions is moved on the server

Remote software protection by orthogonal client replacement. Ceccato et al. SAC 2009

Orthogonal Replacement

There might be portions of the CP that need to be shared by all the variants. Namely some portions of the code that are *invariant*. This invariant part is application dependent and it defines a limit on the degree of protection achieved with this technique



The portion that has to be orthogonal is the white one (namely the one that can be modified)

Remote software protection by orthogonal client replacement. Ceccato et al. SAC 2009

Orthogonal Replacement

- A statement s of code C is orthogonal to a statement p of code C' when the analysis of s in C does not reveal anything about the statement p in C'.
- C and C' are orthogonal if all the statement of C and C' are orthogonal
- Orthogonality is hard to define and quantify
- Clone-based orthogonality: two statements p, q are c-orthogonal if:

$$(p, q) \notin ClonedSt(\, White(C_i),\, White(C_j))$$

Two programs are c-orthogonal if:

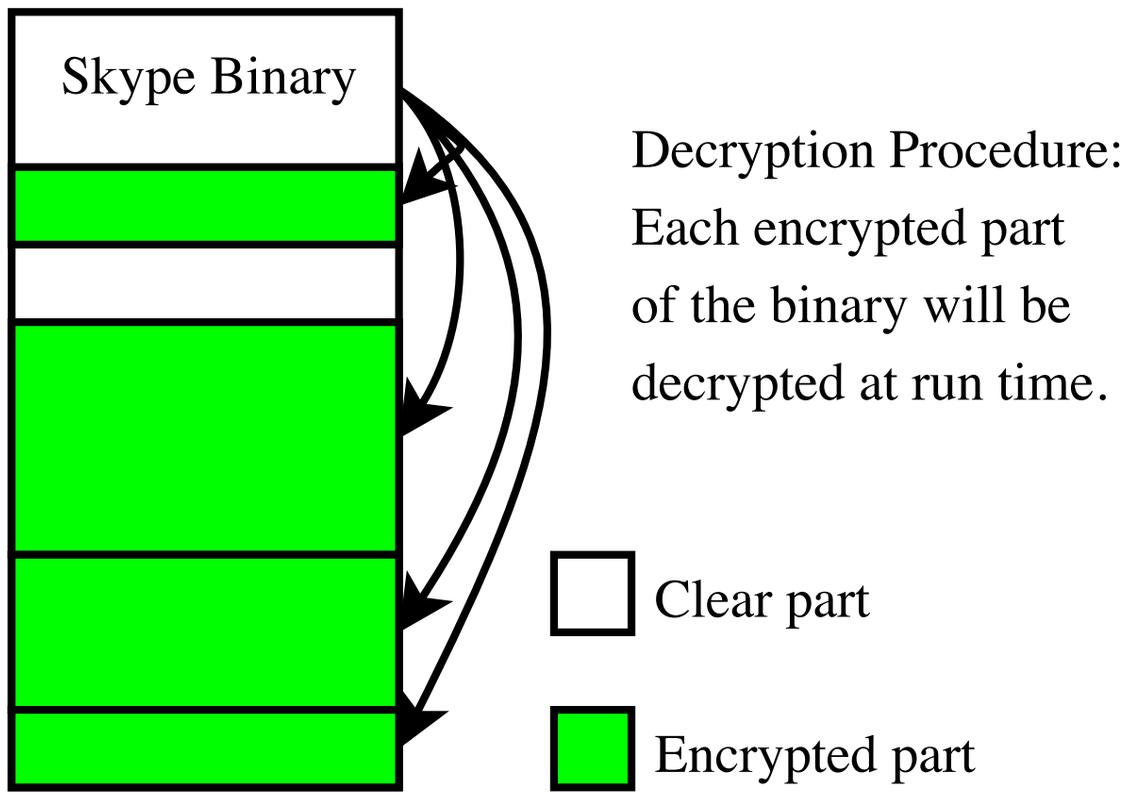
$$\forall p \in White(C_j), \forall q \in White(C_i) \colon p \perp_c q.$$

Remote software protection by orthogonal client replacement. Ceccato et al. SAC 2009

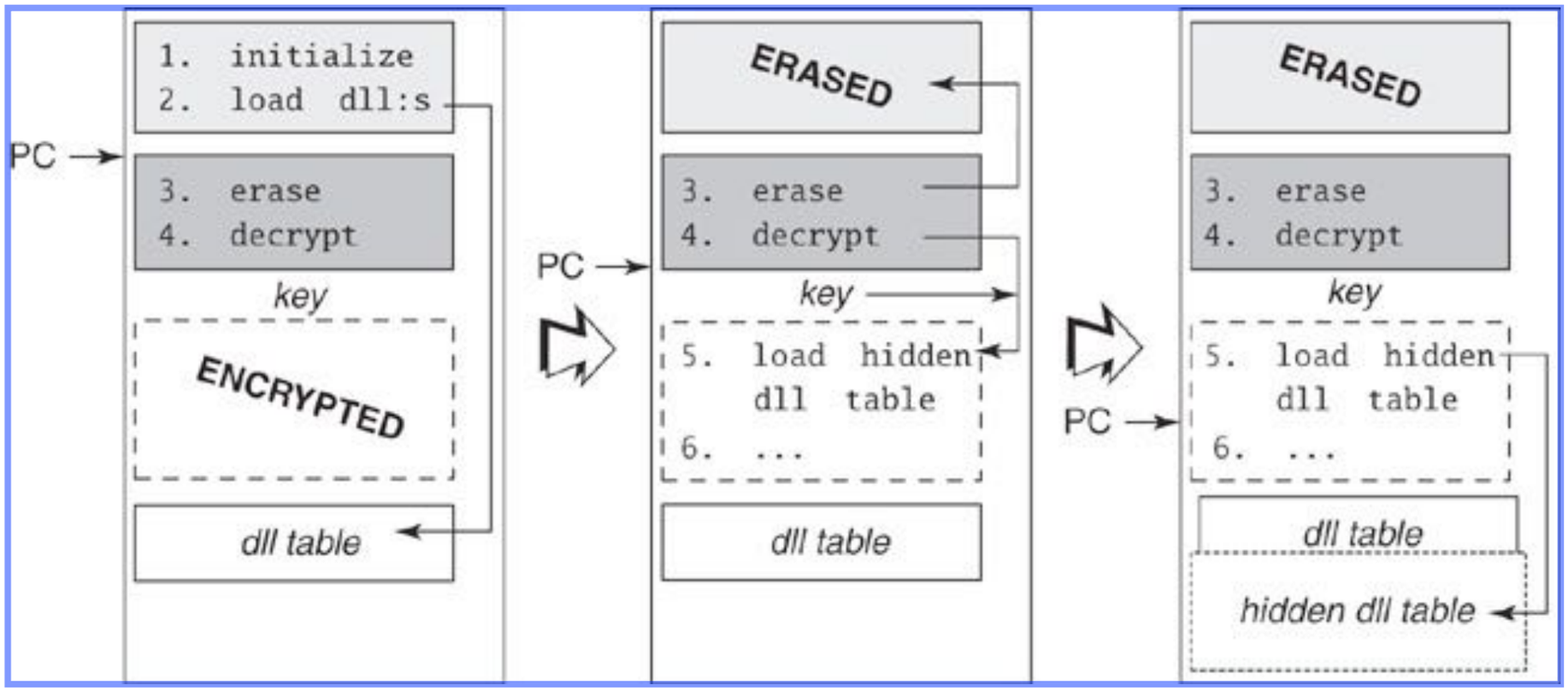
The case of Skype

Binary Packing

- ✓The Skype client is heavily tamperproof and obfuscated to avoid static disassembly
- ✓Some parts of the binary are xored by a hard-coded RSA key
- ✓In memory, Skype is fully encrypted

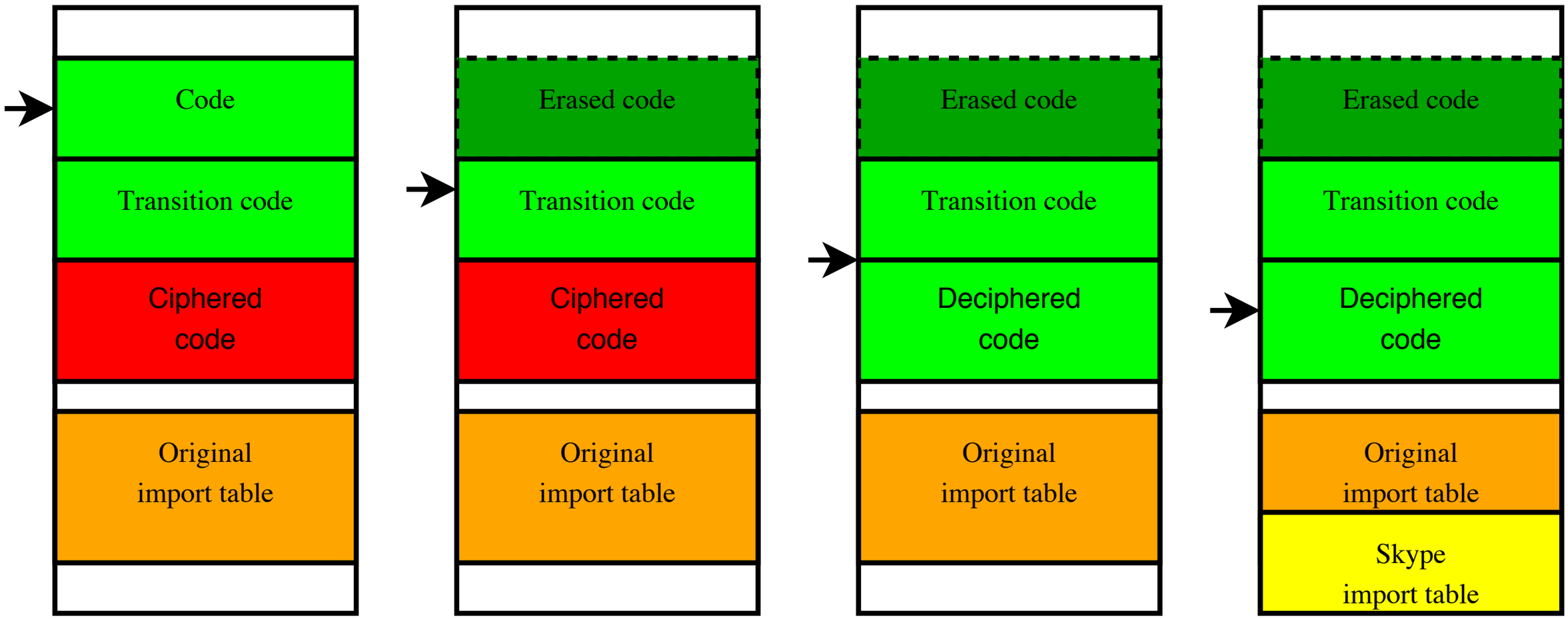


Anti-dumping



- ✓ The client starts performing initializations and loading any necessary dll
- ✓ The program erases the initialization code
- ✓ The program deciphers encrypted areas, the executable itself contains the decryption key (decryption is XOR with the key)
- ✓ When the decrypted code is executed it loads an hidden dll table, erasing part of the original dll table: the client loads 843 dll, but 169 of these are not included in te original dll table.
- ✓ For the dll dynamically loaded and the erased initialization code it is difficult for an attacker to create a new binary by writing out the memory image to a file

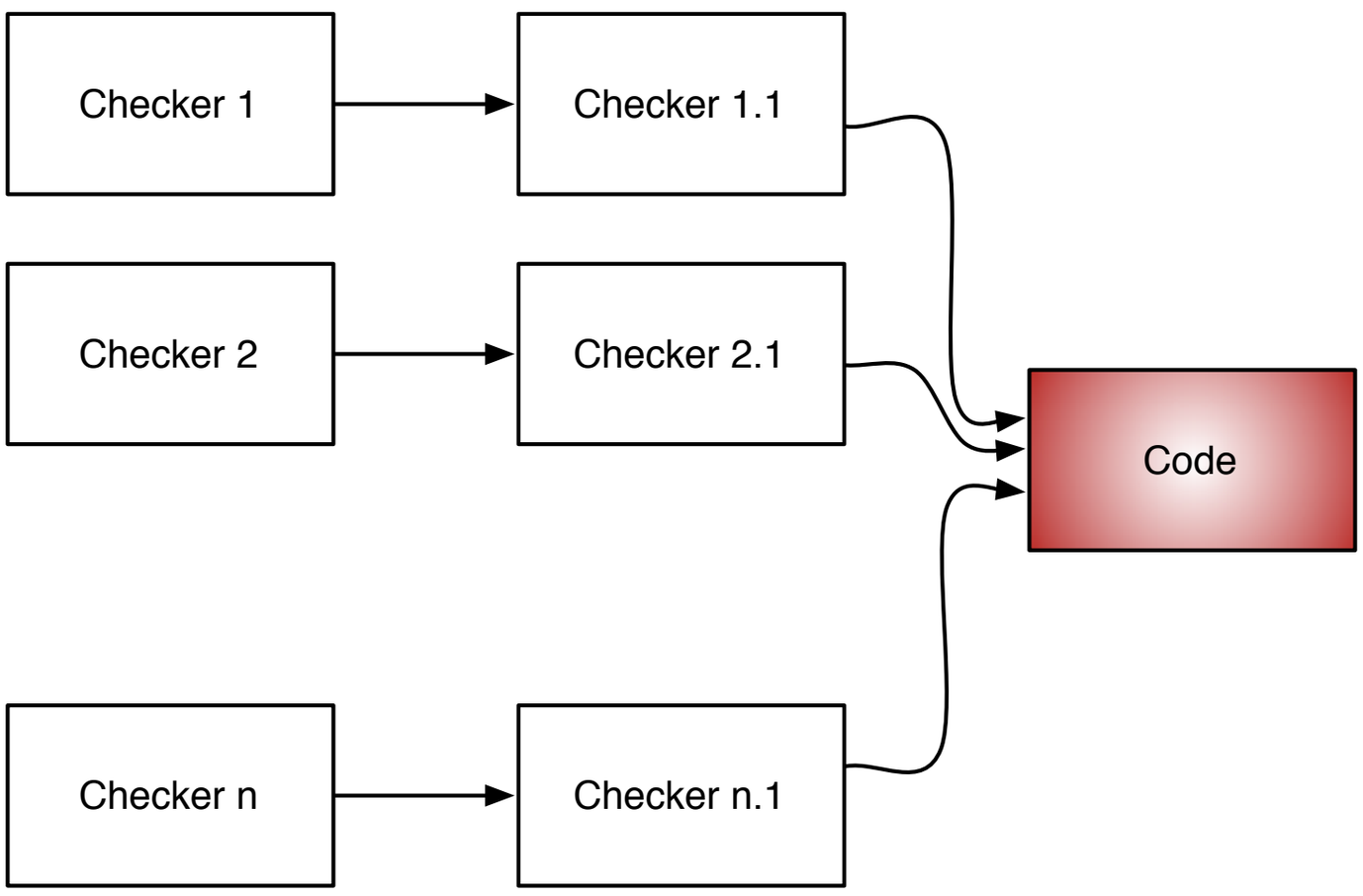
Anti-dumping



- ✓ The client continues by checking for the presence of debuggers
  - ✓ Signatures of known debuggers
  - ✓ Timing test

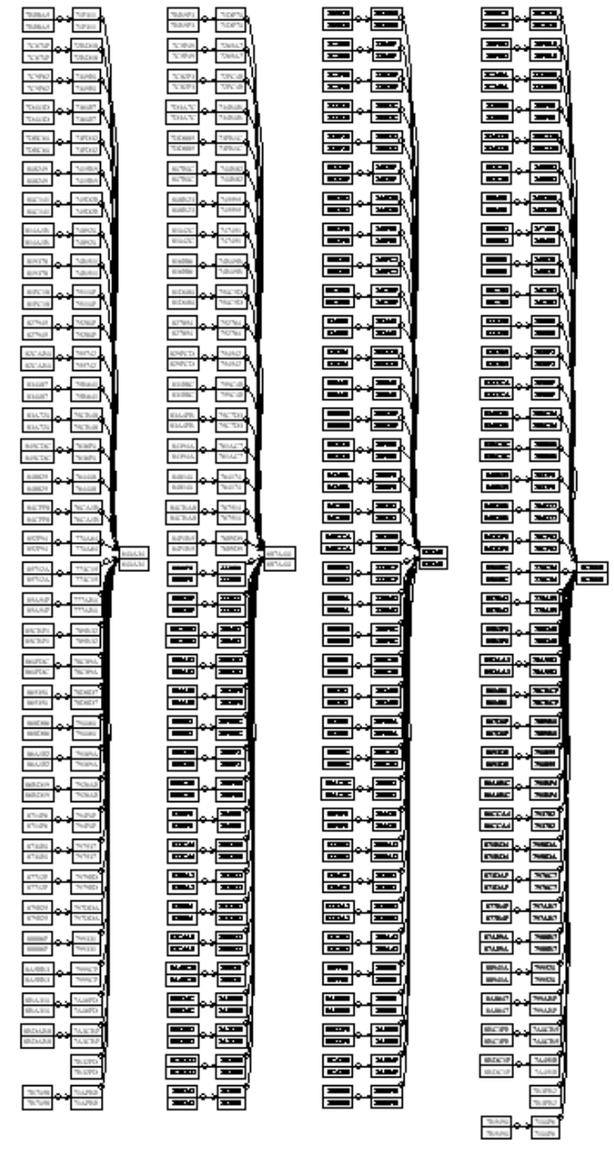
Code Integrity

- ✓In the tamper-proofing stage a network of nearly 300 hash functions checks the client code and also each other
- ✓Each rectangle represents a hash (checksum)
- ✓An arrow represents the link checker/checked
- ✓The hash computes the **next instruction**
- ✓No self repair it simply crashes



Code Integrity

- ✓Each checksumer is a bit different: they *seem* to be **polymorphic**
- ✓They are executed **randomly**
- ✓The pointers initialization is **obfuscated** with computations
- ✓Checksum operator is **randomized** (add, xor, sub, ...)
- ✓Checksumer length is **random**
- ✓Dummy mnemonics are inserted
- ✓Final test is not trivial: it can use final checksum to compute a pointer for next code part.



How to attack?

Attacker goal: build its own binary complete with the RSA key

The first steps of the attack are:

- STEP-1: Find the keys stored in the binary and decrypt the encrypted sections
- STEP-2: Read the hidden dll table and combine it with the original one, making a complete table
- STEP-3: Build a script that finds the beginning and end of each hash function, identify the hash function is possible since: they are composed of:
  - A pointer initialization
  - A loop
  - A lookup
  - A test/computation

We can build a script that spots such code

How to attack?

✓Put a breakpoint on each hash in order to:

- Collect all the values computed by the hash functions during a run of the program
- Replace the body of the function with that value

BUT

✓Software breakpoints change the checksums by replacing an instruction with a trap

✓We only have 4 hardware breakpoints

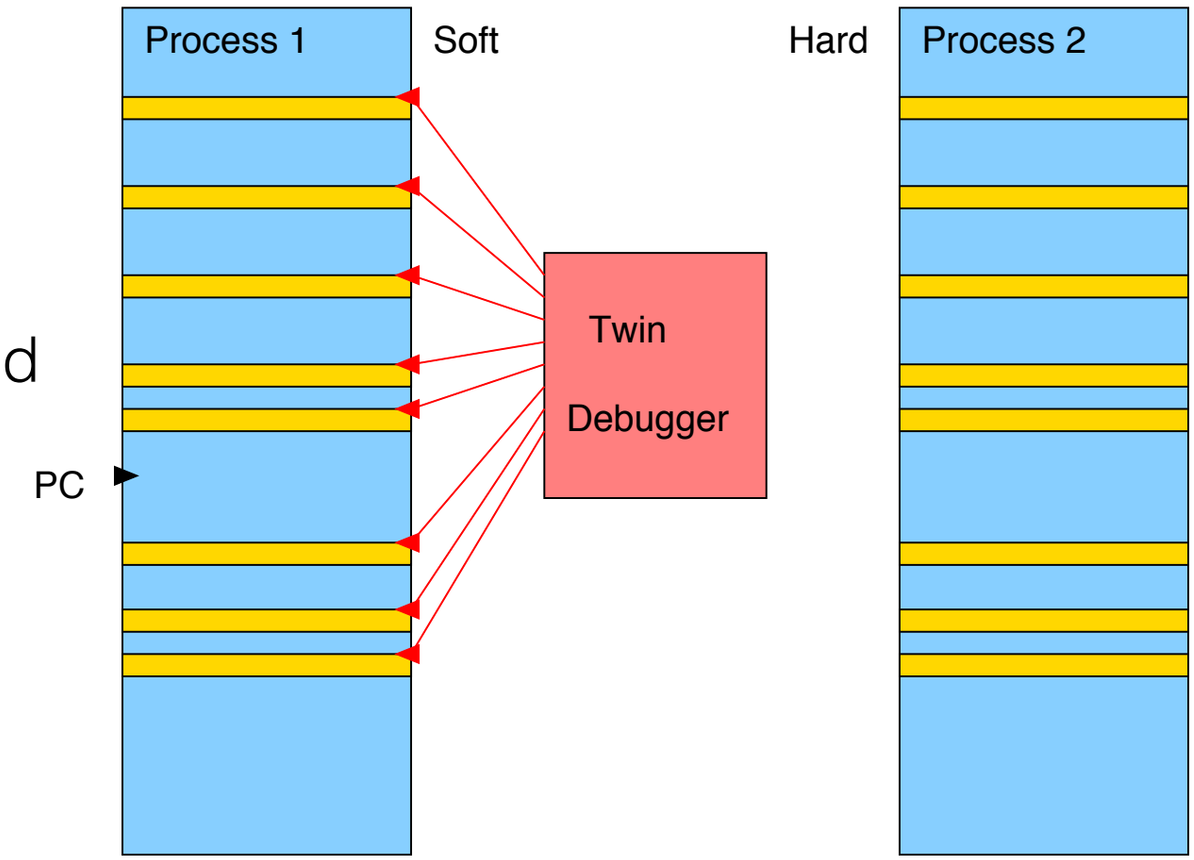
✓⇒ **Twin processes** debugging

✓We run two Skype processes in parallel, both processes are under debugging and one uses hw breakpoints and the other sw breakpoints

How to attack?

- ✓ STEP-4: run Skype to collect the values computed by all the hash functions

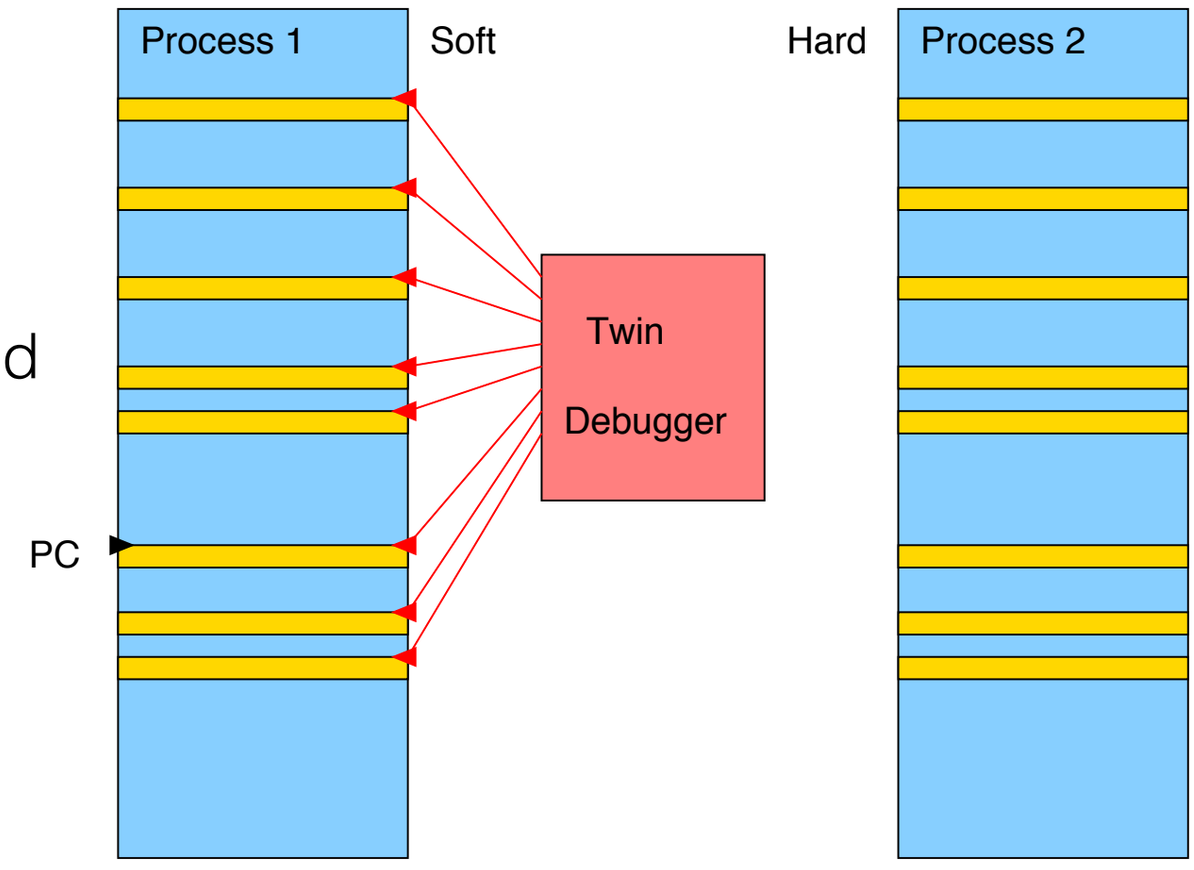
1. Put software breakpoints on every checksumer of process P_{soft}
2. Start the twin process P_{hrad}
3. Run P_{soft} until it reaches a breakpoint
4. Put 2 hardware breakpoints before and after the checksumer of P_{hrad}
5. Use P_{hrad} to compute the checksum value
6. Write it down in H
7. Restart P_{soft} starting at the end of the checksumer and with the return value of the hash set to H
8. Go to point 3



How to attack?

- ✓ STEP-4: run Skype to collect the values computed by all the hash functions

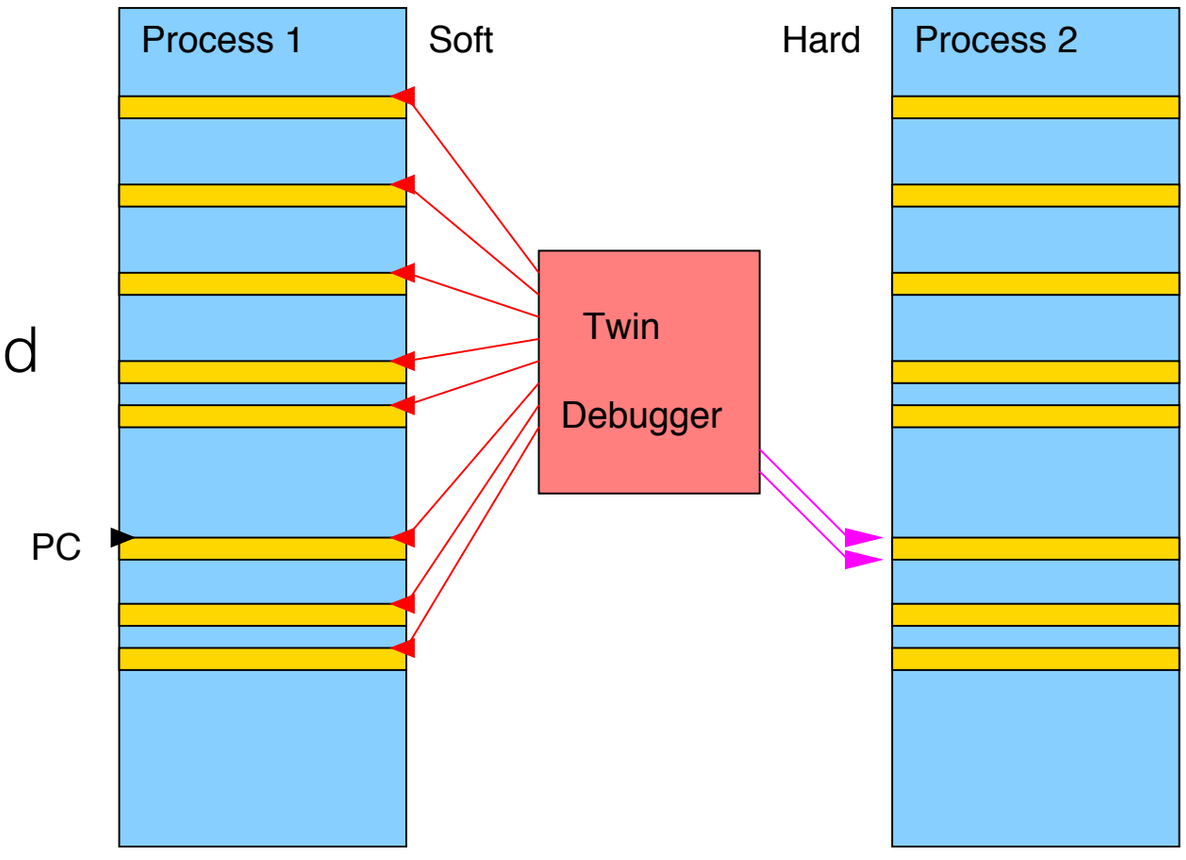
1. Put software breakpoints on every checksumer of process P_{soft}
2. Start the twin process P_{hrad}
3. Run P_{soft} until it reaches a breakpoint
4. Put 2 hardware breakpoints before and after the checksumer of P_{hrad}
5. Use P_{hrad} to compute the checksum value
6. Write it down in H
7. Restart P_{soft} starting at the end of the checksumer and with the return value of the hash set to H
8. Go to point 3



How to attack?

- ✓ STEP-4: run Skype to collect the values computed by all the hash functions

1. Put software breakpoints on every checksumer of process P_{soft}
2. Start the twin process P_{hrad}
3. Run P_{soft} until it reaches a breakpoint
4. Put 2 hardware breakpoints before and after the checksumer of P_{hrad}
5. Use P_{hrad} to compute the checksum value
6. Write it down in H
7. Restart P_{soft} starting at the end of the checksumer and with the return value of the hash set to H
8. Go to point 3



Anti-debugging

- ✓Skype has some protections against debuggers!
- ✓Anti Softice: checks if the drivers of Softice have been loaded
- ✓Generic anti-debugger: The hash system spots software breakpoints as they change the integrity of the binary
- ✓Skype does timing measures in order to check if the process is debugged or not