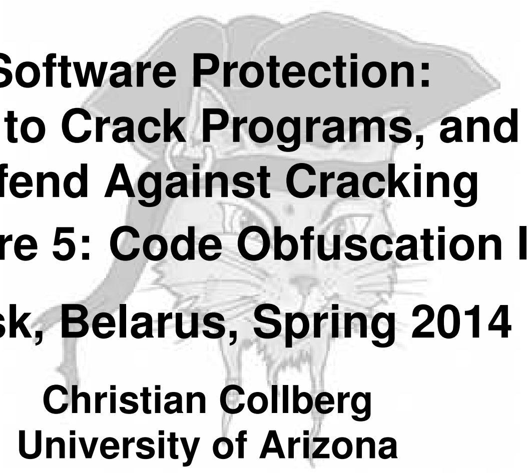


Software Protection: How to Crack Programs, and Defend Against Cracking

Lecture 5: Code Obfuscation II

Minsk, Belarus, Spring 2014

Christian Collberg
University of Arizona

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

Dynamic Obfuscation

Static vs. Dynamic obfuscation

- Static obfuscations transform the code prior to execution.

Static vs. Dynamic obfuscation

- Static obfuscations transform the code prior to execution.
- Dynamic algorithms transform the program at runtime.

Static vs. Dynamic obfuscation

- Static obfuscations transform the code prior to execution.
- Dynamic algorithms transform the program at runtime.
- Static obfuscation counter attacks by static analysis.

Static vs. Dynamic obfuscation

- Static obfuscations transform the code prior to execution.
- Dynamic algorithms transform the program at runtime.
- Static obfuscation counter attacks by static analysis.
- Dynamic obfuscation counter attacks by dynamic analysis.

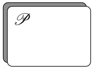
Dynamic Obfuscation: Definitions

- A dynamic obfuscator runs in two phases:
  1. At compile-time transform the program to an initial configuration and add a runtime code-transformer.
  2. At runtime, intersperse the execution of the program with calls to the transformer.

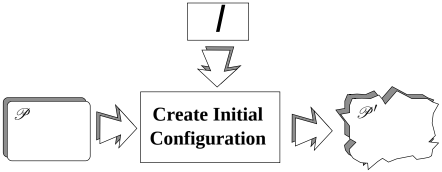
Dynamic Obfuscation: Definitions

- A dynamic obfuscator runs in two phases:
  1. At compile-time transform the program to an initial configuration and add a runtime code-transformer.
  2. At runtime, intersperse the execution of the program with calls to the transformer.
- A dynamic obfuscator turns a “normal” program into a self-modifying one.

Modeling dynamic obfuscation — compile-time

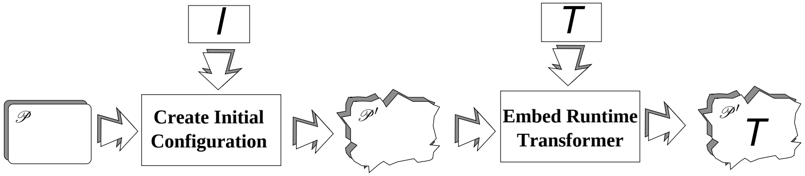


Modeling dynamic obfuscation — compile-time



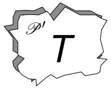
- Transformer *I* creates $\mathscr{P}$'s initial configuration.

Modeling dynamic obfuscation — compile-time



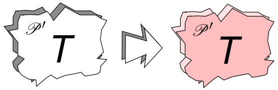
- Transformer I creates $\mathscr{P}$'s initial configuration.
- T is the runtime obfuscator, embedded in $\mathscr{P}'$.

Modeling dynamic obfuscation — runtime



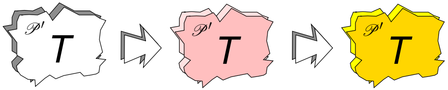
- Transformer T continuously modifies $\mathscr{P}'$ at runtime.

Modeling dynamic obfuscation — runtime



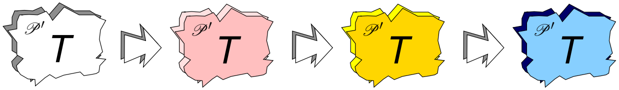
- Transformer T continuously modifies $\mathscr{P}'$ at runtime.

Modeling dynamic obfuscation — runtime



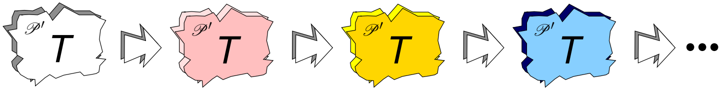
- Transformer T continuously modifies $\mathscr{P}'$ at runtime.

Modeling dynamic obfuscation — runtime



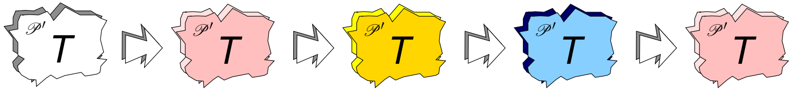
- Transformer T continuously modifies $\mathscr{P}'$ at runtime.

Modeling dynamic obfuscation — runtime



- Transformer T continuously modifies $\mathscr{P}'$ at runtime.

Modeling dynamic obfuscation — runtime



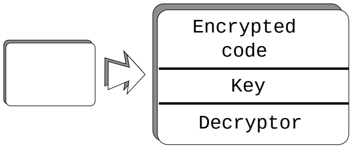
- Transformer T continuously modifies $\mathscr{P}'$ at runtime.
- We'd like an infinite, non-repeating series of configurations.
- In practice, the configurations repeat.

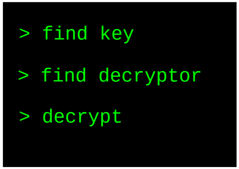
Algorithm Ideas

Basic algorithm ideas

- Build-and-execute: generate code for a routine at runtime, and then jump to it.
- Self-modification: modify the executable code.
- Encryption: The self-modification is decrypting the encrypted code before executing it.
- Move code: Every time the code executes, it is in different location.

File-Level Encryption: Packers

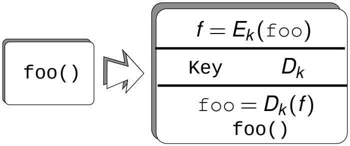




```
> find key

> find decryptor

> decrypt
```

- Packers are simple tools that encrypt the binary, and include a routine that will decrypt at runtime.

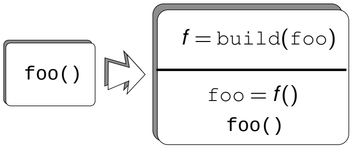
Function-Level Encryption

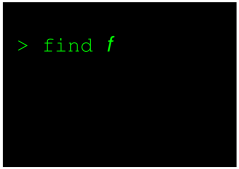


```
> find key
> find D_k
> find f
> decrypt
```

- You can also decrypt a function just before it gets called.

Build-And-Execute

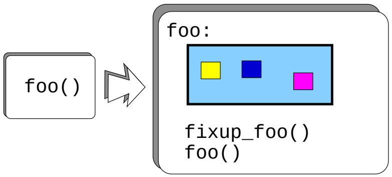


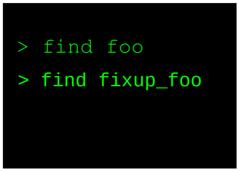


```
> find f
```

- You can generalize “encryption” to any embedded function that constructs the “real” code at runtime.

Self-Modifying Code

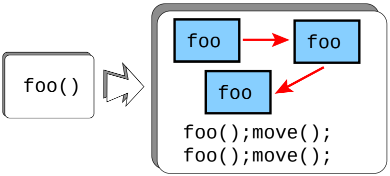




```
> find foo
> find fixup_foo
```

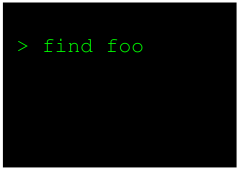
- Leave “holes” in `foo`, fix them just before `foo` gets called.

Move Code Around



```
foo()
```

```
foo();move();
foo();move();
```



```
> find foo
```

- Continously move code around to make it harder to find.

Granularity

- These operations can be applied at different levels of granularity:
  - File-level
  - Function-level
  - Basic block-level
  - Instruction-level

Attack Goals

- The attacker's goal can be to:
  - recover the original code
  - modify the original code

```
int modexp(int y, int x[], int w, int n, int mode) {
    int R, L, k = 0, s = 1, t;
    char* p=&&begin;
    while (p<(char*)&&end) *p++ ^= 99;
    if (mode==1) return 0;
    while (k < w) {
        begin:
            ... ... ...
            ... ... ...
        end:
        k++;
    }
    p=&&begin; while (p<(char*)&&end) *p++ ^= 99;
    return L;
}
int main() {
    makeCodeWritable(···);
    modexp(0, NULL, 0, 0, 1);
    ...
    modexp(···, ···, ···, ···, 0);
}
```

Code Explanation

- The blue code is xor:ed with a key (99).
- When the code is to be executed it gets “decrypted”, executed, and re-encrypted.
- The green code would normally execute at obfuscation time.
- Every subsequent time the `modexp` routine gets called the pink code first decrypts the blue code, executes it, and then the yellow code re-encrypts it.

Practical issues

- Pages have to be modifiable and executable. (See next slide).
- You have to flush the CPU's data cache before executing new code you have generated. (Why?) X86 does this automatically.

```
void makeCodeWritable(caddr_t first, caddr_t last) {
   caddr_t firstpage =
      first - ((int)first % getpagesize());
   caddr_t lastpage =
      last - ((int)last % getpagesize());
   int pages=(lastpage-firstpage)/getpagesize()+1;
   if (mprotect(
         firstpage,
         pages*getpagesize(),
         PROT_READ|PROT_EXEC|PROT_WRITE
       )==-1)
          perror("mprotect");
}
```

Decrypting by Emulation

- “Encrypting” binaries is often re-invented!
- Attack: run the program inside an emulator that prints out every executed instruction.
- The instruction trace can be analyzed (re-rolling loops, removing decrypt-and-jump artifacts, etc.) and the original code recovered.

Replacing Instructions

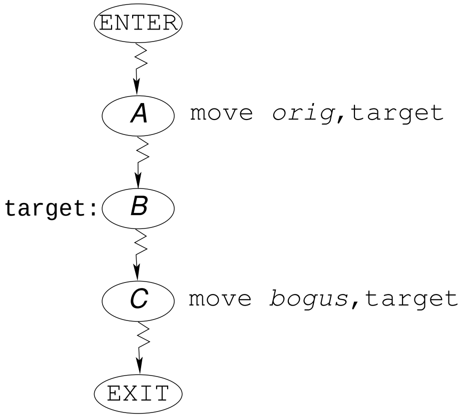
Kanzaki's Algorithm

- Motivation: make it hard for the adversary to snapshot the code.
- Idea: replace real instructions by bogus ones.
- Right before execution, the bogus instruction is replaced by the real one.
- Just after execution, the real instruction is replaced by the bogus one!

```
int player_main (int argc, char *argv[]) {
    char orig = (*(caddr_t)&&target);
    (*(caddr_t)&&target) = 0;

    ... ... ...

    for(i=0;i<len;i++) {
        (*(caddr_t)&&target) = orig;

        ... ... ...

        target:
        printf("%f\n",decoded);
        (*(caddr_t)&&target) = 0;
    }
}
int main (int argc, char *argv[]) {
    makeCodeWritable(...);
    player_main(argc,argv);
}
```

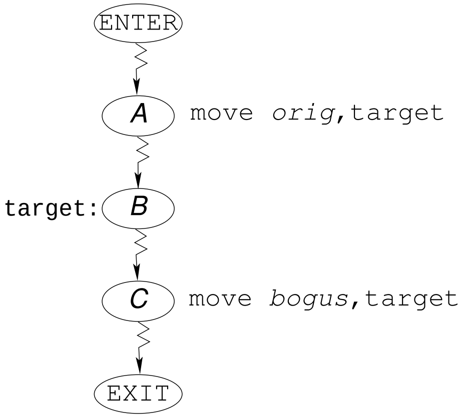
Algorithm Details

- Find three points *A*, *B*, *C* in the control flow graph:



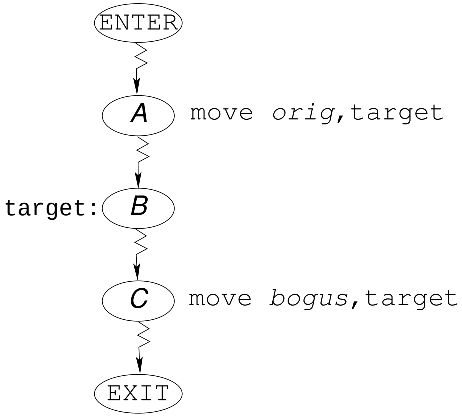
Algorithm Details

- Every path to *B* must flow through *A* and every path from *B* must flow through *C*:



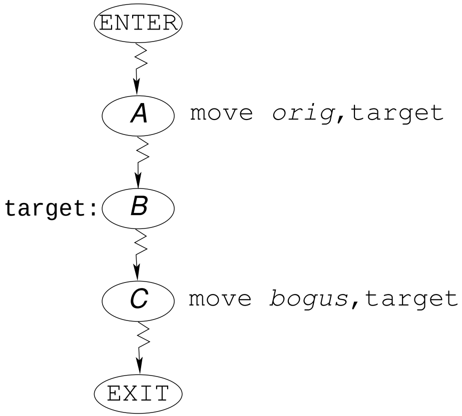
Algorithm Details

- At *A*: insert an instruction which overwrites the target instruction with its original value:



Algorithm Details

- At *C*: insert an instruction which overwrites the target with the bogus value:



Attack: set pages unwritable!

- The attacker calls `mprotect` to set the code region to readable and executable, but not writable. (See next slide).
- When the program tries to write into the code stream the operating system throws an exception.
- Under debugging, see where this happens!

```
(gdb) call (int)mprotect(0x2000,0x3000,5)
(gdb) cont
EXC_BAD_ACCESS, Could not access memory.
KERN_PROTECTION_FAILURE at address: 0x00002934
0x000028c0 in player_main
30            (*(caddr_t)&&target) = orig;
(gdb) x/i $pc
0x28c0 <player_main+220>:       stb     r0,0(r2)
(gdb) print (char)$r0
$7 = -64
(gdb) print/x (int)$r2
$10 = 0x2934
```

Code Merging

Madou’s Algorithm: Dynamic Code Merging

- Motivation: Keep the program in constant flux!

Madou’s Algorithm: Dynamic Code Merging

- Motivation: Keep the program in constant flux!
- Every time the adversary looks at the code, it’s different!

Madou's Algorithm: Dynamic Code Merging

- Motivation: Keep the program in constant flux!
- Every time the adversary looks at the code, it's different!
- Idea: Two or more functions *share* the same location in memory!

Madou's Algorithm: Dynamic Code Merging

- Motivation: Keep the program in constant flux!
- Every time the adversary looks at the code, it's different!
- Idea: Two or more functions *share* the same location in memory!
- Before *f* is called, patch memory to ensure *f* is loaded.

Example: Original Code

- Obfuscate a program that contains two functions f_1 and f_2:

f_1	
0	10
1	5
2	6
3	20
4	99

f_2	
0	10
1	9
2	3
3	20

- To the left is byte index in the function, to the right the code byte at the location.
- Note: At index 0, both f_1 and f_2 have the same code byte (10).

Example: Obfuscation Time

- During obfuscation replace f_1 and f_2 with the template T and two edit scripts e_1 and e_2:

T

0	10
1	?
2	?
3	20
4	99

$$\begin{aligned} e_1 &= [1 \to 5, 2 \to 6] \\ e_2 &= [1 \to 9, 2 \to 3] \end{aligned}$$

Example: Calling $f_1()$ at Run Time

- Program calls $f_1()$: patch T using e_1.
- Replace the code-byte at offset 1 with 5 and the code-byte at offset 2 with 6.

T

0	10
1	?
2	?
3	20
4	99

$$\begin{aligned} e_1 &= [1 \rightarrow 5, 2 \rightarrow 6] \\ e_2 &= [1 \rightarrow 9, 2 \rightarrow 3] \end{aligned}$$

Example: Calling $f_1()$ at Run Time

- If you call f_1 again (without intervening calls to f_2), no need to patch!!!

T

0	10
1	?
2	?
3	20
4	99

$$\begin{aligned} e_1 &= [1 \to 5, 2 \to 6] \\ e_2 &= [1 \to 9, 2 \to 3] \end{aligned}$$

Example: Calling $f_2()$ at Run Time

- If you call f_1 again (without intervening calls to f_2), no need to patch!!!
- Program calls $f_2()$: patch T using e_2.
- T memory region will constantly change, first containing an incomplete function and then alternating between containing the code-bytes for f_1 and f_2.

Algorithm step 1: Clustering

- Decide which functions should be in the same *cluster*, i.e. reside in the same template at runtime.

Algorithm step 1: Clustering…

- Avoid putting f_1 and f_2 in the same cluster if they are called like this:

```
while(1) {
    f1();
    f2();
}
```

Algorithm step 2: Make scripts and patch routine

- Create a template T_k containing the intersection of the code-bytes of the functions in c_k.
- For each function f_i in c_k create an edit script e_i such that applying e_i to the code-bytes of T_k creates the code-bytes of f_i.

Dynamic Code Merging

- Original code:

```
int val = 0;
void f1(int* v) {*v=99;}
void f2(int* v) {*v=42;}
int main (int argc, char *argv[]) {
   f1(&val);
   f2(&val);
}
```

```
EDIT script1[200], script2[200];
char* template;
int template_len, script_len = 0;
typedef void(*FUN)(int*);
int val, state = 0;

void f1_stub() {
   if (state != 1) {
      patch(script1,script_len,template); state = 1;}
   ((FUN)template)(&val);
}
void f2_stub() {
   if (state != 2) {
      patch(script2,script_len,template); state = 2;}
   ((FUN)template)(&val);
}
int main (int argc, char *argv[]) {
   f1_stub(); f2_stub();
}
```

Attacks

- Note: the `patch` routine is in the clear!

Attacks

- Note: the `patch` routine is in the clear!
- Note: the *scripts* are in the clear!

Attacks

- Note: the `patch` routine is in the clear!
- Note: the *scripts* are in the clear!
- Static attack:
  1. Analyze binary, find `patch` routine an scripts.
  2. Running each call to `patch`(T_k, e_i) to recover the code!

Attacks

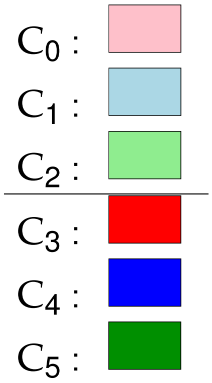
- Note: the `patch` routine is in the clear!
- Note: the *scripts* are in the clear!
- Static attack:
  1. Analyze binary, find `patch` routine an scripts.
  2. Running each call to `patch`(T_k, e_i) to recover the code!
- Counterattack: Encrypt the scripts.

Attacks

- Note: the `patch` routine is in the clear!
- Note: the *scripts* are in the clear!
- Static attack:
  1. Analyze binary, find `patch` routine an scripts.
  2. Running each call to `patch`(T_k, e_i) to recover the code!
- Counterattack: Encrypt the scripts.
- Counter-counterattack: Intercept the decrypted scripts at runtime.

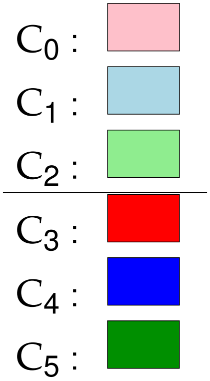
Self-Modifying State Machine

Aucsmith's algorithm



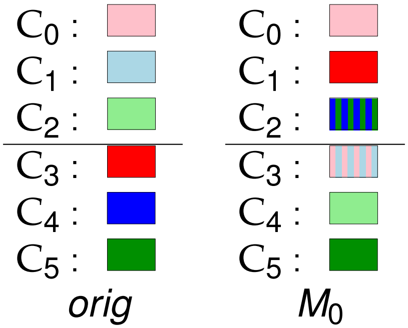
- A function is split into cells.

Aucsmith's algorithm

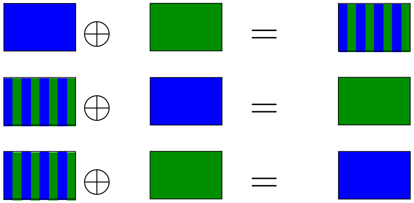


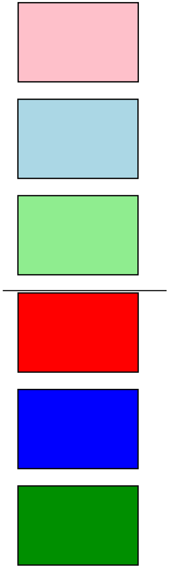
- A function is split into cells.
- The cells are divided into two regions in memory, upper and lower.

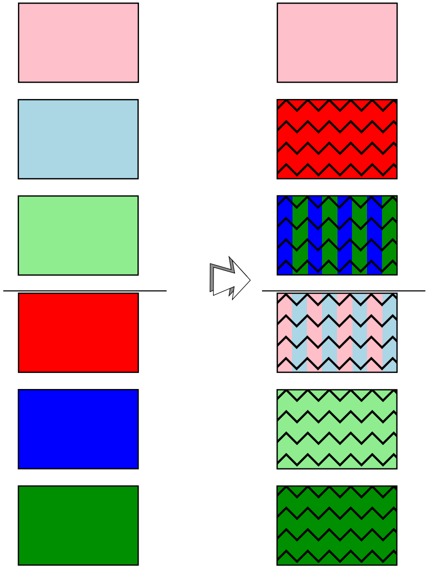
One step

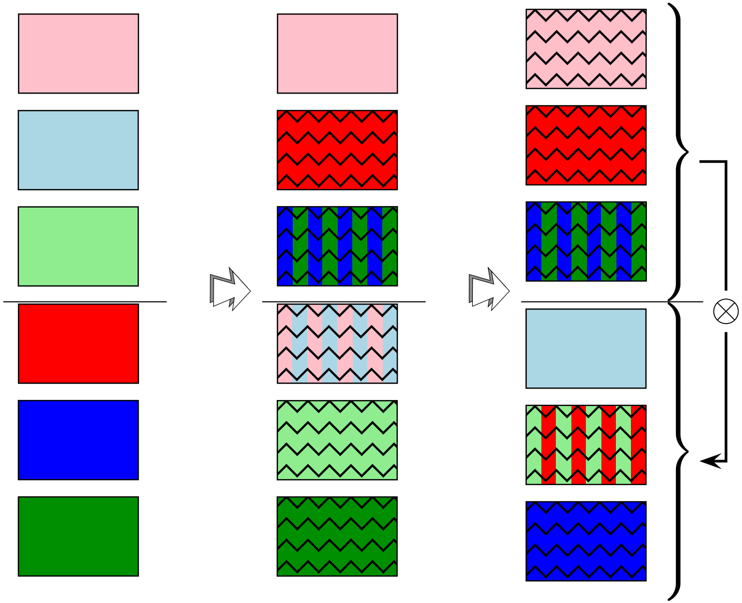


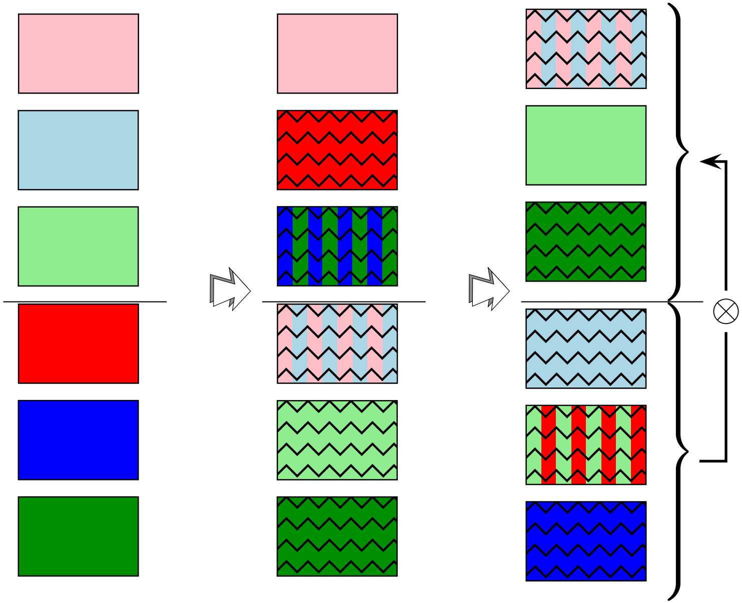
XOR!

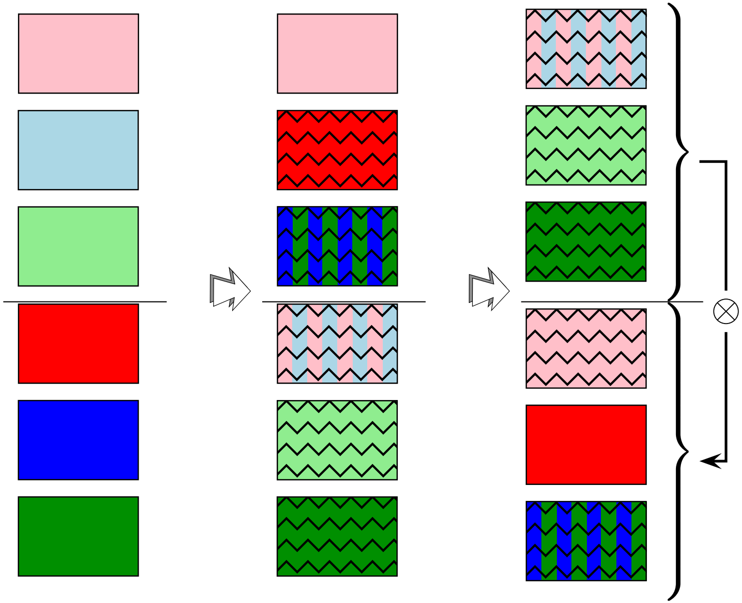


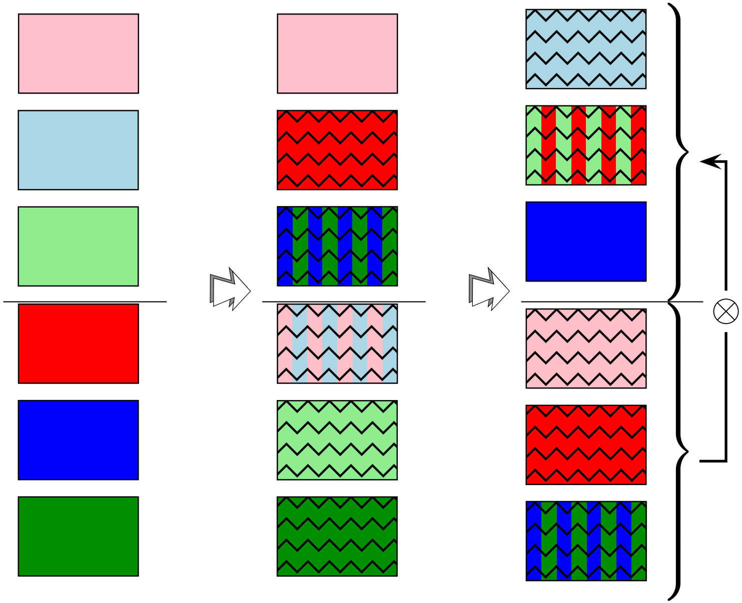


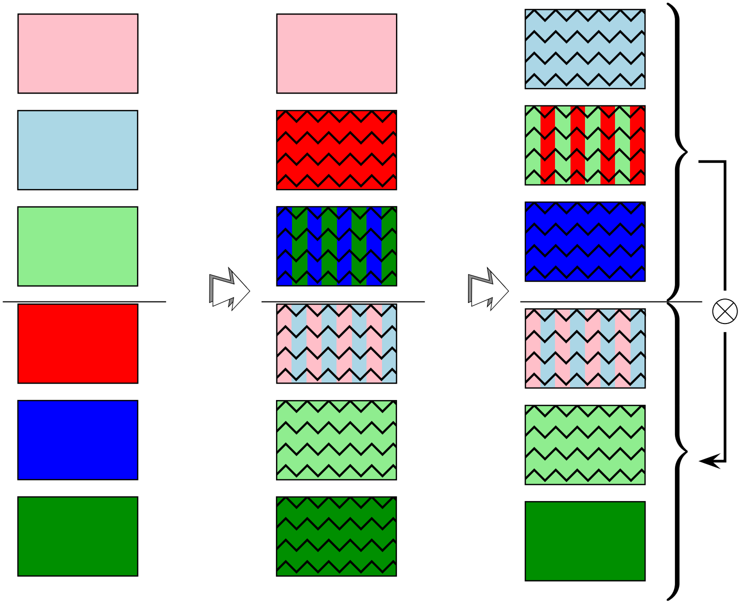


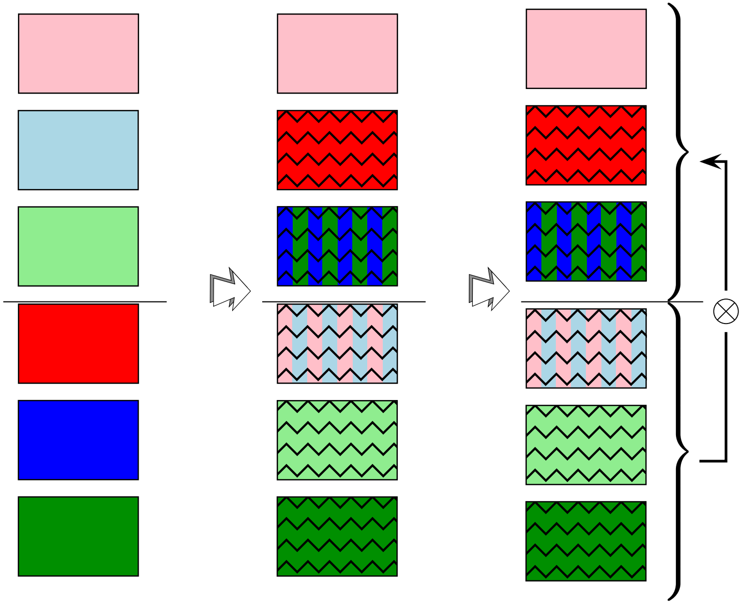




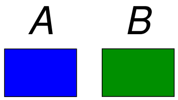




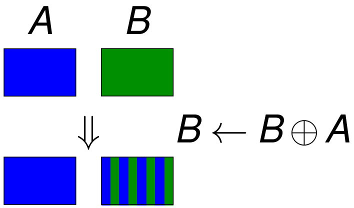




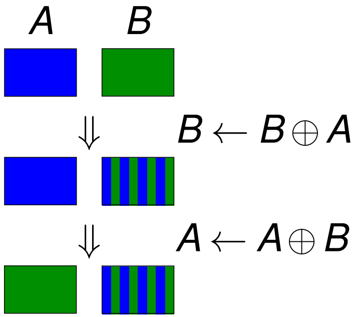
Why does this work?



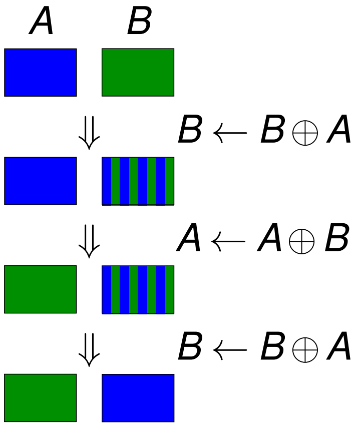
Why does this work?



Why does this work?



Why does this work?


A
B
$B \leftarrow B \oplus A$
$A \leftarrow A \oplus B$
$B \leftarrow B \oplus A$

Runtime Encryption

Code as key material

- Encrypt the code to keep as little code as possible in the clear at any point in time during execution.

Code as key material

- Encrypt the code to keep as little code as possible in the clear at any point in time during execution.
- Extremes:
    1. Decrypt the next instruction, execute it, re-encrypt it, ... ⇒ only one instruction is ever in the clear!

Code as key material

- Encrypt the code to keep as little code as possible in the clear at any point in time during execution.
- Extremes:
  1. Decrypt the next instruction, execute it, re-encrypt it, . . . $\Rightarrow$ only one instruction is ever in the clear!
  2. Decrypt the entire program once, prior to execution, and leave it in cleartext. $\Rightarrow$ easy for the adversary to capture the code.

Code as key material

- The entire program is encrypted — except for `main`.

Code as key material

- The entire program is encrypted — except for `main`.
- Before you jump to a function you decrypt it.

Code as key material

- The entire program is encrypted — except for `main`.
- Before you jump to a function you decrypt it.
- When the function returns you re-encrypt it.

Code as key material

- The entire program is encrypted — except for `main`.
- Before you jump to a function you decrypt it.
- When the function returns you re-encrypt it.
- On entry, a function first encrypts its caller.

Code as key material

- The entire program is encrypted — except for `main`.
- Before you jump to a function you decrypt it.
- When the function returns you re-encrypt it.
- On entry, a function first encrypts its caller.
- Before returning, a function decrypts its caller.

Code as key material

- The entire program is encrypted — except for `main`.
- Before you jump to a function you decrypt it.
- When the function returns you re-encrypt it.
- On entry, a function first encrypts its caller.
- Before returning, a function decrypts its caller.
- $\Rightarrow$ At most two functions are ever in the clear!

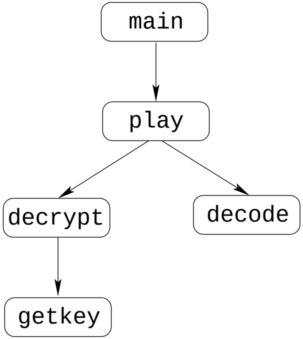
Code as key material

- What do we use as key? The code itself!

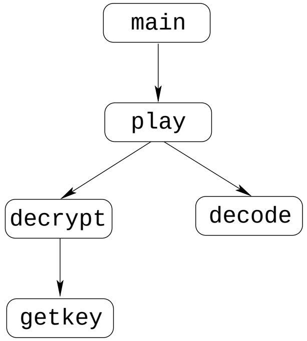
Code as key material

- What do we use as key? The code itself!
- What cipher do we use?
  Something simple!

- Simple case: tree-shaped call-graph:

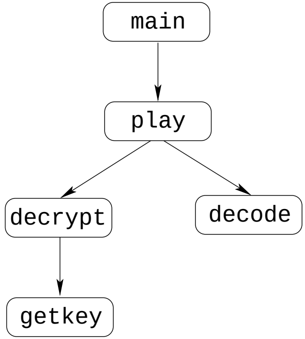


- Simple case: tree-shaped call-graph:



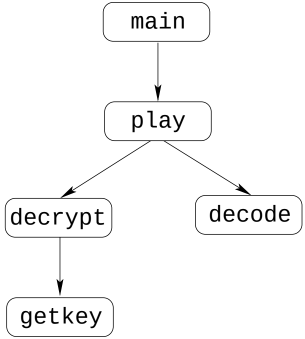
- Before/after procedure call: call guard function to decrypt/re-encrypt the callee.

- Simple case: tree-shaped call-graph:



- Before/after procedure call: call guard function to decrypt/re-encrypt the callee.
- Entry/exit of the callee: encrypt/decrypt the caller.

- Simple case: tree-shaped call-graph:



- Before/after procedure call: call guard function to decrypt/re-encrypt the callee.
- Entry/exit of the callee: encrypt/decrypt the caller.
- Key: Hash of the cleartext of the caller/callee.

```
int player_main (int argc, char *argv[]) {
   int user_key = 0xca7ca115;
   int digital_media[] = {10,102};
   guard(play,playSIZE,player_main,player_mainSIZE);
   play(user_key,digital_media,2);
   guard(play,playSIZE,player_main,player_mainSIZE);
}
int getkey(int user_key) {
   guard(decrypt,decryptSIZE,getkey,getkeySIZE);
   int player_key = 0xbabeca75;
   int v = user_key ^ player_key;
   guard(decrypt,decryptSIZE,getkey,getkeySIZE);
   return v;
}
int decrypt(int user_key, int media) {
   guard(play,playSIZE,decrypt,decryptSIZE);
   guard(getkey,getkeySIZE,decrypt,decryptSIZE);
   int key = getkey(user_key);
   guard(getkey,getkeySIZE,decrypt,decryptSIZE);
   int v = media ^ key;
   guard(play,playSIZE,decrypt,decryptSIZE);
   return v;
```

```
float decode (int digital) {
    guard(play,playSIZE,decode,decodeSIZE);
    float v = (float)digital;
    guard(play,playSIZE,decode,decodeSIZE);
    return v;
}
void play(int user_key, int digital_media[], int len) {
    int i;
    guard(player_main,player_mainSIZE,play,playSIZE);
    for(i=0;i<len;i++) {
        guard(decrypt,decryptSIZE,play,playSIZE);
        int digital = decrypt(user_key,digital_media[i]);
        guard(decrypt,decryptSIZE,play,playSIZE);

        guard(decode,decodeSIZE,play,playSIZE);
        printf("%f\n",decode(digital));
        guard(decode,decodeSIZE,play,playSIZE);
    }
    guard(player_main,player_mainSIZE,play,playSIZE);
}
```

```
void crypto (waddr_t proc,uint32 key,int words) {
   int i;
   for(i=1; i<words; i++) {
      *proc ^= key;
      proc++;
   }
}

void guard (waddr_t proc,int proc_words,
            waddr_t key_proc,int key_words) {
   uint32 key = hash1(key_proc,key_words);
   crypto(proc,key,proc_words);
}
```

Discussion

Code Obfuscation — What's it Good For?

- Diversification — make every program unique to prevent malware attacks

Code Obfuscation — What's it Good For?

- Diversification — make every program unique to prevent malware attacks
- Prevent collusion — make every program unique to prevent diffing attacks

Code Obfuscation — What's it Good For?

- Diversification — make every program unique to prevent malware attacks
- Prevent collusion — make every program unique to prevent diffing attacks
- Code Privacy — make programs hard to understand to protect algorithms

Code Obfuscation — What's it Good For?

- Diversification — make every program unique to prevent malware attacks
- Prevent collusion — make every program unique to prevent diffing attacks
- Code Privacy — make programs hard to understand to protect algorithms
- Data Privacy — make programs hard to understand to protect secret data (keys)

Code Obfuscation — What's it Good For?

- Diversification — make every program unique to prevent malware attacks
- Prevent collusion — make every program unique to prevent diffing attacks
- Code Privacy — make programs hard to understand to protect algorithms
- Data Privacy — make programs hard to understand to protect secret data (keys)
- Integrity — make programs hard to understand to make them hard to change