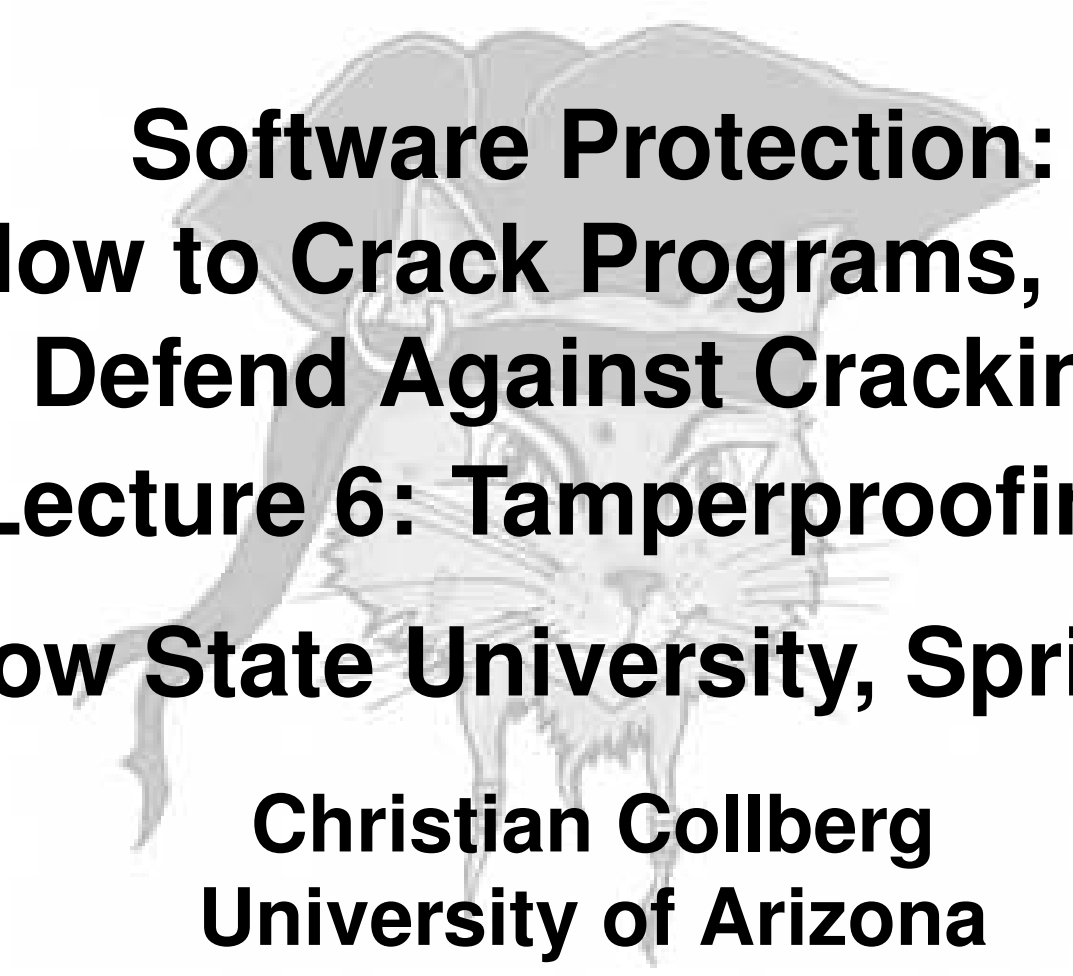


Software Protection: How to Crack Programs, and Defend Against Cracking

Lecture 6: Tamperproofing I

Moscow State University, Spring 2014

Christian Collberg
University of Arizona

www.cs.arizona.edu/~collberg

Today's lecture

1. Tamperproofing

Introduction

What is tamperproofing?

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

- A tamperproofing algorithm
  1. makes tampering difficult

What is tamperproofing?

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

- A tamperproofing algorithm
  1. makes tampering difficult
  2. detects when tampering has occured

What is tamperproofing?

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

- A tamperproofing algorithm
  1. makes tampering difficult
  2. detects when tampering has occured
  3. responds to the attack

What are typical attacks and defenses?

An attacker typically modifies the program with the intent to force it to chose a different execution path than the programmer intended:

1. remove code from and/or insert new code into the executable file prior to execution;

What are typical attacks and defenses?

An attacker typically modifies the program with the intent to force it to chose a different execution path than the programmer intended:

1. remove code from and/or insert new code into the executable file prior to execution;
2. remove code from and/or insert new code into the running program;

What are typical attacks and defenses?

An attacker typically modifies the program with the intent to force it to chose a different execution path than the programmer intended:

1. remove code from and/or insert new code into the executable file prior to execution;
2. remove code from and/or insert new code into the running program;
3. affect the runtime behavior of the program through external agents such as emulators, debuggers, or a hostile operating system.

Algorithms

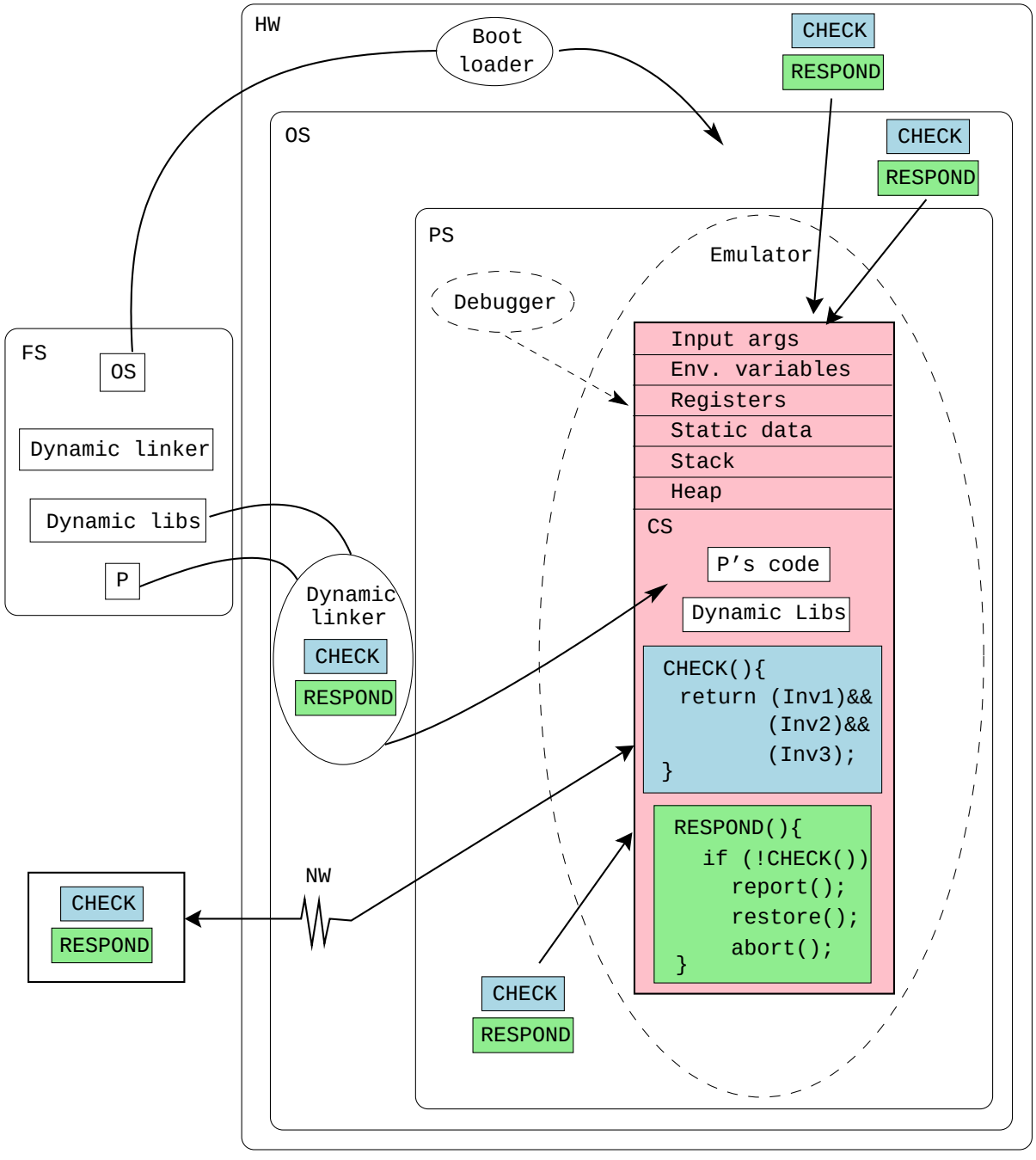
1. introspection, i.e. tamperproofed programs which monitor their own code to detect modifications.

Algorithms

1. introspection, i.e. tamperproofed programs which monitor their own code to detect modifications.
2. various kinds of response mechanisms.

Algorithms

1. introspection, i.e. tamperproofed programs which monitor their own code to detect modifications.
2. various kinds of response mechanisms.
3. oblivious hashing algorithms which examine the *state* of the program for signs of tampering.


HW
Boot loader
CHECK
RESPOND
OS
CHECK
RESPOND
PS
Emulator
Debugger
FS
OS
Dynamic linker
Dynamic libs
P
Input args
Env. variables
Registers
Static data
Stack
Heap
CS
P's code
Dynamic Libs
CHECK(){
return (Inv1)&&
(Inv2)&&
(Inv3);
}
RESPOND(){
if (!CHECK())
report();
restore();
abort();
}
Dynamic linker
CHECK
RESPOND
NW
CHECK
RESPOND
CHECK
RESPOND

How does the adversary attack `P`?

1. Modify files:
   - `P`'s executable file
   - dynamic linker
   - dynamic libraries

How does the adversary attack `P`?

1. Modify files:
   - `P`'s executable file
   - dynamic linker
   - dynamic libraries
2. Modify the operating system

How does the adversary attack `P`?

1. Modify files:
   - `P`'s executable file
   - dynamic linker
   - dynamic libraries
2. Modify the operating system
3. Run `P` under emulation

How does the adversary attack `P`?

1. Modify files:
   - `P`'s executable file
   - dynamic linker
   - dynamic libraries
2. Modify the operating system
3. Run `P` under emulation
4. Modify `P` while running under debugging

What do we want?

Ensure `P` is healthy and the environment isn't hostile:

1. Unadulterated hardware and operating system

What do we want?

Ensure `P` is healthy and the environment isn't hostile:

1. Unadulterated hardware and operating system
2. Unmodified `P`'s code

What do we want?

Ensure `P` is healthy and the environment isn't hostile:

1. Unadulterated hardware and operating system
2. Unmodified `P`'s code
3. Not running under emulation

What do we want?

Ensure `P` is healthy and the environment isn't hostile:

1. Unadulterated hardware and operating system
2. Unmodified `P`'s code
3. Not running under emulation
4. Not being modified by a debugger

What do we want?

Ensure `P` is healthy and the environment isn't hostile:

1. Unadulterated hardware and operating system
2. Unmodified `P`'s code
3. Not running under emulation
4. Not being modified by a debugger
5. The right dynamic libraries have been loaded

Checking for tampering — code checking

- Check that `P`'s code hashes to a known value:

```
if (hash(P's code) != 0xca7ca115)
   return false;
```

How do we respond to tampering?

1. Terminate the program.

How do we respond to tampering?

1. Terminate the program.
2. Restore the program to its correct state, by patching the tampered code.

How do we respond to tampering?

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.

How do we respond to tampering?

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.

How do we respond to tampering?

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”.

How do we respond to tampering?

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.

Checking the Environment

Tampering with the environment

Hack libc!

Emulate! **Hack OS!**

Debug! **Hack HW!**

Tampering with the environment

libc OK?

Emulated? **OS OK?**

Debugged? **HW OK?**

Hack libc!

Emulate! **Hack OS!**

Debug! **Hack HW!**

Checking the Environment

- “Am I being run under emulation?”

Checking the Environment

- “Am I being run under emulation?’
- “Are common attack tools installed?”

Checking the Environment

- “Am I being run under emulation?’
- “Are common attack tools installed?”
- “Are common attack tools running?”

Checking the Environment

- “Am I being run under emulation?’
- “Are common attack tools installed?”
- “Are common attack tools running?”
- “Is a debugger attached to my process?”

Checking the Environment

- “Am I being run under emulation?’
- “Are common attack tools installed?”
- “Are common attack tools running?”
- “Is a debugger attached to my process?”
- “Is the operating system at the proper patch level?”

Checking the Environment

- “Am I being run under emulation?’
- “Are common attack tools installed?”
- “Are common attack tools running?”
- “Is a debugger attached to my process?”
- “Is the operating system at the proper patch level?”
- “Are the right dynamic libraries being loaded?”

Does the User have a Debugger?

- Check if a program named gdb exists:

```
> find /usr -name gbd -print
```

- See if a program named gdb is running:

```
> ps -ax | grep gdb
12233 ttys000    0:00.02 gdb /bin/ls
```

- See if gdb exists, under a different name.

Am I Already Being Debugged?

- On Linux, you can only debug/trace a program once.
- Start *P* under debugging, if you fail, *P* is already being debugged!

```
#include <sys/ptrace.h>
int main() {
   if (ptrace(PTRACE_TRACEME))
       printf("I'm being traced!\n");
}
```

```
> gcc -g -o traced traced.c
> traced
> gdb traced
(gdb) run
I'm being traced!
```

Am I Running Unusually Slow?

- Certain operations, such as catching exceptions, are slower when being run under debugging.

```
#include <stdio.h>
#include <stdint.h>
#include <signal.h>
#include <unistd.h>
#include <setjmp.h>

jmp_buf env;

void handler(int signal) {
   longjmp(env,1);
}
```

```
int main() {
    signal(SIGFPE, handler);
    uint32_t start,stop;
    int x = 0;
    if (setjmp(env) == 0) {
        asm volatile (
            "cpuid\n"
            "rdtsc\n" : "=a" (start)
         );
        x = x/x;
    } else {
        asm volatile (
           "cpuid\n"
           "rdtsc\n" : "=a" (stop)
        );
        uint32_t elapsed = stop - start;
        if (elapsed>40000) printf("Debugged!\n");
        else               printf("Not debugged!\n");
    }
}
```

Am I Running Unusually Slow?

- Here's the output when run normally:

```
> gcc -o cycles cycles.c
> cycles
elapsed 31528: Not debugged!
```

- Here's the output when under a debugger:

```
> gcc -o cycles cycles.c
> gdb cycles
(gdb) handle SIGFPE noprint nostop
(gdb) run
elapsed 79272: Debugged!
```

Introspection

Checking by introspection

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

Checking by introspection

- Augment the program with functions that compute a hash over a code region to compare to an expected value.
- How can we be sure that the attacker won't tamper with the hash computation itself?

Checking by introspection

- Augment the program with functions that compute a hash over a code region to compare to an expected value.
- 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, so that checkers can check each other and responders can repair code that has been tampered with.

Checking by introspection

- Augment the program with functions that compute a hash over a code region to compare to an expected value.
- 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, so that checkers can check each other and responders can repair code that has been tampered with.
  2. hide the hash values so they won't give away the location of the checkers.

Checking by introspection

- Augment the program with functions that compute a hash over a code region to compare to an expected value.
- 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, so that checkers can check each other and responders can repair code that has been tampered with.
  2. hide the hash values so they won't give away the location of the checkers.
- We'll see a clever attack on all introspection algorithms!

Inserting Guards

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

Attack model — Find the guards

1. Search for patterns in the static code, for example two code segment addresses followed by a test:

```
start = 0xbabebabe;
end   = 0xca75ca75;
while (start < end) {
```

2. Search for patterns in the execution, such as data reads into the code.

Attack model — Disable the guards

1. Replace the if-statement by `if (0)...`:

```
if (0)
    abort();
```

2. Pre-compute the hash value and substitute it into the response code:

```
goto *expected_value;
```

Chang & Atallah

- Invented by two Purdue University researchers, Mike Atallah and Hoi Chang:

- Patented and with assistance from Purdue a start-up, Arxan, was spun off.

Chang & Atallah

- Checkers compute a hash over a region and compare to the expected value.

Chang & Atallah

- Checkers compute a hash over a region and compare to the expected value.
- Checkers check the code and check each other as well!

Chang & Atallah

- 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.

Chang & Atallah

- 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!

Chang & Atallah

- 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.
- Multiple checkers can check the same region.

Chang & Atallah

- 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.
- Multiple checkers can check the same region.
- Multiple responders can repair a tampered region.

```
int main (int argc, char *argv[]) {
   int user_key = 0xca7ca115;
   int media[] = {10,102};
   play(user_key,media,2);
}
int getkey(int user_key) {
   int player_key = 0xbabeca75;
   return user_key ^ player_key;
}
int decrypt(int user_key, int media) {
   int key = getkey(user_key);
   return media ^ key;
}
float decode (int digital) {return (float)digital;}
void play(int user_key, int media[], int len) {
   int i;
   for(i=0;i<len;i++)
      printf("%f\n",decode(decrypt(user_key,media[i])));
}
```

```
#define getkeyHASH 0xce1d400a
#define getkeySIZE 14
uint32 getkeyCOPY[] =
   {0x83e58955,0x72b820ec,0xc7080486,...};

#define decryptHASH 0x3764e45c
#define decryptSIZE 16
uint32 decryptCOPY[] =
   {0x83e58955,0xaeb820ec,0xc7080486,...};

#define playHASH 0x4f4205a5
#define playSIZE 29
uint32 playCOPY[] =
   {0x83e58955,0xedb828ec,0xc7080486,...};
```

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


   A();
}

int A() {


   B();
}

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

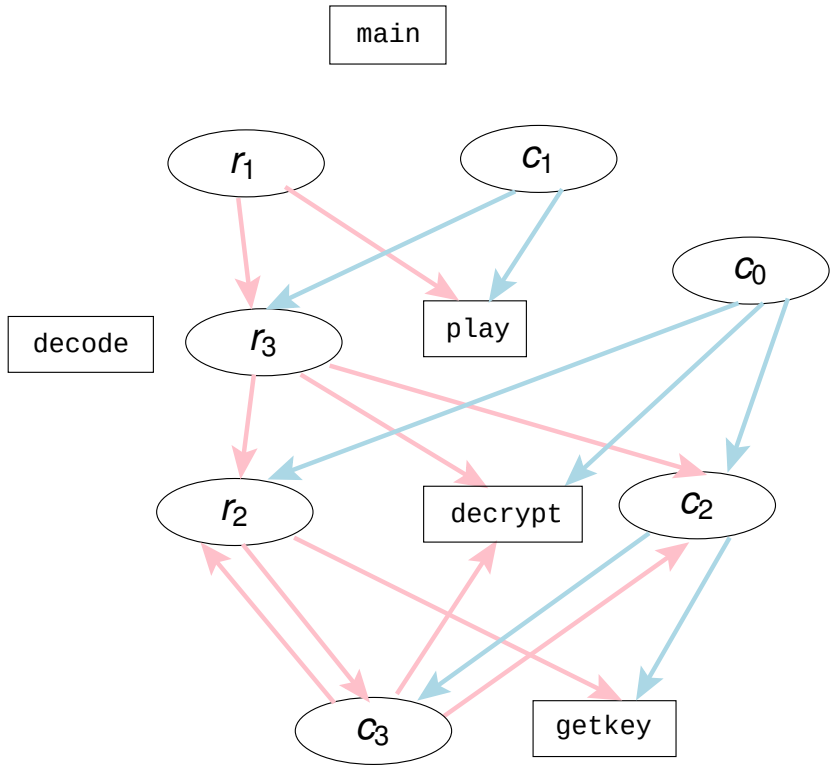
```
uint32 B_COPY[]={0x83e58955,0xaeb820ec,0xc7080486,...};

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



   A();
}

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

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

```
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() {
    ...
}
```

```
uint32 getkeyCOPY[] ={0x83e58955,0x72b820ec,0xc7080486,...};
uint32 decryptCOPY[]={0x83e58955,0xaeb820ec,0xc7080486,...};
uint32 playCOPY[]   ={0x83e58955,0xedb828ec,0xc7080486,...};
uint32 decryptVal;

int main (int argc, char *argv[]) {
    uint32 playVal = hash((waddr_t)play,29);
    int user_key = 0xca7ca115;
    decryptVal = hash((waddr_t)decrypt,16);
    int media[] = {10,102};
    if (playVal != 0x4f4205a5)
        memcpy((waddr_t)play,playCOPY,29*sizeof(uint32));
    play(user_key,media,2);
}

int getkey(int user_key) {
    decryptVal = hash((waddr_t)decrypt,16);
    int player_key = 0xbabeca75;
    return user_key ^ player_key;
}
```

```
int decrypt(int user_key, int media) {
    uint32 getkeyVal = hash((waddr_t)getkey,14);
    if (getkeyVal != 0xce1d400a)
        memcpy((waddr_t)getkey,getkeyCOPY,14*sizeof(uint32));
    int key = getkey(user_key);
    return media ^ key;
}

float decode (int digital) {
    return (float)digital;
}
void play(int user_key, int media[], int len) {
    if (decryptVal != 0x3764e45c)
        memcpy((waddr_t)decrypt,decryptCOPY,16*sizeof(uint32));
    int i;
    for(i=0;i<len;i++)
       printf("%f\n",decode(decrypt(user_key,media[i])));
}
```

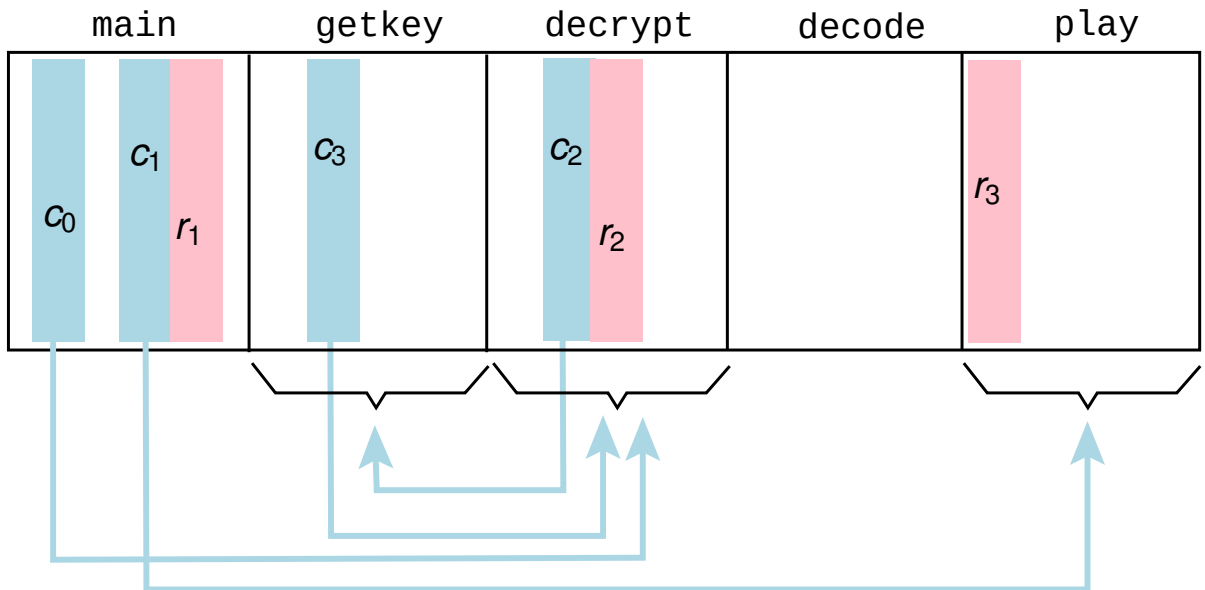
Checker network



- code — code blocks
- c_i — checkers
- r_i — repairers

Checker Network

Here's the corresponding code, as it is laid out in memory:



blue represent checkers, pink repairers.

Generating hash functions

Generating hash functions

- Prevent collusive attacks $\Rightarrow$ generate a large number of different-looking hash functions.

Generating hash functions

- Prevent collusive attacks $\Rightarrow$ generate a large number of different-looking hash functions.
- Self-collusive attacks = the adversary scans through the program for pieces of similar-looking code.

Generating hash functions

- Prevent collusive attacks $\Rightarrow$ generate a large number of different-looking hash functions.
- Self-collusive attacks = the adversary scans through the program for pieces of similar-looking code.
- No need to be “cryptographically secure”.

Generating hash functions

- Prevent collusive attacks $\Rightarrow$ generate a large number of different-looking hash functions.
- Self-collusive attacks = the adversary scans through the program for pieces of similar-looking code.
- No need to be “cryptographically secure”.
- No need to generate a uniform distribution of values.

Generating hash functions

- Prevent collusive attacks $\Rightarrow$ generate a large number of different-looking hash functions.
- Self-collusive attacks = the adversary scans through the program for pieces of similar-looking code.
- No need to be “cryptographically secure”.
- No need to generate a uniform distribution of values.
- Must be simple, fast, stealthy!

hash1

```
typedef unsigned int uint32;
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;
}
```

- Inline the function for better stealth.

hash2

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

- Will the compiler generate different code than for `hash1`???

hash3

```
int32 hash3 (addr_t start,addr_t end,int step) {
   uint32 h = *start;
   while(1) {
      start+=step;
      if (start>=end) return h;
      h ^= *start;
   }
}
```

- Step through the code region in more or less detail $\Rightarrow$ balance performance and accuracy.

hash4

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

- Scan backwards.
- Obfuscate to prevent pattern-matching attacks: add (and then subtract out) a random value (`rnd`).

hash5

```
uint32 hash5 (addr_t start, addr_t end, uint32 C) {
   uint32 h = 0;
   while (start < end) {
      h = C*(*start + h);
      start++;
   }
   return h;
}
```

Obfuscating hash5

- Generate 2,916,864 variants, each less than 50 bytes of x86!
- Reorder basic blocks, invert conditional branches. . .
- Replace multiplication instructions by combinations of shifts adds, and address computations. . .
- Permute instructions within blocks. . .
- Permute register assignments. . .
- Replace instructions with equivalents. . .

The Skype obfuscated protocol

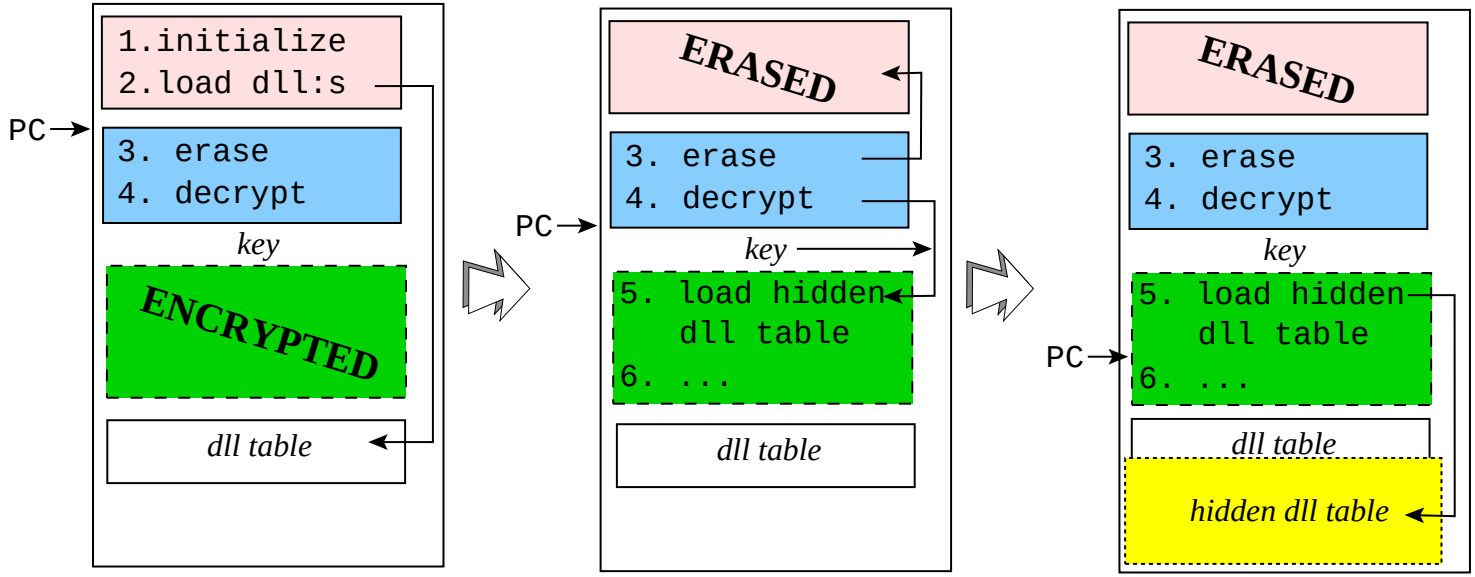
The Skype obfuscated protocol

- Voice-over-IP service where users are charged for computer-to-phone and phone-to-computer calls.
- The Skype client is heavily tamperproofed and obfuscated.
- 2005: Skype was bought by eBay for $2.6 billion.
- 2006: Hacked by two researchers at the EADS Corporate Research Center in France.

The Skype obfuscated protocol

- The client binary contains:
  1. hardcoded RSA keys
  2. the IP address and port number of a known server
- Break the protection and build your own VoIP network!

Skype protection: Stage 1



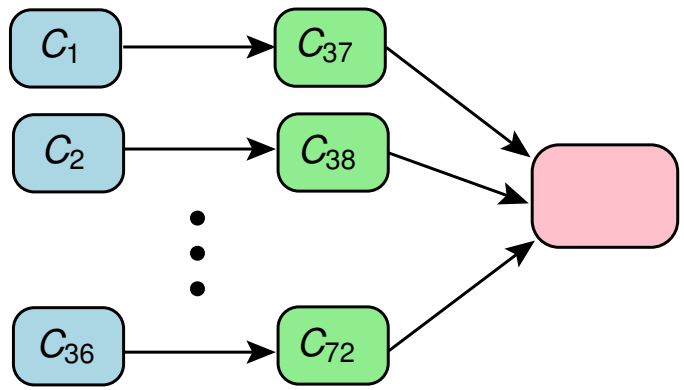
- pink: cleartext code, loads dlls.
- blue: erase pink code, decrypts green code.
- green: loads hidden dlls (yellow).
- Erasing and hiding dlls: hard to recreate binary.

Skype protection: Stage 2

- Check for debuggers:
  1. Signatures of known debuggers
  2. Timing tests

Skype protection: Stage 3

- Checker network:



- Hash function computes the address of the next location to be executed!
- Hash functions are obfuscated, but not enough — attacked by pattern-matching.

```
uint32 hash7() {
   addr_t addr;
   addr = (addr_t)((uint32)addr^(uint32)addr);
   addr = (addr_t)((uint32)addr + 0x688E5C);
   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 ⊕ 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;
}
```

Other obfuscations

- All function calls are done indirectly.
- Insert dummy code protected by opaque predicates.
- The code raises bogus exceptions, the exception handler repairs register values, returns back to the original location.

Attacking the Skype Client

1. Goal: build your own Skype binary!
2. Goal: insert your own RSA keys!
3. First: remove the encryption and tamperproofing

Attacking the Skype Client

1. Find the keys stored in the binary and decrypt the encrypted sections.
2. Read the hidden dll table and combine it with the original one, making a complete table.
3. Build a script which runs over the decrypted binary and finds beginning/end of every hash function.

Finding the Hash Functions

1. Distinctive structure: initialize, loop, read memory, compute hash.
2. Step 1: Use simple pattern matching to find all functions.

```
uint32 hash7() {
   addr_t addr;
   addr = (addr_t)((uint32)addr^(uint32)addr);
   addr = (addr_t)((uint32)addr + 0x688E5C);
   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 ⊕ 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;
}
```

Finding the Hash Values

1. Step 2: Run every hash function, collect their output values.
2. Step 3: Replace the body of the function with that value.

Running the hash functions

- Try 1: Set software breakpoints on every function header!

Running the hash functions

- Try 1: Set software breakpoints on every function header!
- Nope: software breakpoints change the executable!

Running the hash functions

- Try 1: Set software breakpoints on every function header!
- Nope: software breakpoints change the executable!
- Try 2: Set hardware breakpoints on every function header!

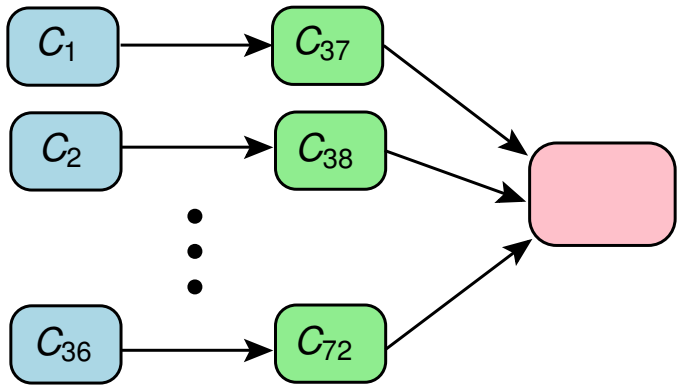
Running the hash functions

- Try 1: Set software breakpoints on every function header!
- Nope: software breakpoints change the executable!
- Try 2: Set hardware breakpoints on every function header!
- Nope: only 4 hardware breakpoints, and $>$ 300 functions!

Running the hash functions

- Try 3: run Skype twice, in parallel, both processes under debugging, but one using hardware breakpoints, the other software breakpoints.
- See next slide.
- Alternative attack: run each function in an emulator

Twin Processes Debugging



> **gdb** S_{soft}

> **break** c_1
> **break** c_2

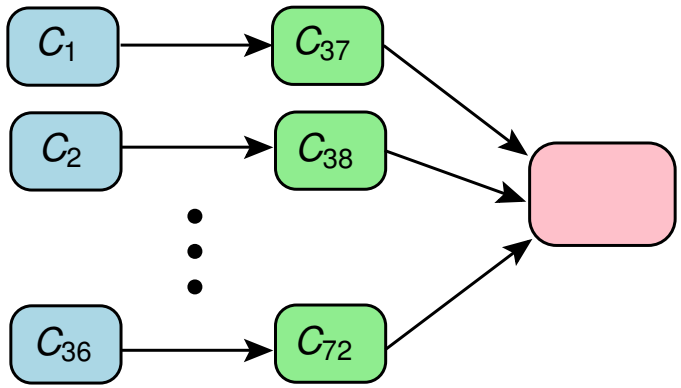
> **...**

> **run**

Break on c_{20}**!**

> **gdb** S_{hard}

Twin Processes Debugging



> gdb S_{soft}

> break c_1
> break c_2

> ...

> run

Break on c_{20}!

> gdb S_{hard}

> hbreak start of c_{20}
> hbreak end of c_{20}

> run

Break on c_{20}!

Record hash value!!

1. Start one Skype process S_{soft}, setting software breakpoints at the beginning of every hash function.

1. Start one Skype process S_{soft}, setting software breakpoints at the beginning of every hash function.
2. Start another Skype process S_{hard}.

1. Start one Skype process S_{soft}, setting software breakpoints at the beginning of every hash function.
2. Start another Skype process S_{hard}.
3. Run S_{soft} until a breakpoint at the beginning of a hash function is reached at some address *start*.

1. Start one Skype process S_{soft}, setting software breakpoints at the beginning of every hash function.
2. Start another Skype process S_{hard}.
3. Run S_{soft} until a breakpoint at the beginning of a hash function is reached at some address *start*.
4. Set a hardware breakpoint at *start* in the S_{hard} and at address *end*.

1. Start one Skype process S_{soft}, setting software breakpoints at the beginning of every hash function.
2. Start another Skype process S_{hard}.
3. Run S_{soft} until a breakpoint at the beginning of a hash function is reached at some address *start*.
4. Set a hardware breakpoint at *start* in the S_{hard} and at address *end*.
5. Run S_{hard} until *end* is reached.

1. Start one Skype process S_{soft}, setting software breakpoints at the beginning of every hash function.
2. Start another Skype process S_{hard}.
3. Run S_{soft} until a breakpoint at the beginning of a hash function is reached at some address *start*.
4. Set a hardware breakpoint at *start* in the S_{hard} and at address *end*.
5. Run S_{hard} until *end* is reached.
6. Record the result *hash* of the hash computation.

1. Start one Skype process S_{soft}, setting software breakpoints at the beginning of every hash function.
2. Start another Skype process S_{hard}.
3. Run S_{soft} until a breakpoint at the beginning of a hash function is reached at some address *start*.
4. Set a hardware breakpoint at *start* in the S_{hard} and at address *end*.
5. Run S_{hard} until *end* is reached.
6. Record the result *hash* of the hash computation.
7. Restart S_{soft} starting at address *end* and with the return value of the hash function set to *hash*.

Hiding hash values

Unstealthy constants

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

- Attack: scan the program for code that appears to compare a computed hash against a (weird) expected value.
- Also, collusive attacks!

Simple fix. . .

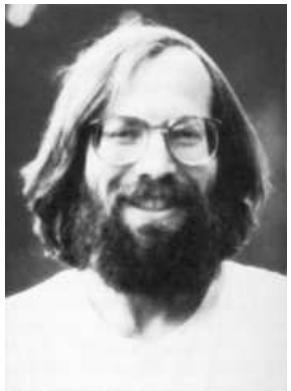
```
h1 = hash(orig_start,orig_end);
h2 = hash(copy_start,copy_end);
if (h1 != h2) abort();
```

- Add a copy of every region you're hashing to the program.
- In the worst case, your program has now doubled in size!
- $f() = f()$ may not be all that common in real code.

Algorithm

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

- Hide the constants by constructing a hash function that (unless the code has been hacked) always hashes to zero!
- Code is more natural — no weird constants!
- Invented by Bob Tarjan and others at InterTrust:



hash5

```
uint32 hash5 (addr_t start, addr_t end, uint32 C) {
   uint32 h = 0;
   while (start < end) {
      h = C*(*start + h);
      start++;
   }
   return h;
}
```

- uses the `hash5` hash function.

Algorithm

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

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

- `hash5` is invertible.
- Insert an empty slot (a 32-bit word) within the region you're protecting, and later give this slot a value that makes the region hash to zero.

Algorithm

TAMPERPROOF(P, n):

1. Insert n checkers of the form

```
if (hash(start,end)) RESPOND
```

randomly throughout the program.

Algorithm

TAMPERPROOF(P, n):

1. Insert n checkers of the form

```
if (hash(start,end)) RESPOND
```

   randomly throughout the program.

2. Randomize the placement of basic blocks.

Algorithm

TAMPERPROOF(P, n):

1. Insert n checkers of the form

   ```
   if (hash(start,end)) RESPOND
   ```

   randomly throughout the program.
2. Randomize the placement of basic blocks.
3. Insert at least n corrector slots $c_1, \ldots, c_n$.

Algorithm

TAMPERPROOF(P, n):

1. Insert n checkers of the form

   ```
   if (hash(start,end)) RESPOND
   ```

   randomly throughout the program.
2. Randomize the placement of basic blocks.
3. Insert at least n corrector slots $c_1, \ldots, c_n$.
4. Compute n overlapping regions $I_1, \ldots, I_n$, each I_i associated with one corrector c_i.

Algorithm

TAMPERPROOF(P, n):

1. Insert n checkers of the form

   ```
   if (hash(start,end)) RESPOND
   ```

   randomly throughout the program.
2. Randomize the placement of basic blocks.
3. Insert at least n corrector slots $c_1, \ldots, c_n$.
4. Compute n overlapping regions $I_1, \ldots, I_n$, each I_i associated with one corrector c_i.
5. Associate each checker with a region I_i and set c_i such that I_i hashes to zero.

System design

- describes a complete and practical system for doing tamperproofing and fingerprinting.

System design

- describes a complete and practical system for doing tamperproofing and fingerprinting.
- When during the translation and installation process do you insert fingerprints and tamperproofing code?
  1. At the source code level before compilation?

System design

- describes a complete and practical system for doing tamperproofing and fingerprinting.
- When during the translation and installation process do you insert fingerprints and tamperproofing code?
  1. At the source code level before compilation?
  2. At the binary code level post link time?

System design

- describes a complete and practical system for doing tamperproofing and fingerprinting.
- When during the translation and installation process do you insert fingerprints and tamperproofing code?
  1. At the source code level before compilation?
  2. At the binary code level post link time?
  3. During installation on the end user's site?

System design

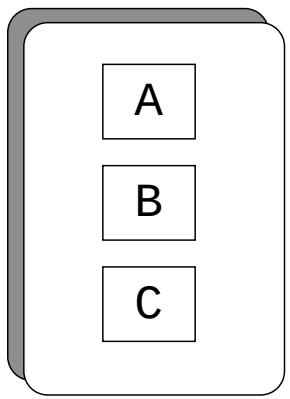
- describes a complete and practical system for doing tamperproofing and fingerprinting.
- When during the translation and installation process do you insert fingerprints and tamperproofing code?
  1. At the source code level before compilation?
  2. At the binary code level post link time?
  3. During installation on the end user's site?
- The more work you do on the user's site, the more he can learn about your method of protection!

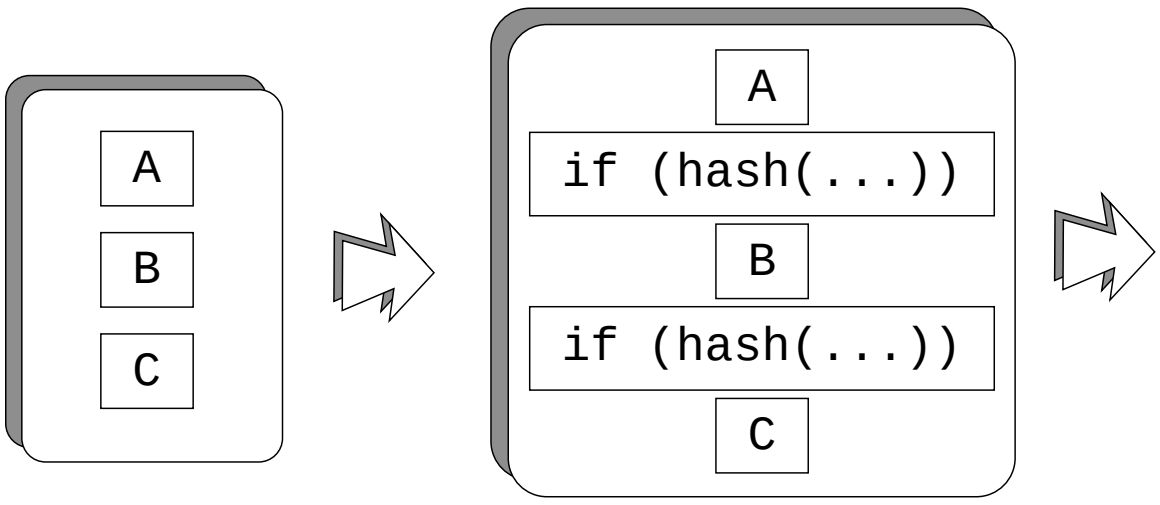
System design

- spreads fingerprinting and tamperproofing work out over compile time, post link, and installation time.

System design

- spreads fingerprinting and tamperproofing work out over compile time, post link, and installation time.
- At the source code level insert checkers of the form `if (hash(start,end)) RESPOND()`:


A
B
C


A
B
C
A
if (hash(...))
B
if (hash(...))
C

System design

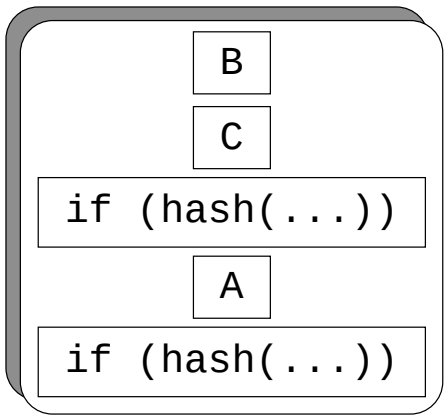
- On the binary executable randomize the basic blocks and checkers. This spreads the checkers evenly over the program and helps with preventing collusive attacks.

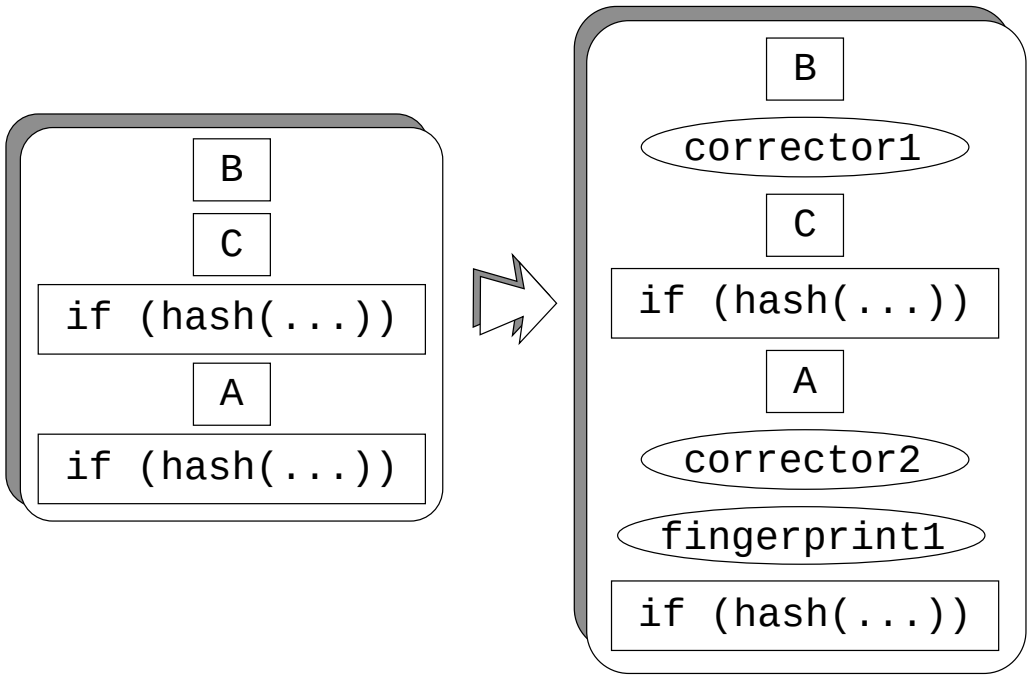
System design

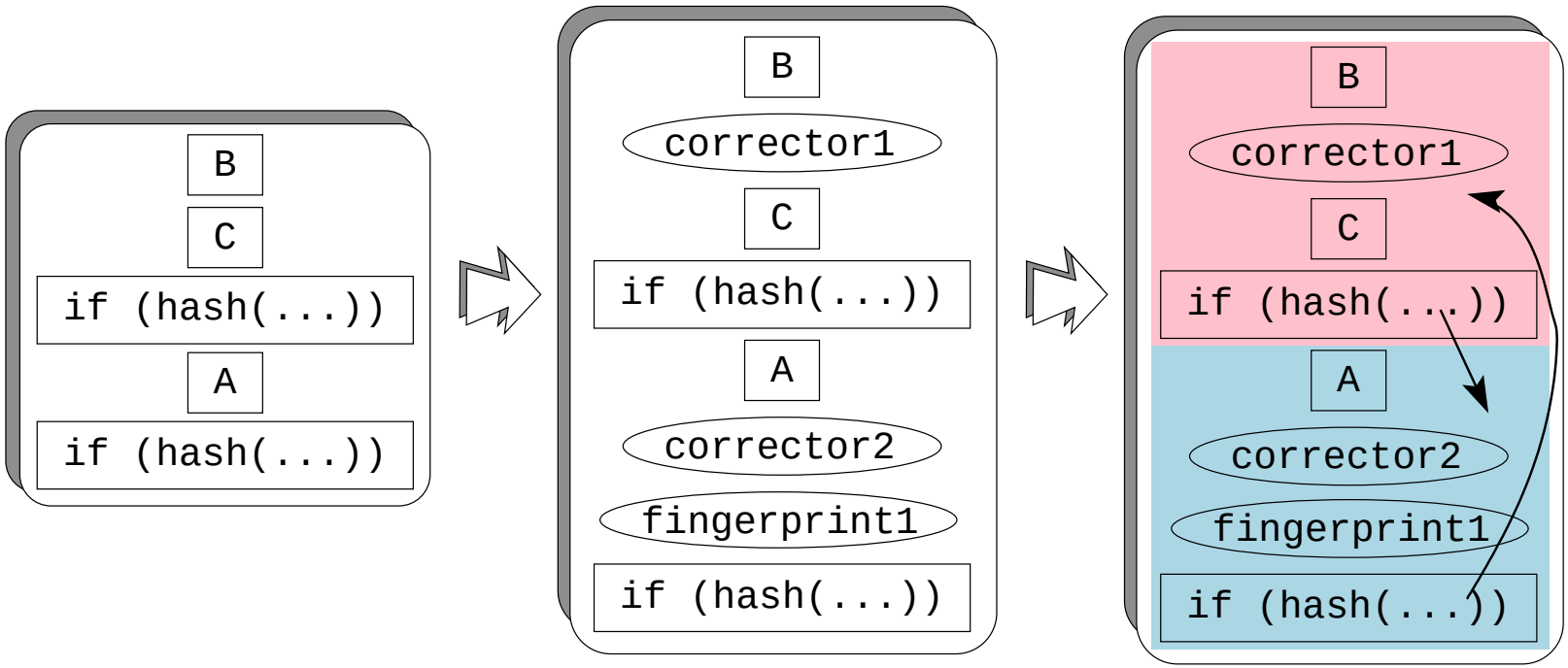
- On the binary executable randomize the basic blocks and checkers. This spreads the checkers evenly over the program and helps with preventing collusive attacks.
- Insert empty 32-bit slots for correctors and fingerprints.

System design

- On the binary executable randomize the basic blocks and checkers. This spreads the checkers evenly over the program and helps with preventing collusive attacks.
- Insert empty 32-bit slots for correctors and fingerprints.
- Create overlapping intervals, assign each checker to a region by filling in `start` and `end`.


B
C
if (hash(...))
A
if (hash(...))


B
C
if (hash(...))
A
if (hash(...))
B
corrector1
C
if (hash(...))
A
corrector2
fingerprint1
if (hash(...))

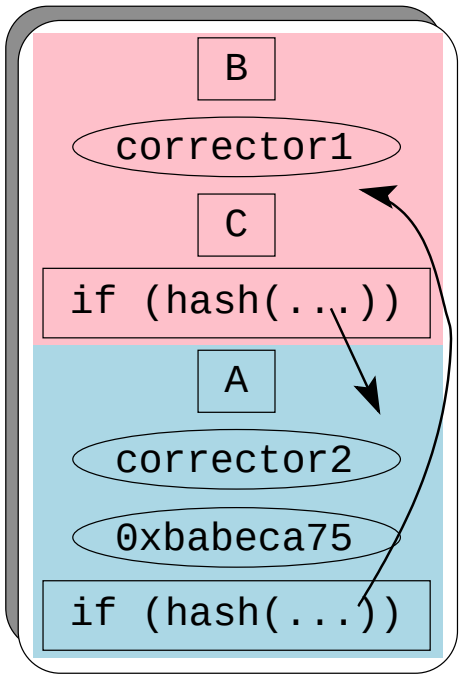

B
C
if (hash(...))
A
if (hash(...))
B
corrector1
C
if (hash(...))
A
corrector2
fingerprint1
if (hash(...))
B
corrector1
C
if (hash(...))
A
corrector2
fingerprint1
if (hash(...))

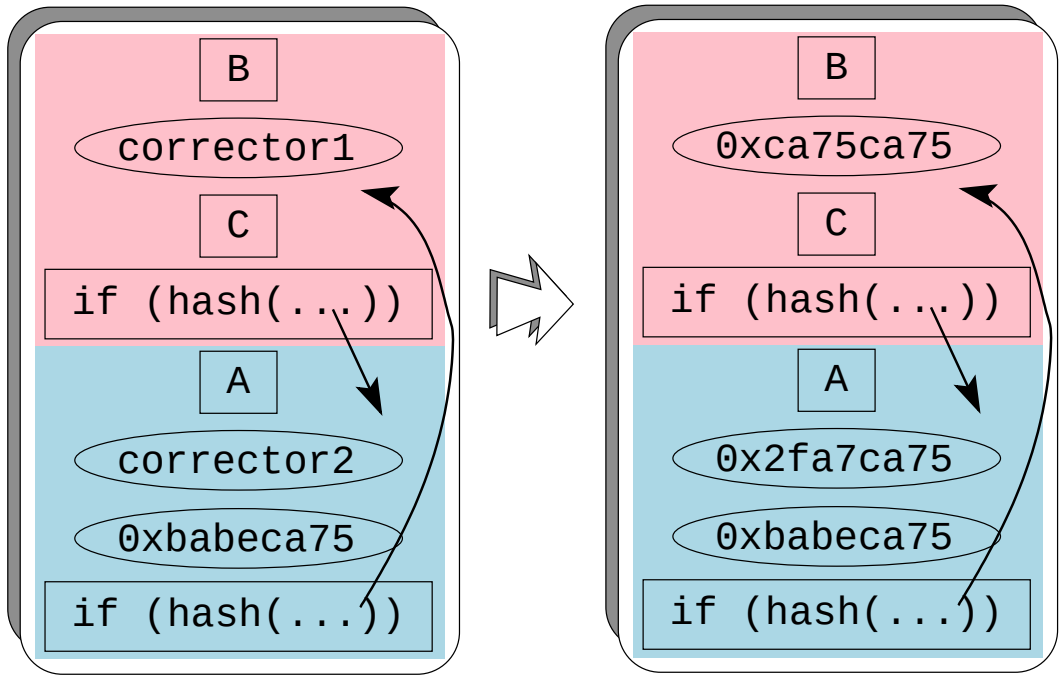
System design

- During installation ifill in the user's fingerprint values.

System design

- During installation ifill in the user's fingerprint values.
- Compute and fill in corrector vales such that each checker hashes to zero.


B
corrector1
C
if (hash(...))
A
corrector2
0xbabeca75
if (hash(...))


B
corrector1
C
if (hash(...))
A
corrector2
0xbabeca75
if (hash(...))
B
0xca75ca75
C
if (hash(...))
A
0x2fa7ca75
0xbabeca75
if (hash(...))

```
int main (int argc, char *argv[]) {
   int user_key = 0xca7ca115;
   int digital_media[] = {10,102};
   play(user_key,digital_media,2);
}
 int getkey(int user_key) {
   int player_key = 0xbabeca75;
   return user_key ^ player_key;
}
int decrypt(int user_key, int media) {
   int key = getkey(user_key);
   return media ^ key;
}
float decode (int digital) {return (float)digital;}
void play(int user_key, int digital_media[], int len) {
   int i;
   for(i=0;i<len;i++)
      printf("%f\n",decode(decrypt(user_key,digital_medi
}
```

Algorithm — Example

```
#define interval1K          3
#define interval1START      (waddr_t)main
#define interval1END        (waddr_t)decode
#define interval1CORRECTOR "0x2e1e55ec"

#define interval2K          5
#define interval2START      (waddr_t)RESPOND
#define interval2END        (waddr_t)play
#define interval2CORRECTOR "0x2cdbf568"

#define interval3K          7
#define interval3START      (waddr_t)getkey
#define interval3END        (waddr_t)LAST_FUN
#define interval3CORRECTOR "0x28d32bb6"
```

Algorithm — Example

```
//----------------- Begin interval 1 ----------------
uint32 main (uint32 argc, char *argv[]) {
   uint32 user_key = 0xca7ca115;
   uint32 digital_media[] = {10,102};
   play(user_key,digital_media,2);
}

//----------------- Begin interval 2 ----------------

void RESPOND(int i){
   printf("\n*** interval%i hacked!\n",i);
   abort();
}
```

Algorithm — Example

```
//----------------- Begin interval 3 ---------------

uint32 getkey(uint32 user_key) {
    uint32 player_key = 0xbabeca75;
    if ( hash5(interval1START,interval1END,interval1K) ) {
        RESPOND(1);
        asm volatile (
            "       .align 4                                \n\t"
            "       .long " interval1CORRECTOR " \n\t"
        );
    }
    return user_key ^ player_key;
}
```

```
uint32 decrypt(uint32 user_key, uint32 media) {
    uint32 key = getkey(user_key);
    return media ^ key;
}

//---------------- End interval 1 ---------------

float decode (uint32 digital) {
    if ( hash5(interval2START,interval2END,interval2K) ) {
        RESPOND(2);
        asm volatile (
            "       .align 4                                \n\t"
            "       .long " interval2CORRECTOR " \n\t"
        );
    }
    return (float)digital;
}

//---------------- End interval 2 ---------------
```

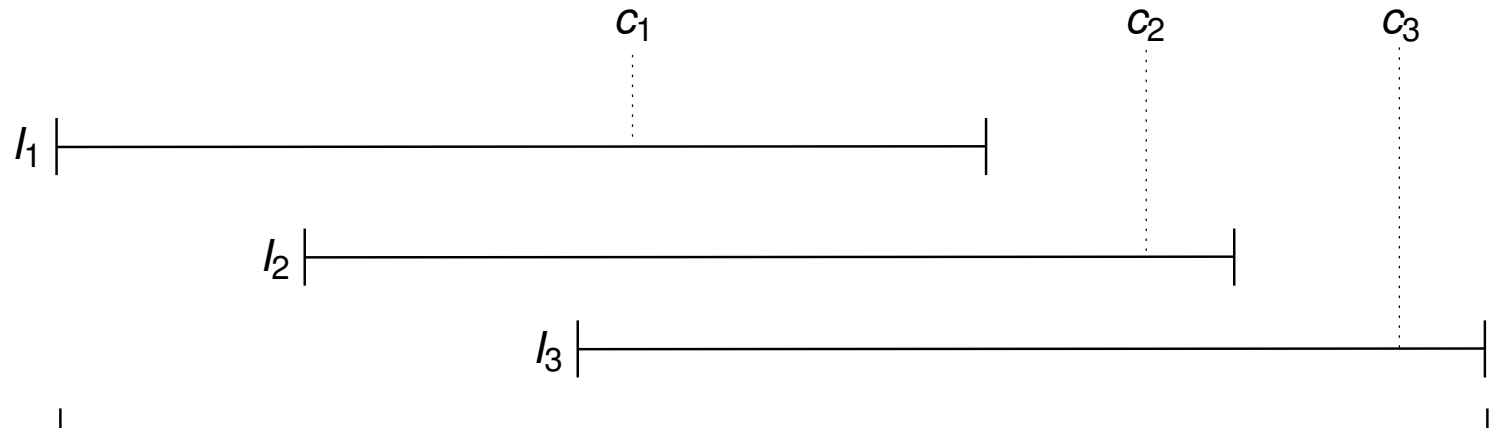
```
void play(uint32 user_key, uint32 digital_media[], uint3
    uint32 i;
    for(i=0;i<len;i++)
        printf("%f\n",decode(decrypt(user_key,digital_medi
    asm volatile (
        "       jmp L1                                  \n\t"
        "       .align 4                                \n\t"
        "       .long " interval3CORRECTOR " \n\t"
        "L1:    \n\t"
    );
    if (hash5(interval3START,interval3END,interval3K))
        RESPOND(3);
}

//---------------- End interval 3 ---------------
void LAST_FUN(){}
```

Algorithm

- randomly places large numbers of checkers all over the program, but makes sure that every piece of code is covered by *multiple* checkers.

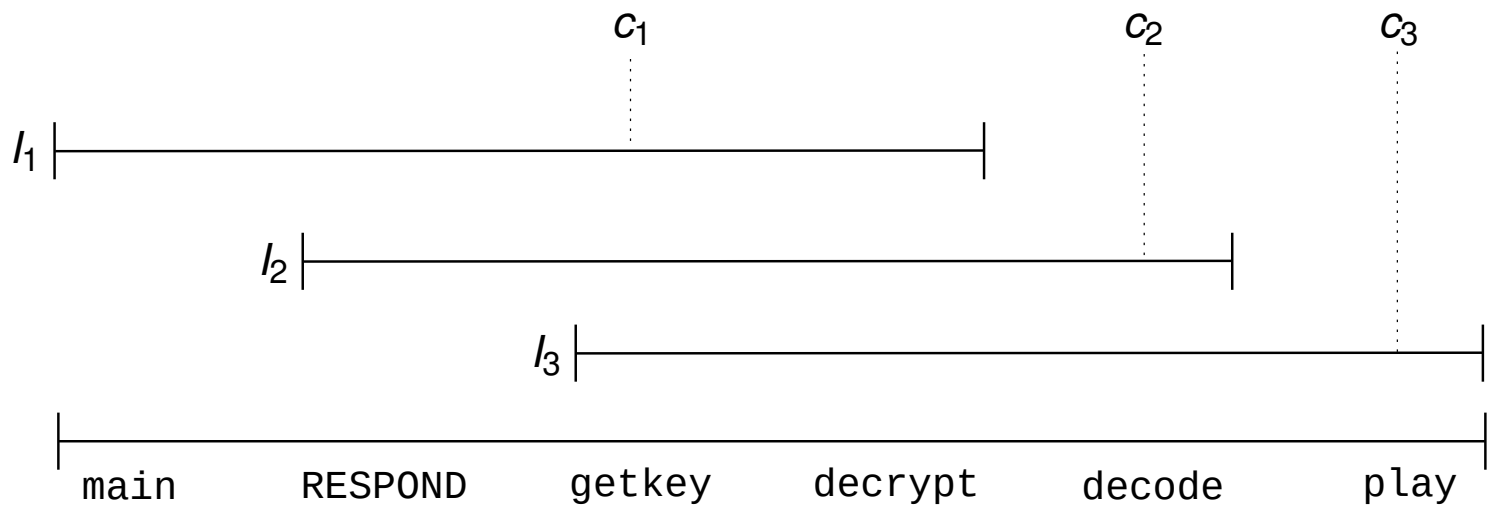
Algorithm

- randomly places large numbers of checkers all over the program, but makes sure that every piece of code is covered by *multiple* checkers.
- Each interval has a checker that tests that interval, and each interval I_i has a corrector c_i that you fill in to make sure that the checker hash function hashes to zero.



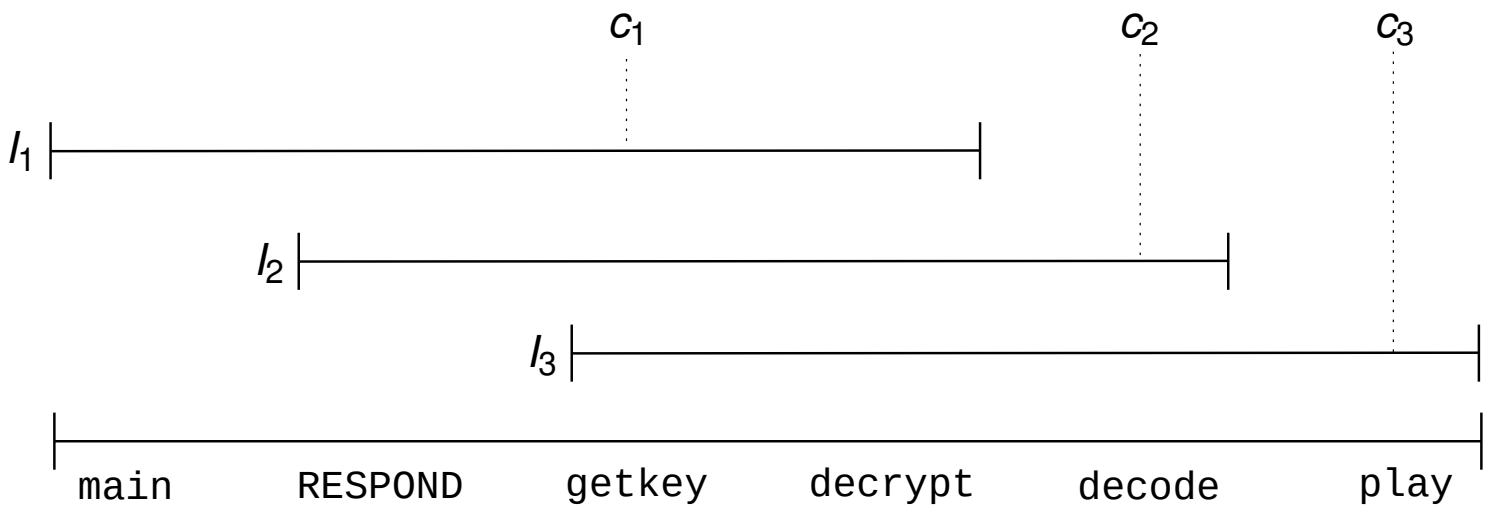
Algorithm

- Compute the correctors in the order $c_1, c_2, c_3, \ldots$ to avoid circular dependencies.
  1. set c_1 so that interval I_1 hashes to zero,



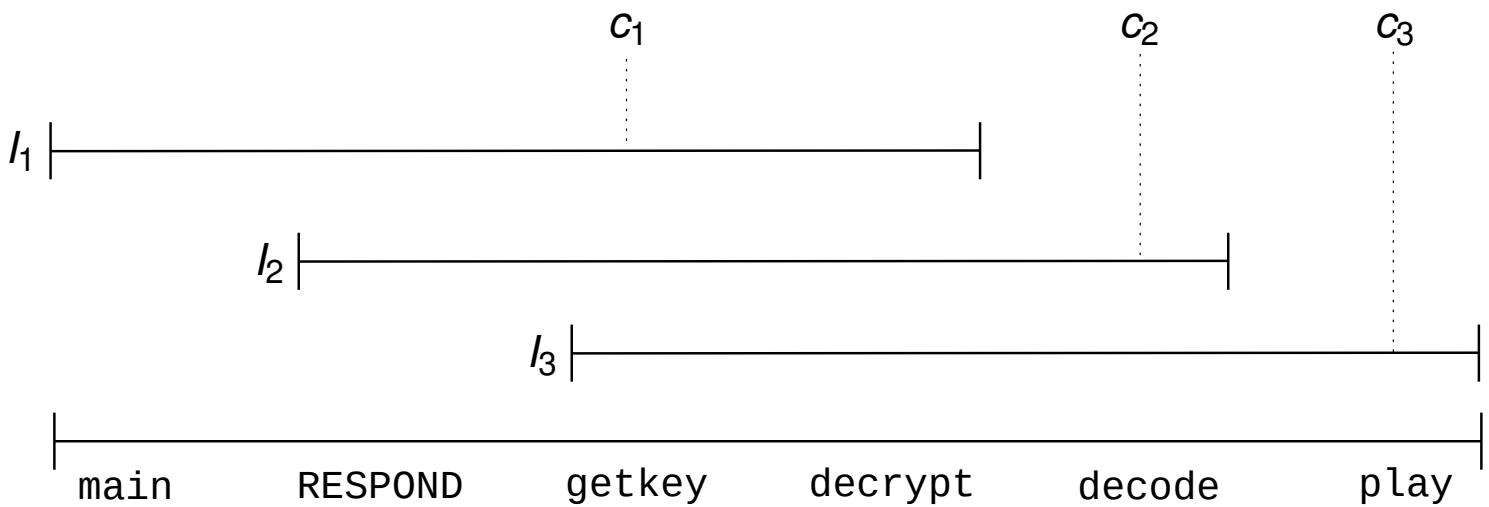
Algorithm

- Compute the correctors in the order $c_1, c_2, c_3, \ldots$ to avoid circular dependencies.
  1. set c_1 so that interval I_1 hashes to zero,
  2. I_2 only has one fill it in so that I_2 hashes to zero, etc.



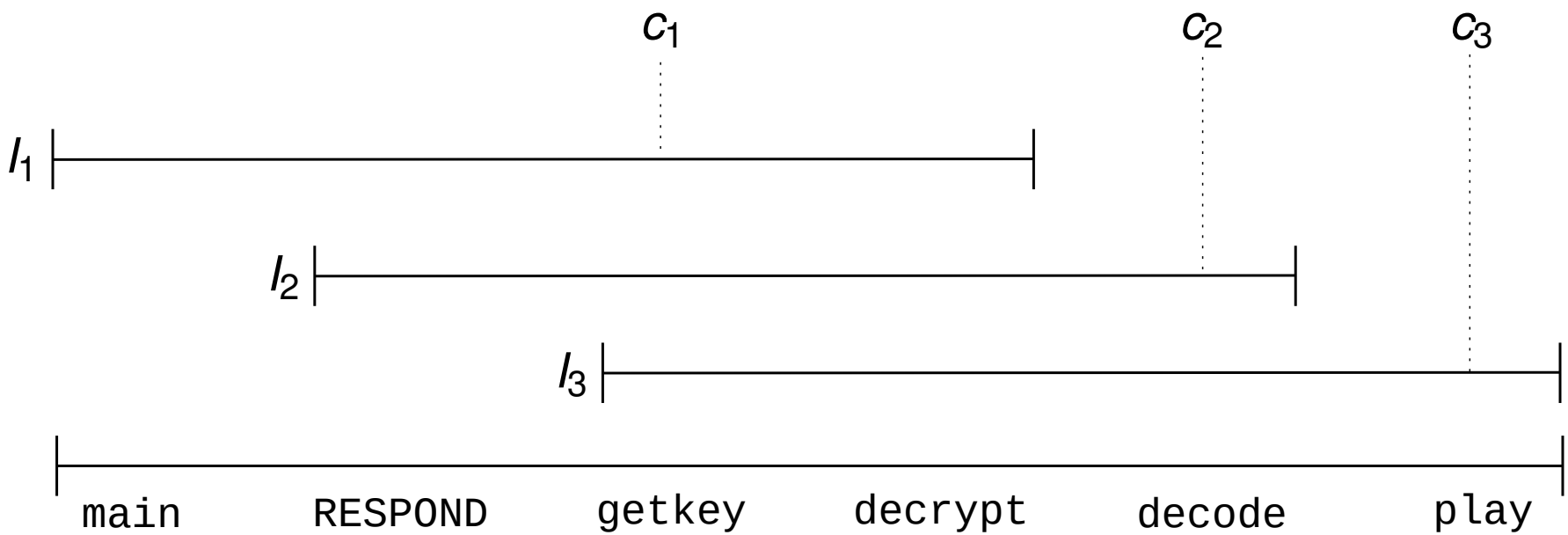
Algorithm

- Compute the correctors in the order $c_1, c_2, c_3, \ldots$ to avoid circular dependencies.
  1. set c_1 so that interval I_1 hashes to zero,
  2. I_2 only has one fill it in so that I_2 hashes to zero, etc.
  3. etc.



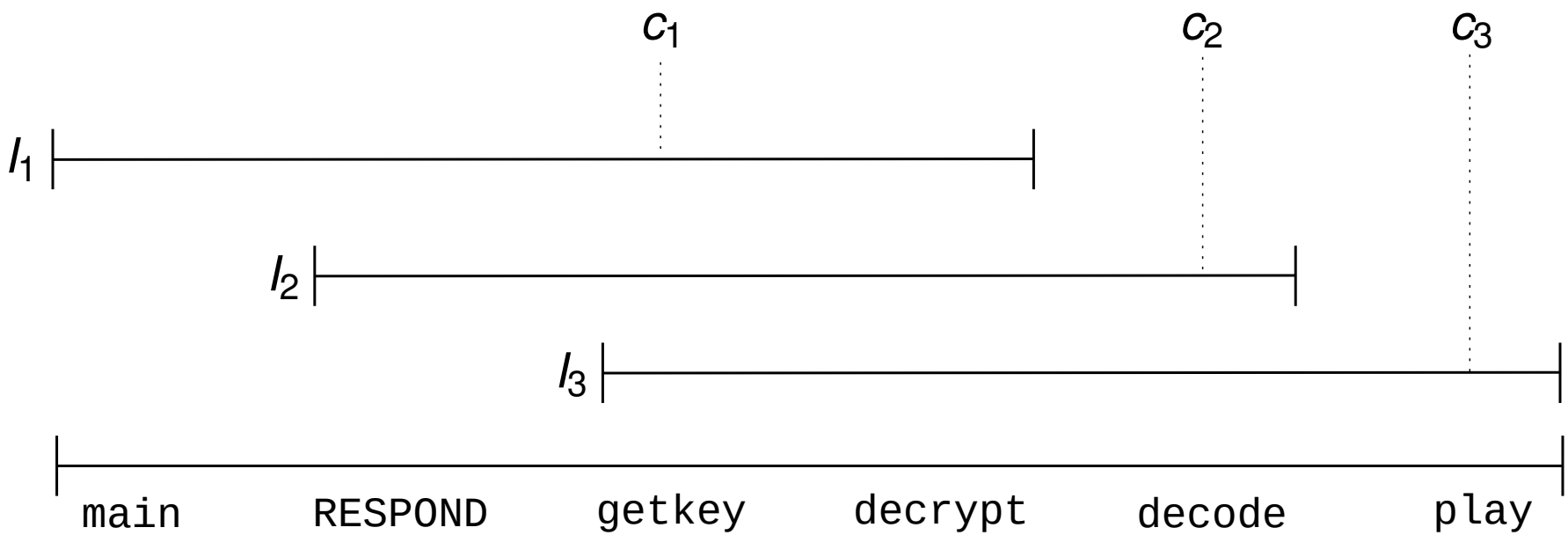
Algorithm

- Here, the *overlap factor* is 2.



Algorithm

- Here, the *overlap factor* is 2.
- The authors suggest that an overlap factor of 6 gives the right trade-off between resilience and overhead.



Computing corrector slot values

```
uint32 hash5 (addr_t start, addr_t end, uint32 C) {
   uint32 h = 0;
   while (start < end) {
      h = C*(*start + h);
      start++;
   }
   return h;
}
```

- Hash an incomplete range (the corrector slot value is unknown) and then later solve for the corrector slot.

Computing corrector slot values

- $x = [x_1, x_2, \ldots, x_n]$ is the list of n 32-bit words.

Computing corrector slot values

- $x = [x_1, x_2, \ldots, x_n]$ is the list of n 32-bit words.
- x has one empty corrector slot slot.

Computing corrector slot values

- $x = [x_1, x_2, \ldots, x_n]$ is the list of n 32-bit words.
- x has one empty corrector slot slot.
- The region hashes to $h(x)$:

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

Computing corrector slot values

- $x = [x_1, x_2, \ldots, x_n]$ is the list of n 32-bit words.
- x has one empty corrector slot slot.
- The region hashes to $h(x)$:

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

- C is a small, odd, constant multiplier.

Computing corrector slot values

- $x = [x_1, x_2, \ldots, x_n]$ is the list of n 32-bit words.
- x has one empty corrector slot slot.
- The region hashes to $h(x)$:

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

- C is a small, odd, constant multiplier.
- All computations are done modulo 2^{32}.

Computing corrector slot values

- One of the values in the region, say x_k, is the empty corrector slot.

Computing corrector slot values

- One of the values in the region, say x_k, is the empty corrector slot.
- Find a value for x_k such that $h(x) = 0$!

Computing corrector slot values

- One of the values in the region, say x_k, is the empty corrector slot.
- Find a value for x_k such that $h(x) = 0$!
- Let z be the part of the hash-vlaue that excludes x_k:

$$z = \sum_{i \neq k}^{n} C^{n-i+1} x_i$$

Computing corrector slot values

- One of the values in the region, say x_k, is the empty corrector slot.
- Find a value for x_k such that $h(x) = 0$!
- Let z be the part of the hash-vlaue that excludes x_k:

$$z = \sum_{i \neq k}^{n} C^{n-i+1} x_i$$

- We're looking for a value for x_k such that

$$C^{n-k+1} x_k + z = 0 \pmod{2^{32}}$$

Computing corrector slot values

- One of the values in the region, say x_k, is the empty corrector slot.
- Find a value for x_k such that $h(x) = 0$!
- Let z be the part of the hash-vlaue that excludes x_k:

$$z = \sum_{i \neq k}^{n} C^{n-i+1} x_i$$

- We're looking for a value for x_k such that

$$C^{n-k+1} x_k + z = 0 \pmod{2^{32}}$$

- This is a modular linear equation!

Computing corrector slot values

Theorem (Modular linear equation)

The modular linear equation $ax \equiv b \pmod{n}$ *is solvable if* $d|b$, *where* $d = \gcd(a,n) = ax' + ny'$ *is given by Euclid's extended algorithm. If* $d|b$ *there are* d *solutions:*

$$\begin{aligned} x_0 &= x'(b/d) \bmod n \\ x_i &= x_0 + i(n/d) \quad \text{where} \quad i = 1,2,\ldots,d-1 \end{aligned}$$

Computing corrector slot values

You get,

$$\begin{aligned} C^{n-k+1} x_k &= -z \pmod{2^{32}} \\ d &= \gcd(C^{n-k+1}, 2^{32}) = C^{n-k+1} x' + 2^{32} y' \\ x_0 &= x'(-z/d) \bmod 2^{32} \end{aligned}$$

Since C is odd, $d = 1$, and you get the solution

$$x_0 = -zx' \pmod{2^{32}}$$

Example

Let $x = [1, 2, x_3, 4]$ be the region, and $C = 3$ the multiplier:

$$z = \sum_{i \neq 3}^{4} C^{n-i+1} x_i = 1 \cdot 3^4 + 2 \cdot 3^3 + 4 \cdot 3^1 = 147$$

$$3^2 x_3 =$$

Example

Let $x = [1, 2, x_3, 4]$ be the region, and $C = 3$ the multiplier:

$$\begin{aligned} z &= \sum_{i \neq 3}^{4} C^{n-i+1} x_i = 1 \cdot 3^4 + 2 \cdot 3^3 + 4 \cdot 3^1 = 147 \\ 3^2 x_3 &= -147 \pmod{2^{32}} \\ d &= \end{aligned}$$

Example

Let $x = [1, 2, x_3, 4]$ be the region, and $C = 3$ the multiplier:

$$
\begin{aligned}
z &= \sum_{i \neq 3}^{4} C^{n-i+1} x_i = 1 \cdot 3^4 + 2 \cdot 3^3 + 4 \cdot 3^1 = 147 \\
3^2 x_3 &= -147 \pmod{2^{32}} \\
d &= \gcd(3^2, 2^{32}) = 1 \\
&=
\end{aligned}
$$

Example

Let $x = [1, 2, x_3, 4]$ be the region, and $C = 3$ the multiplier:

$$
\begin{aligned}
z &= \sum_{i \neq 3}^{4} C^{n-i+1} x_i = 1 \cdot 3^4 + 2 \cdot 3^3 + 4 \cdot 3^1 = 147 \\
3^2 x_3 &= -147 \pmod{2^{32}} \\
d &= \gcd(3^2, 2^{32}) = 1 \\
&= 3^2 \cdot 954437177 + 2^{32} \cdot (-2) \\
x_3 &=
\end{aligned}
$$

Example

Let $x = [1, 2, x_3, 4]$ be the region, and $C = 3$ the multiplier:

$$
\begin{aligned}
z &= \sum_{i \neq 3}^{4} C^{n-i+1} x_i = 1 \cdot 3^4 + 2 \cdot 3^3 + 4 \cdot 3^1 = 147 \\
3^2 x_3 &= -147 \pmod{2^{32}} \\
d &= \gcd(3^2, 2^{32}) = 1 \\
&= 3^2 \cdot 954437177 + 2^{32} \cdot (-2) \\
x_3 &= 954437177 \cdot (-147/1) \bmod 2^{32} \\
&=
\end{aligned}
$$

Example

Let $x = [1, 2, x_3, 4]$ be the region, and $C = 3$ the multiplier:

$$
\begin{aligned}
z &= \sum_{i \neq 3}^{4} C^{n-i+1} x_i = 1 \cdot 3^4 + 2 \cdot 3^3 + 4 \cdot 3^1 = 147 \\
3^2 x_3 &= -147 \pmod{2^{32}} \\
d &= \gcd(3^2, 2^{32}) = 1 \\
&= 3^2 \cdot 954437177 + 2^{32} \cdot (-2) \\
x_3 &= 954437177 \cdot (-147/1) \bmod 2^{32} \\
&= 1431655749
\end{aligned}
$$

Example — Checking the Result

We get:

$$\begin{aligned} h(x) &= (1 \cdot 3^4 + 2 \cdot 3^3 + 1431655749 \cdot 3^2 + \\ &\quad 4 \cdot 3^1) \bmod 2^{32} \\ &= \end{aligned}$$

Example — Checking the Result

We get:

$$\begin{aligned} h(x) &= (1 \cdot 3^4 + 2 \cdot 3^3 + 1431655749 \cdot 3^2 + \\ &\quad 4 \cdot 3^1) \bmod 2^{32} \\ &= 0 \end{aligned}$$

as expected.

Attacking self-hashing algorithms

Attacking self-hashing algorithms

- How to attack introspection algorithms?
  1. Analyze the code to locate the checkers, or

Attacking self-hashing algorithms

- How to attack introspection algorithms?
  1. Analyze the code to locate the checkers, or
  2. Analyze the code to locate the responders, then

Attacking self-hashing algorithms

- How to attack introspection algorithms?
  1. Analyze the code to locate the checkers, or
  2. Analyze the code to locate the responders, then
  3. Remove or disable them without destroying the rest of the program.

Attacking self-hashing algorithms

- How to attack introspection algorithms?
  1. Analyze the code to locate the checkers, or
  2. Analyze the code to locate the responders, then
  3. Remove or disable them without destroying the rest of the program.
- Attack can just as well be external to the program!

Memory System

- Processors treat code and data differently.

Memory System

- Processors treat code and data differently.
- TLBs (Translation Lookaside Buffers) and caches are split in separate parts for code and data.

Memory System

- 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 algorithms code is accessed
  1. as code (when it's being executed) and

  $\Rightarrow$ sometimes a function will be read into the I-cache and sometimes into the D-cache.

Memory System

- 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 algorithms code is accessed
  1. as code (when it's being executed) and
  2. as data (when it's being hashed).

  $\Rightarrow$ sometimes a function will be read into the I-cache and sometimes into the D-cache.

Attack Idea

- Attack: modify the OS such that
  1. redirect reads of the code to the original, unmodified program (hash values will be computed as expected!)

Attack Idea

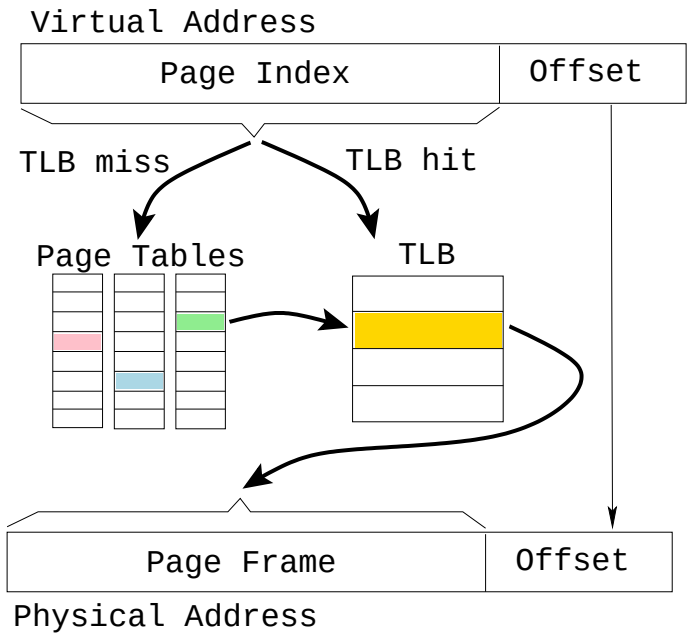
- Attack: modify the OS such that
  1. redirect reads of the code to the original, unmodified program (hash values will be computed as expected!)
  2. redirect execution of the code to the modified program (the modified code will get executed!)

Attack Algorithm

ATTACK(P, K):

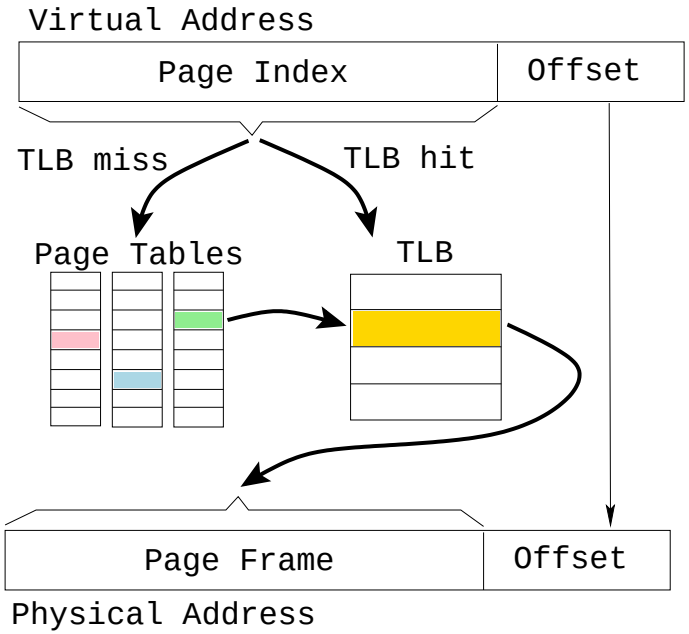
1. Copy program P to P_{orig}.
2. Modify P as desired to a hacked version P'.
3. Modify the operating system kernel K such that data reads are directed to P_{orig}, instruction reads to P'. □

Typical Memory Management System

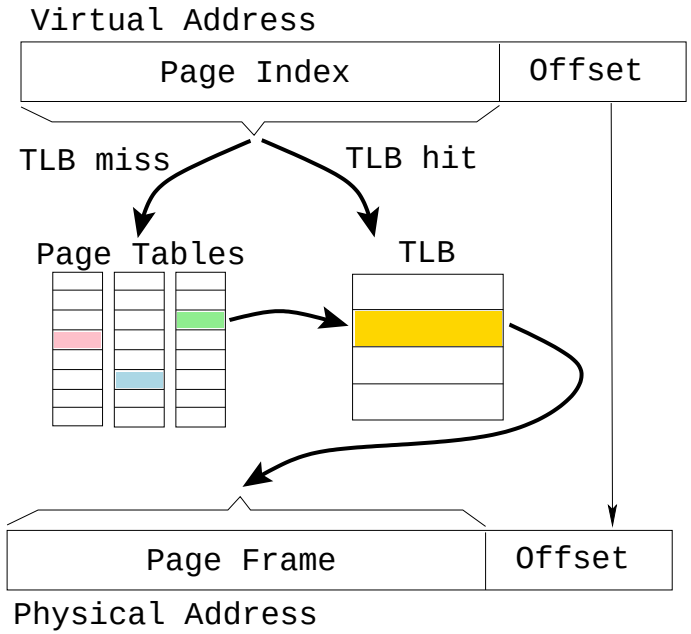


- On a TLB miss walk the page tables (slow), and update the TLB with the new virtual-to-physical address mapping.

Memory Management - TLB Miss

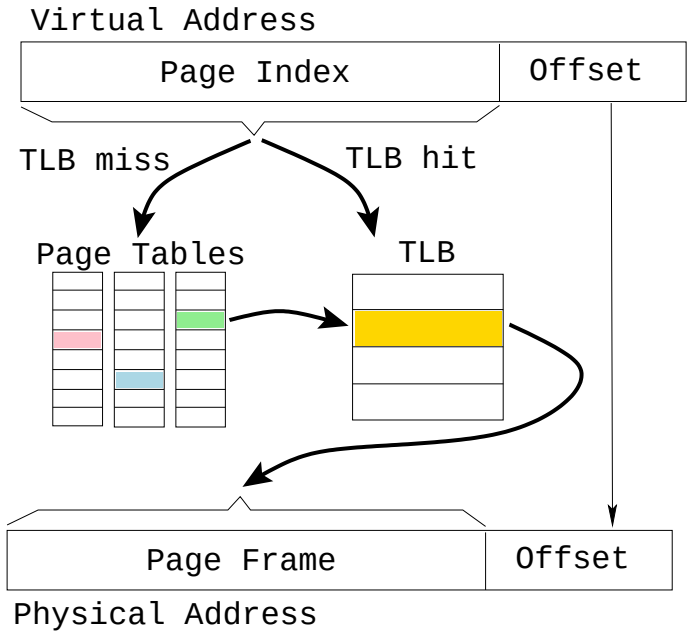


Memory Management - TLB Miss



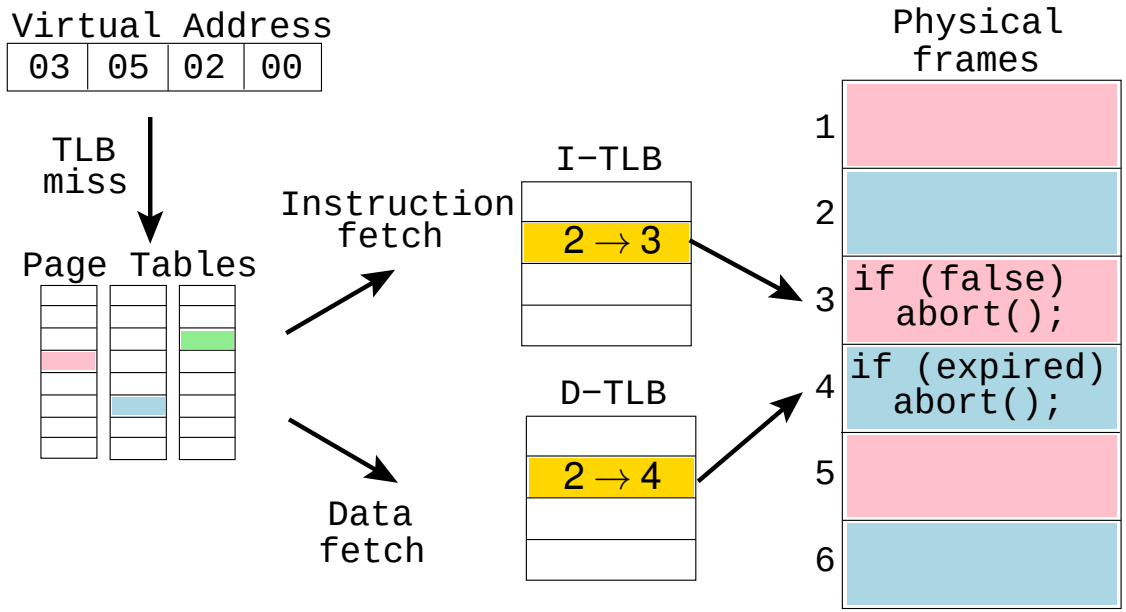
- TLB miss caused by data fetch $\Rightarrow$ CPU throws Exception1.

Memory Management - TLB Miss



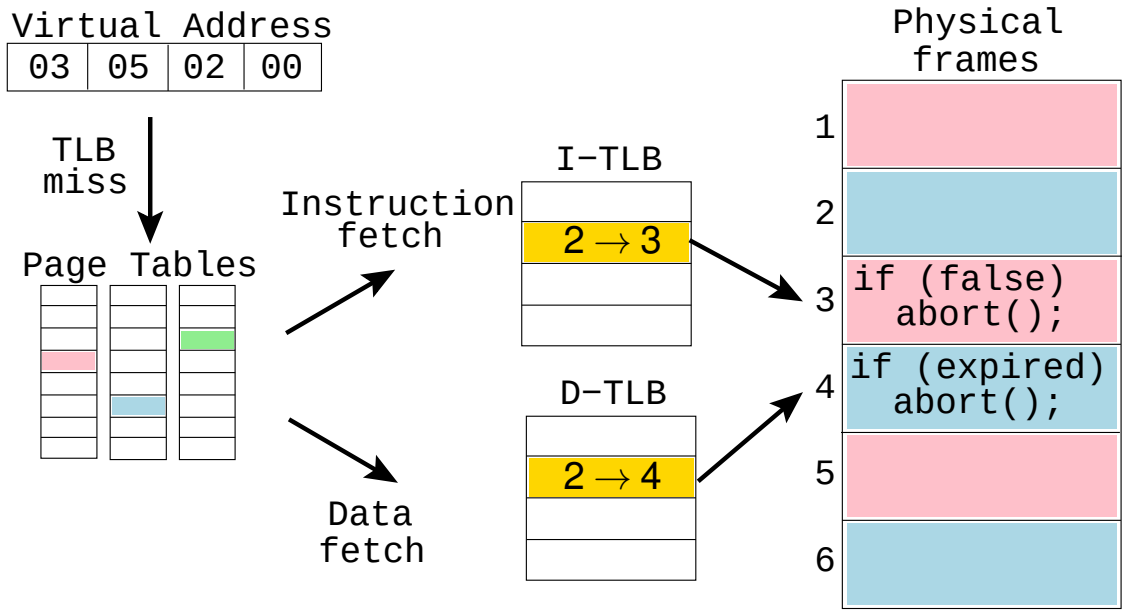
- TLB miss caused by data fetch $\Rightarrow$ CPU throws Exception1.
- TLB miss caused by instructon fetch $\Rightarrow$ CPU throws Exception2.

Attack Details — Memory Layout



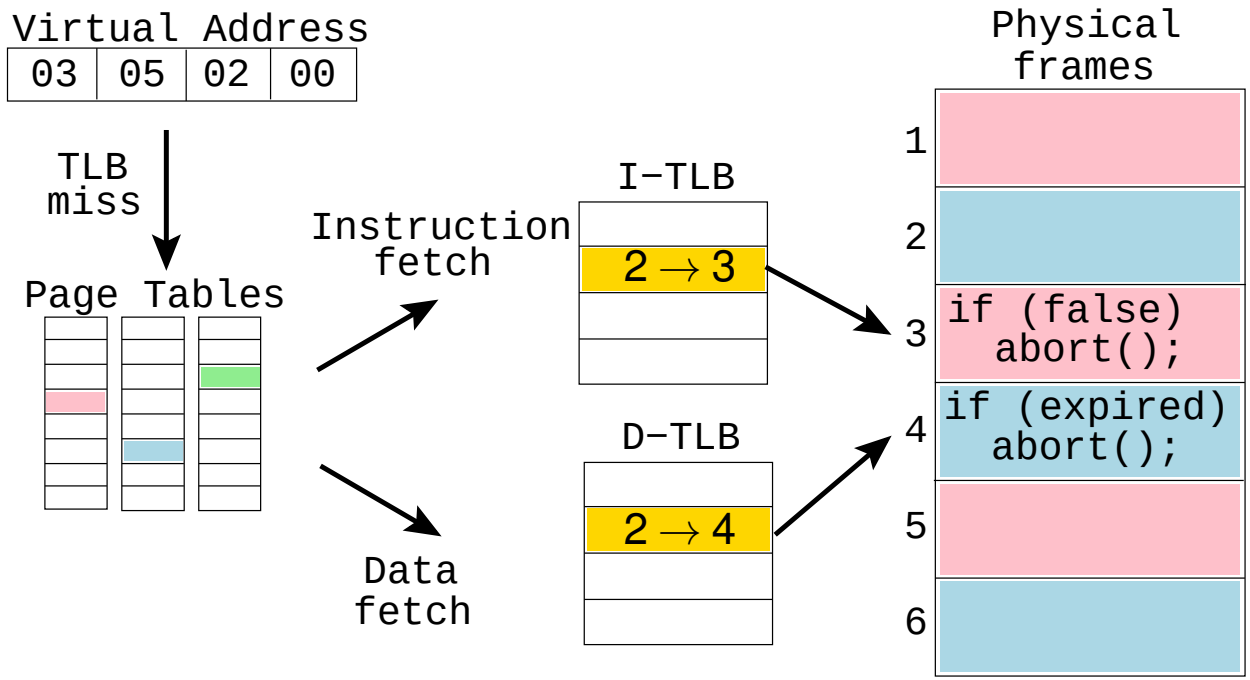
1. Copy P to P_{orig} and hack P.

Attack Details — Memory Layout



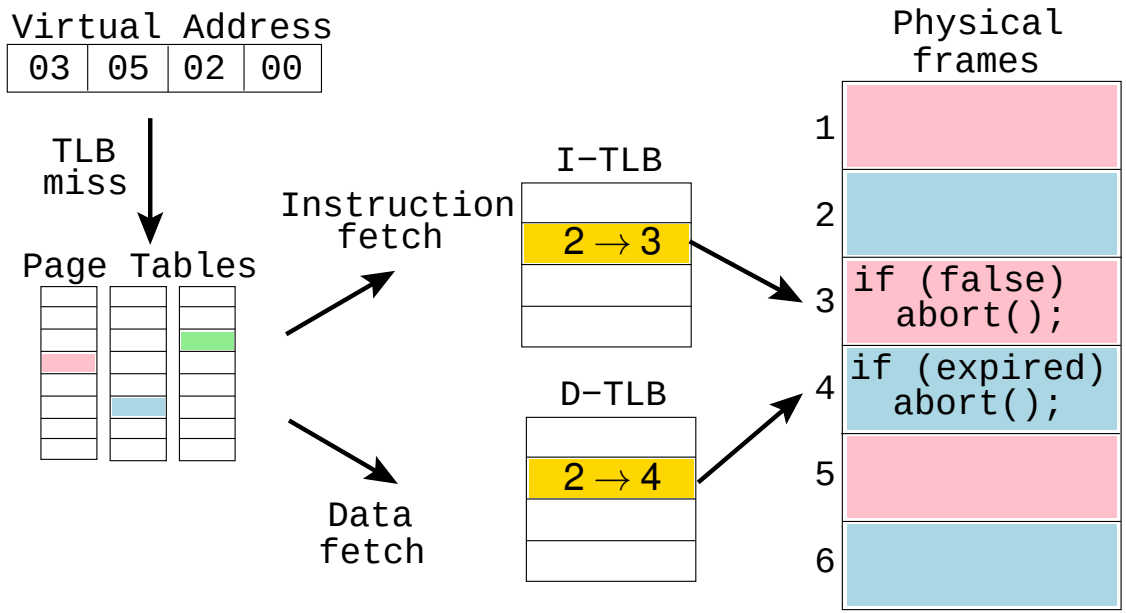
1. Copy P to P_{orig} and hack P.
2. Rearrange the physical memory: frame i comes from the hacked P and frame $i+1$ is the original frame from P_{orig}.

Attack Details — Memory Layout



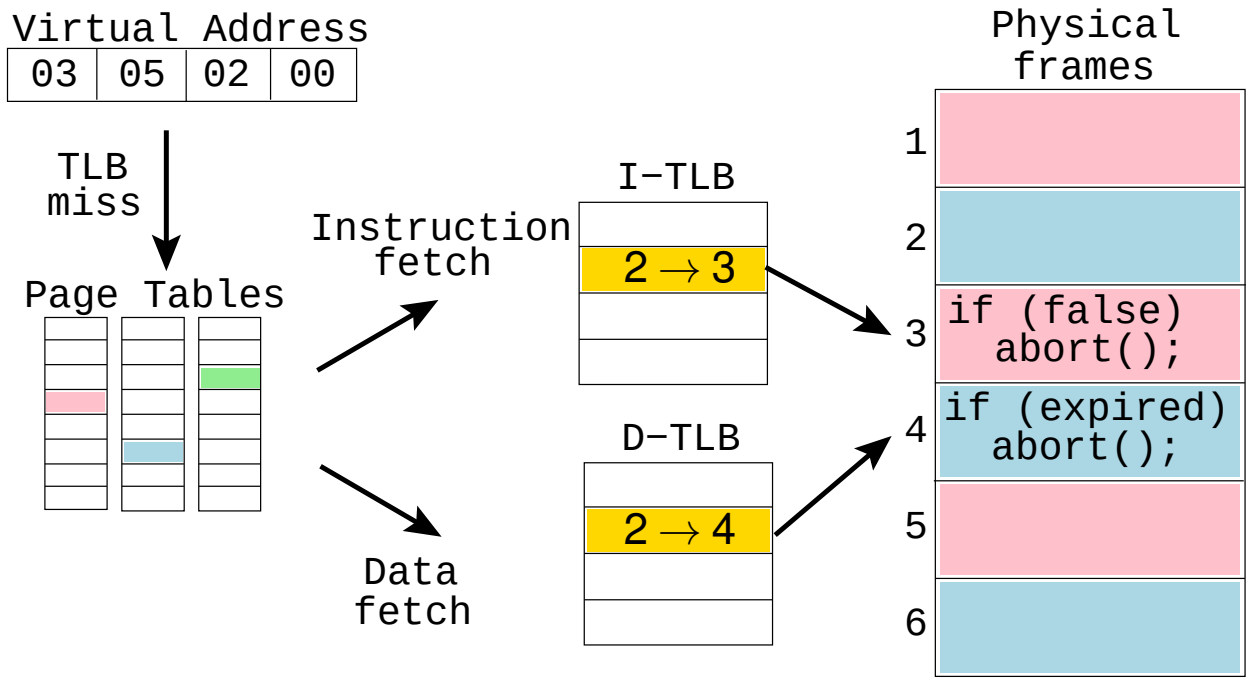
- The attacker has modified the program to bypass a license-expired check.
- The original program pages are in blue.
- The modified program pages are in pink.

Attack Details — Modify the Kernel



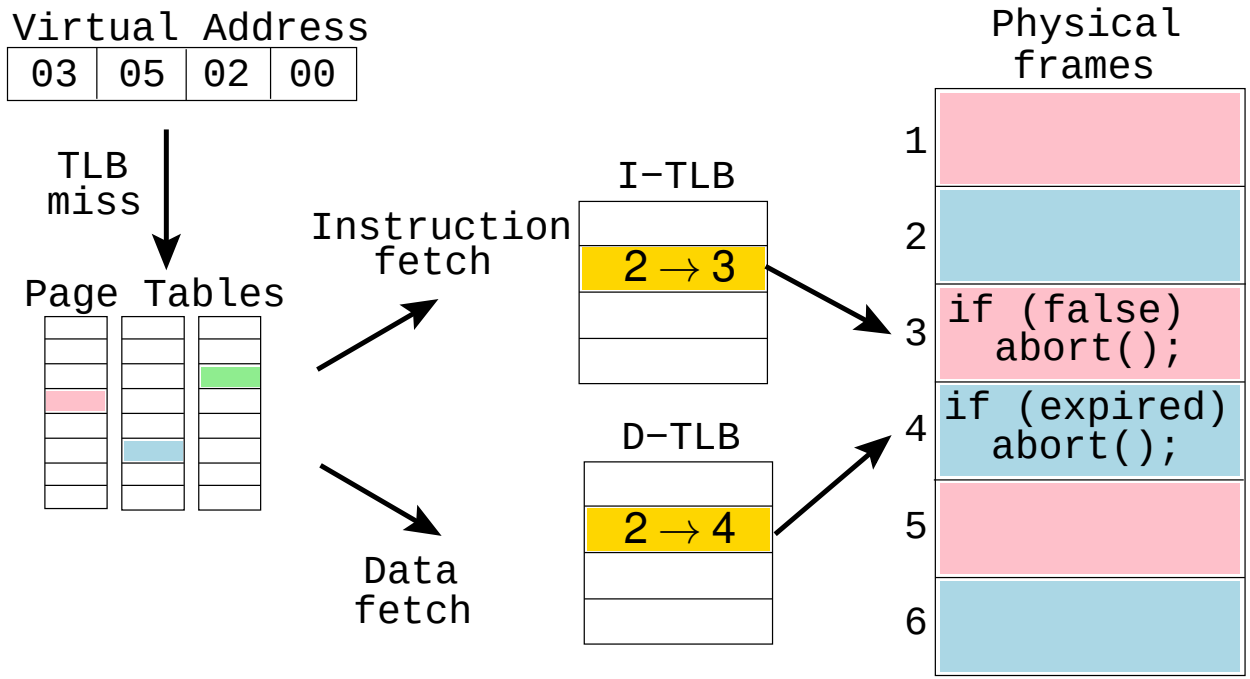
3. If a page table lookup yields a $v \rightarrow p$ virtual-to-physical address mapping, I-TLB is updated with $v \rightarrow p$ and D-TLB with $v \rightarrow p + 1$.

Attack Details — Execution Behavior



1. 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.

Attack Details — Execution Behavior



2. The program tries to read its own code in order hash the processor throws a D-TLB-miss exception, and the OS updates the D-TLB to refer to the original, unmodified, page.

State inspection

What's wrong with introspection algorithms?

- Introspection algorithms
  1. read their own code segment (unusual)!
  2. only check the validity of the code itself (not runtime data, function return values, ...).

What's wrong with introspection algorithms?

- Introspection algorithms
  1. read their own code segment (unusual)!
  2. only check the validity of the code itself (not runtime data, function return values, ...).
- Oblivious algorithms
  1. detect tampering from the *side-effects* the code produces
  2. check the correctess of data and control-flow

Oblivious $\Rightarrow$ the adversary should be unaware that his code is being checked.

Oblivious hashing

- More stealthy than introspection techniques.
  - We don't read our own code!
- An advanced form of assertion checking:

```
ASSERT x < 100;
ASSERT y != null;
```

- Works on Java as well as binary code.

Oblivious hashing

- IDEA: overlap basic blocks of x86 instructions.

Oblivious hashing

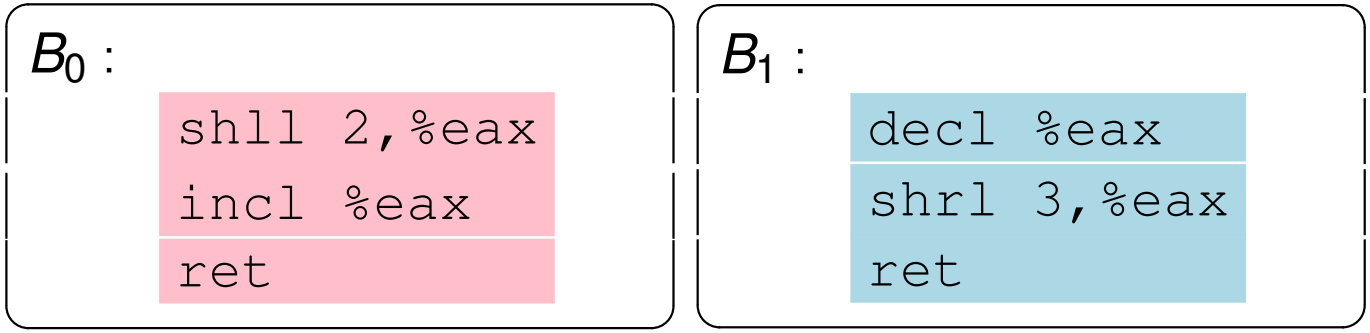
- IDEA: overlap basic blocks of x86 instructions.
- When one block executes it also computes a hash over the second block!

Oblivious hashing

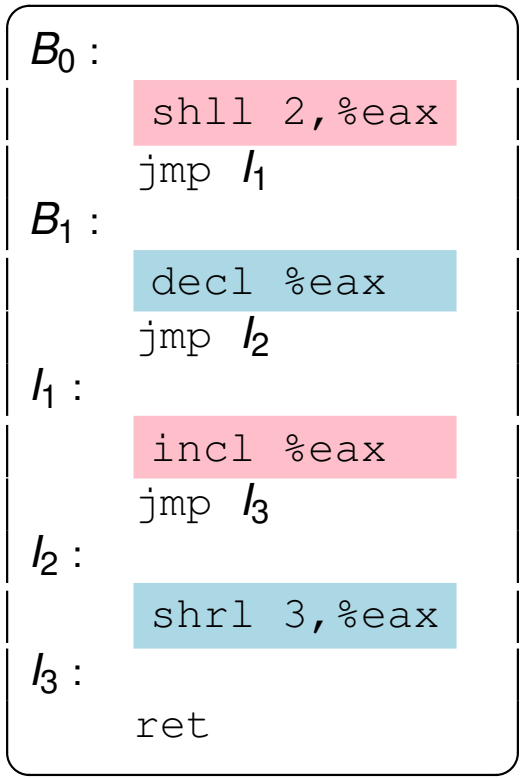
- IDEA: overlap basic blocks of x86 instructions.
- When one block executes it also computes a hash over the second block!
- The hash is computed *without* reading the code!

Oblivious hashing

- IDEA: overlap basic blocks of x86 instructions.
- When one block executes it also computes a hash over the second block!
- The hash is computed *without* reading the code!
- Invulnerable to memory splitting attacks!



Merge the blocks by interleaving the instructions, inserting jumps to maintain semantics:



- The merged block has two entry points, B_0 and B_1. The two blocks should also share instruction bytes.
- Replace the `jmp` with `xorl` that takes a 4-byte literal argument:

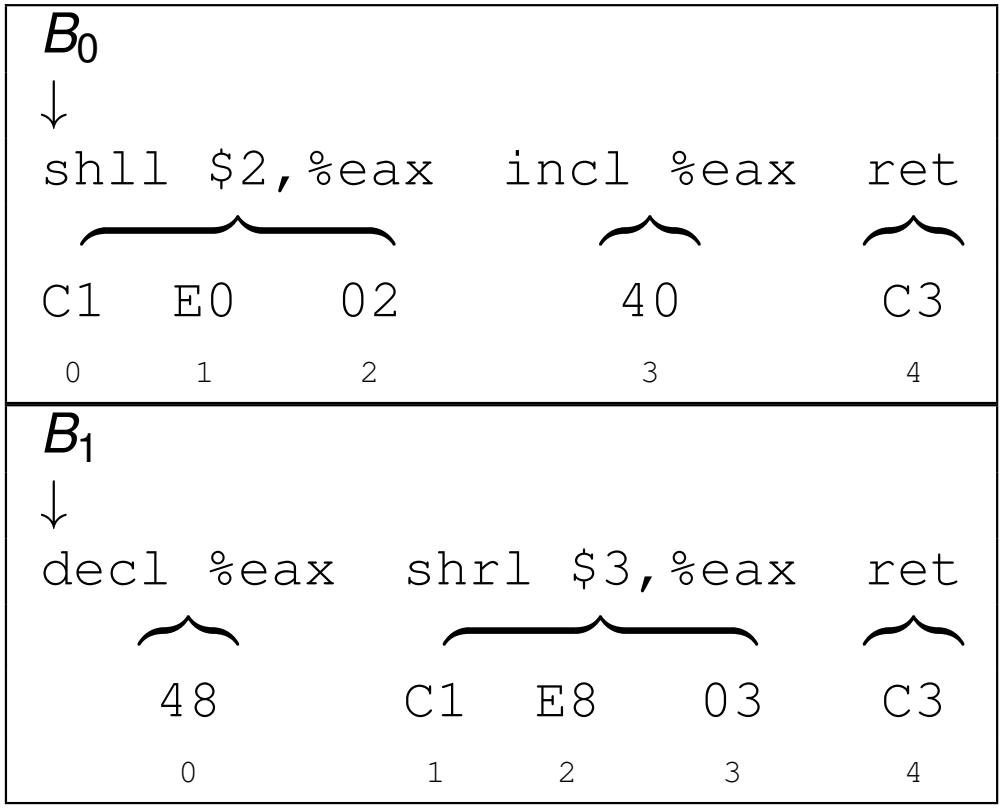
B_0 :

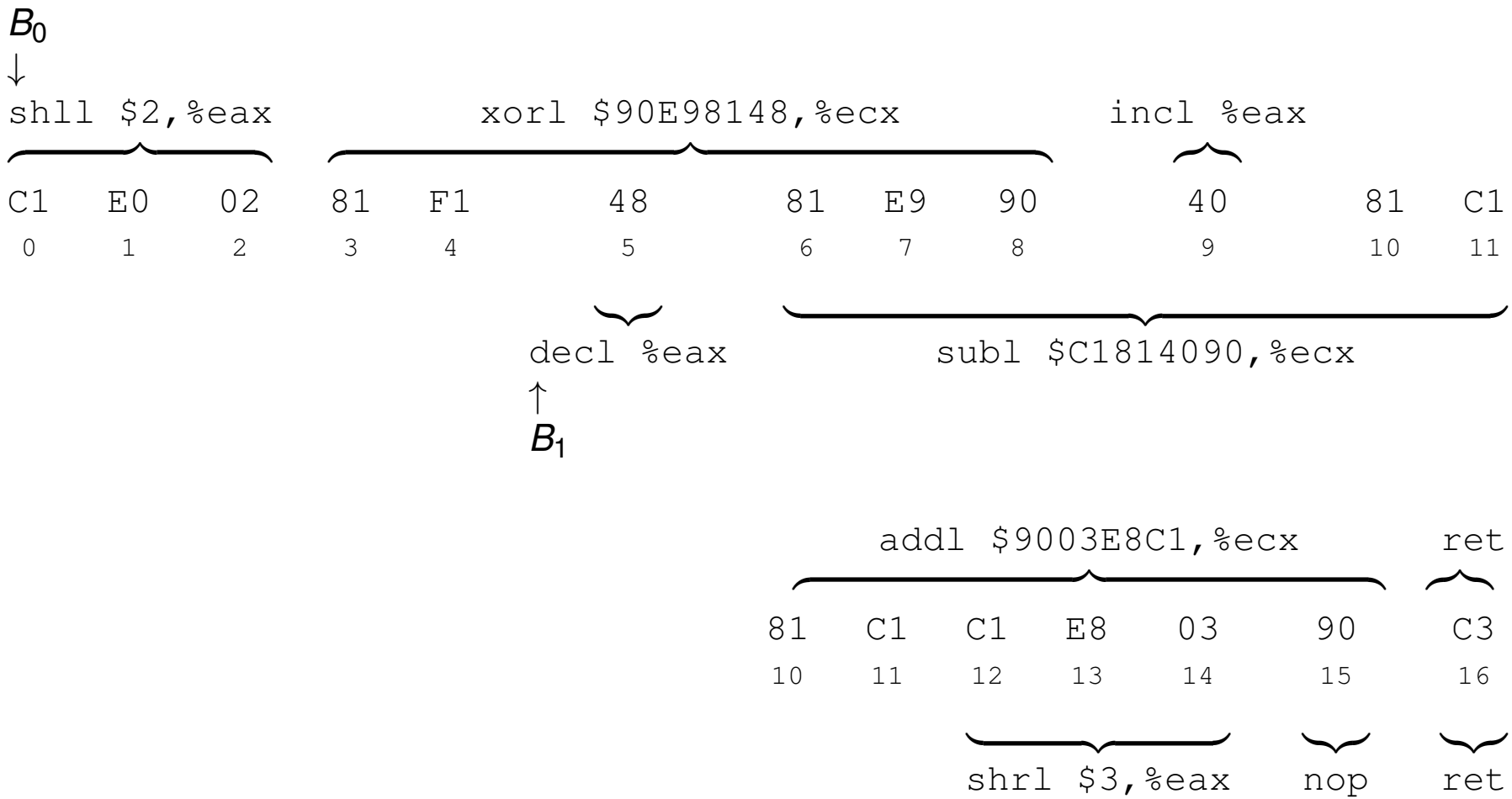
```
shll 2,%eax
xorl %ecx,next 4 bytes  // used to be jmp l1
```

B_1 :

```
decl %eax
jmp l2
nop
incl %eax
...
```

- The `xorl` instruction has, embedded in its immediate operand, the four bytes from `decl;jmp;nop`!


B_0
shll $2,%eax
incl %eax
ret
C1 E0 02
40
C3
0 1 2 3 4
B_1
decl %eax
shrl $3,%eax
ret
48
C1 E8 03
C3
0 1 2 3 4

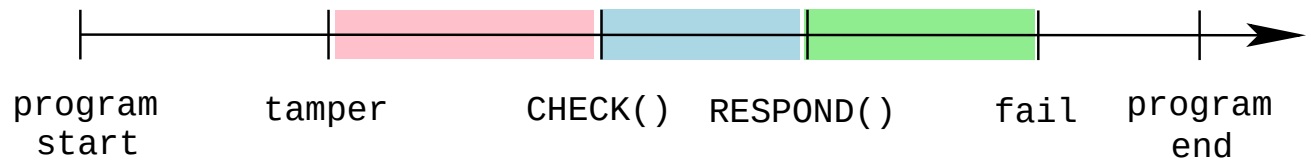

B_0
shll $2,%eax
xorl $90E98148,%ecx
incl %eax
C1 E0 02 81 F1 48 81 E9 90 40 81 C1
0 1 2 3 4 5 6 7 8 9 10 11
decl %eax
subl $C1814090,%ecx
B_1
addl $9003E8C1,%ecx
ret
81 C1 C1 E8 03 90 C3
10 11 12 13 14 15 16
shrl $3,%eax
nop
ret

Oblivious hashing

- Executing one block means also computing a hash over the other block into register `%ecx`!
- You can check the hash as usual.
- Clever use of the x86's architectural (mis-)features!
- Overhead: up to 3x slowdown.

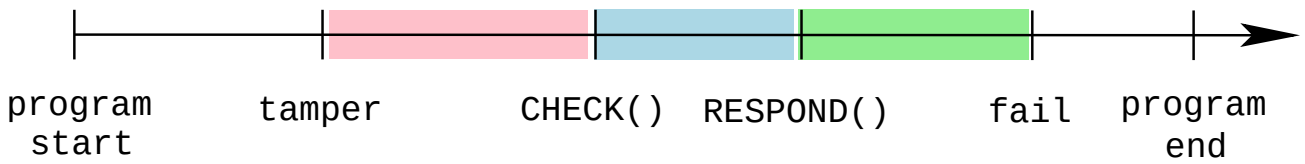
Response Mechanisms

Response Mechanisms



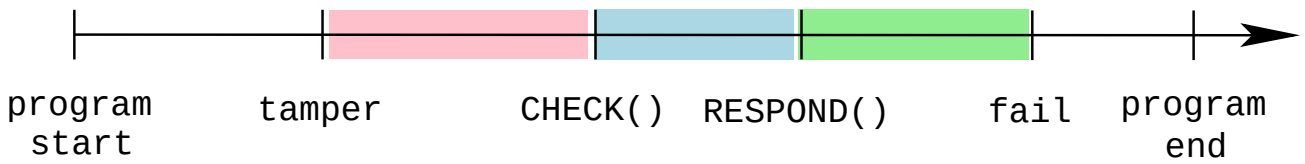
- `CHECK` checks for tampering,

Response Mechanisms



- `CHECK` checks for tampering,
- Later `RESPOND` takes action,

Response Mechanisms



- `CHECK` checks for tampering,
- Later `RESPOND` takes action,
- Later still, the program actually fails

Response Mechanisms

```
boolean tampered = false;
int global = 10;
          ...
if (hash(...)!=0xb1acca75) tampered = true;
          ...
if (tampered) global = 0;
          ...
printf("%i",10/global);
```

- `RESPOND` corrupts program state so that the actual failure follows much later

Response Mechanisms

```
#include <time.h>
int global = 10;
          ...
if (time(0) % 2 == 0)
   printf("%i",10/global);
          ...
if (getpid() % 2 == 0)
   x = 5/global;
          ...
x = 3/global;
```

- Introduce a number of failure sites and probabilistically choose between them.
- Every time the attacker runs the hacked program it is likely to fail in one of the two green spots.

Response Mechanisms

spatial separation: There should be as little static and dynamic connection between the `RESPOND` site and the failure site as possible.

Response Mechanisms

spatial separation: There should be as little static and dynamic connection between the `RESPOND` site and the failure site as possible.

temporal separation: A significant length of time should pass between the execution of `RESPOND` and the eventual failure.

Response Mechanisms

spatial separation: There should be as little static and dynamic connection between the `RESPOND` site and the failure site as possible.

temporal separation: A significant length of time should pass between the execution of `RESPOND` and the eventual failure.

stealth: The test, response, and failure code you insert in the program should be stealthy

Response Mechanisms

spatial separation: There should be as little static and dynamic connection between the `RESPOND` site and the failure site as possible.

temporal separation: A significant length of time should pass between the execution of `RESPOND` and the eventual failure.

stealth: The test, response, and failure code you insert in the program should be stealthy

predictability: Once the tamper response has been invoked, the program should eventually fail.

Response Mechanisms

- Think about legal implications of your tamper response mechanism!
- Don't deliberately destroy data...
- What if tamper-response was issued erroneously? ("I forgot my password, and after three tries the program destroyed my home directory!")
- Watch out for unintended consequences. (the program crashes with a file open...)

Response Mechanisms

- `RESPOND` to set a global pointer variable to `NULL`, causing the program to crash when the pointer is later dereferenced.

Response Mechanisms

- `RESPOND` to set a global pointer variable to `NULL`, causing the program to crash when the pointer is later dereferenced.
- If the program doesn't have enough pointer variables creates new ones by adding a layer of indirection to non-pointer variables.

Response Mechanisms

- `RESPOND` to set a global pointer variable to `NULL`, causing the program to crash when the pointer is later dereferenced.
- If the program doesn't have enough pointer variables creates new ones by adding a layer of indirection to non-pointer variables.
- Assumes that there are enough global variables to choose from.

```
int tampered=0;
int v;

void f() {
    v = 10;
}

void g() {
    f();
}

void h() {
}

int main() {
    if (...)
        tampered=1;
    h();
    g();
}
```

```
int tampered=0;
int v;
int *p_v = &v;

void f() {
    *p_v = 10;
}

void g() {
    f();
}

void h() {
}

int main() {
    if (...)
        tampered=1;
    h();
    g();
}
```

```
int tampered=0;
int v;
int *p_v = &v;

void f() {
    *p_v = 10;
}

void g() {
    f();
}

void h() {
    if (tampered)
        p_v = NULL;
}

int main() {
    if (...)
        tampered=1;
    h();
    g();
}
```

Example

1. Create a global pointer variable `p_v`.

Example

1. Create a global pointer variable `p_v`.
2. To make the program crash you should set `p_v` to `NULL`. But where?

Example

1. Create a global pointer variable `p_v`.
2. To make the program crash you should set `p_v` to `NULL`. But where?
3. You want to avoid `g` and `main` since they will be on the call stack when `f` throws the *pointer-reference-to-nil* exception. (Check the stacktrace.)

Example

1. Create a global pointer variable `p_v`.
2. To make the program crash you should set `p_v` to `NULL`. But where?
3. You want to avoid `g` and `main` since they will be on the call stack when `f` throws the *pointer-reference-to-nil* exception. (Check the stacktrace.)
4. Insert the failure-inducing code in `h` which is "many" calls away and not in the same call-chain as `f`.

Discussion

Trustworthiness

- Tamperproofing is about trustworthiness:
  - Can I trust my program when it's running on an untrusted site?

Trustworthiness

- Tamperproofing is about trustworthiness:
  - Can I trust my program when it's running on an untrusted site?
- For us to trust *P*, the adversary
  - cannot add/remove/change *P*'s code!
  - cannot modify *P*'s environment!

Trustworthiness

- Tamperproofing is about trustworthiness:
  - Can I trust my program when it's running on an untrusted site?
- For us to trust *P*, the adversary
  - cannot add/remove/change *P*'s code!
  - cannot modify *P*'s environment!
- Essential for DRM, network gaming,...

Basic operations

- Check *P*'s environment:
  - Am I running under a debugger?
  - Am I running under emulation?
  - Has the OS been hacked?

Basic operations

- Check *P*'s environment:
  - Am I running under a debugger?
  - Am I running under emulation?
  - Has the OS been hacked?
- Check *P*'s code:
  - Have the executable bits been changed?

Basic operations

- Check *P*'s environment:
  - Am I running under a debugger?
  - Am I running under emulation?
  - Has the OS been hacked?
- Check *P*'s code:
  - Have the executable bits been changed?
- Check *P*'s dynamic data:
  - Is *P* in a legal executable state?

In practice. . .

- Use a combination of operations!
  - Check the environment
  - Check the code
  - Check the state

In practice. . .

- Use a combination of operations!
  - Check the environment
  - Check the code
  - Check the state
- You must check the checking code!
  - Simple attack: remove the checkers!

In practice. . .

- Use a combination of operations!
  - Check the environment
  - Check the code
  - Check the state
- You must check the checking code!
  - Simple attack: remove the checkers!
- The response must be stealthy!
  - Simple attack: trace back from failure!

In practice. . .

- Use a combination of operations!
  - Check the environment
  - Check the code
  - Check the state
- You must check the checking code!
  - Simple attack: remove the checkers!
- The response must be stealthy!
  - Simple attack: trace back from failure!
- The detection must be stealthy!
  - Simple attack: detect reads of executable pages!

Future Lectures

1. NO LECTURE April 9!
2. NO LECTURE April 16!
3. TWO MORE LECTURES!
4. One lecture on hardware protection.
5. One lecture on software watermarking/birthmarking/similarity (maybe).

Website

1. Please check the website for important announcements:

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

`Teaching/mgu/2014`