Pierrick Brunet
Serge Guelton
Adrien Guinet
Juan Manuel Martinez

Challenges when building an LLVM-based obfuscator

quarkslab
SECURING EVERY BIT OF YOUR DATA

Table of Contents

Table of Contents

Security through Obscurity

> *"Code obfuscation is transforming the software program into code that's difficult to disassemble and understand, but has the same functionality as the original."*
>
> – *Wikipedia*

Security through Obscurity

"Code obfuscation is transforming the software program into code that's difficult to disassemble and understand, but has the same functionality as the original."

– *Wikipedia*

"Obfuscated "source code" is not real source code and does not count as source code."

– www.gnu.org/philosophy/free-sw.html

Obfuscation: Holy Grail

"An access to the binary does no yield more information than what can be observed from the output of the binary"

White Box Analysis Black Box Analysis

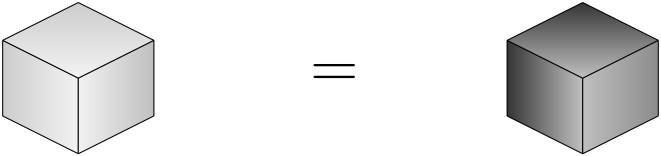
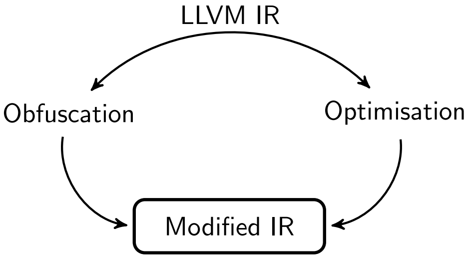


Table of Contents

Architecture

Obfuscations are mainly done on the LLVM **Internal Representation**



Obfuscation = Optimisation = LLVM pass

Architecture

Obfuscations are mainly done on the LLVM **Internal Representation**

Advantages (in theory)

- Language-agnostic obfuscations
- Backend independent obfuscations

Disadvantages

- Some CPU-specific tricks can't be implemented in a generic way
- Some information are not available at IR level (function size, function pointers value, ...)

Flow of compilation

Mixing optimisations and obfuscations

- First rule: obfuscations must *survive* LLVM optimisations
- Performance is important: run classical LLVM optimisations first
- Then obfuscations are applied
- And a post optimisations pass is done

Flow of compilation

Mixing optimisations and obfuscations

Trust me, I have a Graph

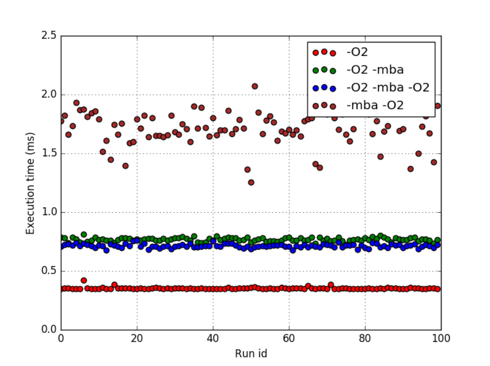


Table of Contents

Table of Contents

Obfuscate Predicates

Replace constants by computations depending on the context (`r`).
Example:

```
42 == (r|58) & (~5) & 47
```

Advantages

- Can be fully implemented at IR level
- Is language and backend independant
- Even works for vectorized operations

Obfuscate Predicates

Which context value to choose ? x or i ?

```
int x = ... ;
for(int i = 0; i < n; ++i)
  s += 42;
```

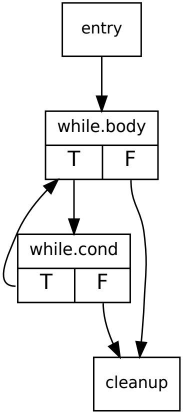
Problem of randomness

- Based on a fixed random-seed to enable reproductibility
- Unguaranteed performance reproductibility across seeds

Table of Contents

Control-Flow-Graph Flattening

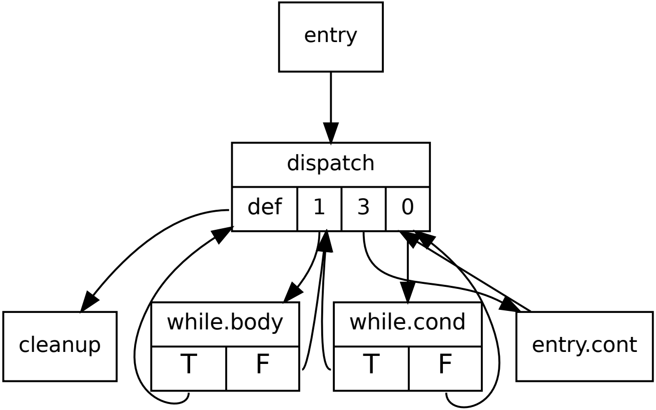
Transform all branches to jumps to a dispatcher with a switch statement:



CFG for 'check' function

Control-Flow-Graph Flattening

Transform all branches to jumps to a dispatcher with a switch statement:



CFG for 'check' function

Control-Flow-Graph Flattening

Windows Exceptions

Windows exceptions impose restrictions on the CFG:

- Treat blocks with the same parent exception pad as belonging to the same *set*.
- Invokes, exception-handling pads and exception-handling returns edges are left as they are.
- Flatten each *set*.

Table of Contents

Anti-Debug/Jailbreak/Emulator

Check the executable's environment to detect jailbroken devices:

- Periodic checks injected in the code
- Startup checks (platform dependent!)
- Implement the checks in C

```c
bool test_jailbreak() {
  FILE *fp;
  const char path[] = "/private/random_name";
  if((fp = fopen(path, "w")) != NULL) {
    fclose(fp);
    unlink(path);
    return true; // detected
  }
  return false; // not detected
}
```

Anti-Debug/Jailbreak/Emulator

Inserting checks at startup is platform dependant:

Problem

Windows:

- LLVM's `global_ctors` priorities are broken on Windows
- Sections `.CRT$XCA` … `.CRT$XCZ`
- TLS constructors, executed for each thread, even before initializing the CRT

Anti-Debug/Jailbreak/Emulator

Implement the complex checks in C .

Problem

- Cannot rely on calls to a library: easy to identify and isolate, and can't be obfuscated by the user
- Really hard to pre-generate the IR for every target platform

Solution: *clang-ception*

Compile C code on demand by using clang within clang.

- Shared resources: `LLVMContext` (and global variables, ugh!)
- The user can write its own checks
- Can easy apply obfuscations on the C code

Table of Contents

Table of Contents

Code integrity

On the previous anti-jailbreak test function...

```c
bool test_jailbreak() {
  FILE *fp;
  const char path[] = "/private/random_name";
  if((fp = fopen(path, "w")) != NULL) {
    fclose(fp);
    unlink(path);
    return true; // detected
  }
  return false; // not detected
}
```

How to check the function hasn't been modified?

Goal of code integrity

Verify code wasn't tampered *a priori* / at runtime

Code integrity checks

At the binary level

- One check at startup $\Rightarrow$ easy to remove
- Injection at various places $\Rightarrow$ potential performance issues:
  - using siphash 1-2 (non-linear *fast* hash)
  - using an Intel i7-6700HQ CPU: $\sim$ 0.7 cycles per byte
  - on a 60MiB binary (like clang): $\sim$ 44M cycles => $\sim$ 20 ms at 2Ghz

$\Rightarrow$ We also need function-level integrity checks!

Code integrity checks

At the binary level

- One check at startup $\Rightarrow$ easy to remove
- Injection at various places $\Rightarrow$ potential performance issues:
  - using siphash 1-2 (non-linear *fast* hash)
  - using an Intel i7-6700HQ CPU: $\sim$ 0.7 cycles per byte
  - on a 60MiB binary (like clang): $\sim$ 44M cycles => $\sim$ 20 ms at 2Ghz

$\Rightarrow$ We also need function-level integrity checks!

At the function level

- Hashes each function individually
- Check hashes at the beginning of each function, before each call sites...
- Cross check functions between them

Code integrity: function level

Basic idea/example

```
void foo() {
  puts("hello world!");
}
```

becomes:

```
static unsigned hashes[] = { ... };
static unsigned hash(void* begin, void* end) { ... }
void foo() {
  if (hash(&&begin, &&end) != hashes[foo_id])
    exit(...);
begin:
  puts("hello world!");
end:
}
```

Code integrity: function level

Basic idea/example

```
void foo() {
  puts("hello world!");
}
```

becomes:

```
static unsigned hashes[] = { ... };
static unsigned hash(void* begin, void* end) { ... }
void foo() {
  if (hash(&&begin, &&end) != hashes[foo_id])
    exit(...);
begin:
  puts("hello world!");
end:
}
```

Doing this at IR time: pros

- Cross-language/platform way to insert the hash function (using *clang-ception*)
- Easier to insert checks than at backend time

Code integrity: function level

Basic idea/example

```
void foo() {
  puts("hello world!");
}
```

becomes:

```
static unsigned hashes[] = { ... };
static unsigned hash(void* begin, void* end) { ... }
void foo() {
  if (hash(&&begin, &&end) != hashes[foo_id])
    exit(...);
begin:
  puts("hello world!");
end:
}
```

Doing this at IR time: cons

- How to get a pointer to the end of a function?
- Obviously, the assembly code of functions isn't available
  $\Rightarrow$ IR$\Leftrightarrow$backend cooperation!

Issue with end-of-function

The problem

No object in the LLVM IR is associated with the *end* of a function

A solution...

Issue with end-of-function

The problem

No object in the LLVM IR is associated with the *end* of a function

A solution...

`llvm::BlockAddress` to the rescue!

- `llvm::Constant`, lowers to the address of a block within a function.
- Extension: `blockaddress(null, foo)` as the address of the `end` label for the `foo` function.
- Add `endfuncptr` in the LLVM-IR format

Problems:

- Incompatible modification of the LLVM IR format:
  - Potential problem for iOS apps!
- Is a hack...

IR-backend cooperation

General idea

- Put placeholders for function hash values
- Use a post-processing tool that "fixes" the placeholders

Issues

- Need a cross-platform/cross-format (PE,ELF,Mach0) tool (free-ad: LIEF [1] is a good framework for this!)
- Not trivial to obfuscate hashes values and function pointers

[1] https://github.com/quarkslab/LIEF

Code integrity: remaining problems

Non exhaustive list of gotcha

- Dynamic relocations within code
- Assume function code is contiguous: technically no guarantee in LLVM
- Relies on undefined behavior: pointer arithmetic and dereferencing a function pointer is UB
- C function pointer != beginning of function code

Table of Contents

The JIT way

Idea

Use the LLVM jitter to generate binary code of a function at IR-time

Benefits

Drawbacks

The JIT way

Idea

Use the LLVM jitter to generate binary code of a function at IR-time

Benefits

- A function becomes an array of bytes, treated (almost) as any other data array
- *Free* integrity checks
- *Free* on-the-fly decryption/reencryption

Drawbacks

The JIT way

Idea

Use the LLVM jitter to generate binary code of a function at IR-time

Benefits

- A function becomes an array of bytes, treated (almost) as any other data array
- *Free* integrity checks
- *Free* on-the-fly decryption/reencryption

Drawbacks

- Final symbols address unknown at IR-time
- C++ exception frames to register "by hand"
- NX-bit protection

The JIT way

Idea

Use the LLVM jitter to generate binary code of a function at IR-time

Benefits

- A function becomes an array of bytes, treated (almost) as any other data array
- *Free* integrity checks
- *Free* on-the-fly decryption/reencryption

Drawbacks

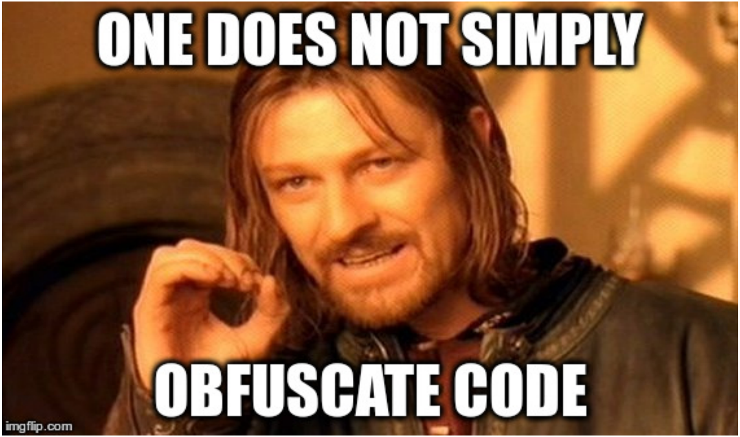
- Final symbols address unknown at IR-time
- C++ exception frames to register "by hand"
- NX-bit protection

Hard to make it work in real-life applications!

Table of Contents

Table of Contents

Choice



To the road of performances...

Problem

- Impossible to apply all the obfuscations all the time
- Locality: let the user decide which code need to be protected

$\Rightarrow$ Need a way to tell the compiler what to apply and where!

To the road of performances...

Problem

- Impossible to apply all the obfuscations all the time
- Locality: let the user decide which code need to be protected

$\Rightarrow$ Need a way to tell the compiler what to apply and where!

In LLVM

- Function attributes to give hints to some optimisation passes
- (De)activation of optimisations with flags / optimization level
- But the compilation flow is "statically" written in the pass manager builder

$\Rightarrow$ No way to let the user specify the compilation flow!

Pragma-based compilation flow

Use pragma on code blocks/functions

```
#pragma global run_pass CallGraphFlat()

void foo(int) { ... }

#pragma run_pass OpaquePredicates(ratio=.9)
#pragma run_pass OpaqueZero()
int func_to_protect(int i)
{
  foo(0);
  return i*5;
}
```

Runs `OpaqueZero` and `OpaquePredicates` on `func_to_protect`, then `CallGraphFlat` on the whole module.

Architecture

Pragma-based compilation flow

Use pragma on code blocks/functions

```
schedule:
 - !function
   name: func_to_protect
   passes:
    - OpaqueZero()
    - OpaquePredicates(ratio=.9)
 - !module
   passes:
    - CallGraphFlat()
```

Runs `OpaqueZero` and `OpaquePredicates` on `func_to_protect`, then `CallGraphFlat` on the whole module.

Architecture

Pragma-based compilation flow

Use pragma on code blocks/functions

```
schedule:
 - !function
   name: func_to_protect
   passes:
    - OpaqueZero()
    - OpaquePredicates(ratio=.9)
 - !module
   passes:
    - CallGraphFlat()
```

Runs `OpaqueZero` and `OpaquePredicates` on `func_to_protect`, then `CallGraphFlat` on the whole module.

Architecture

```
name: OpaquePredicates
level: basic block
options:
    - name: ratio
      values: [0.,1.]
```

Pragma-based compilation flow

Use pragma on code blocks/functions

```
schedule:
 - !function
   name: func_to_protect
   passes:
    - OpaqueZero()
    - OpaquePredicates(ratio=.9)
 - !module
   passes:
    - CallGraphFlat()
```

Runs `OpaqueZero` and `OpaquePredicates` on `func_to_protect`, then `CallGraphFlat` on the whole module.

Architecture

- Custom pass factory to instantiate pass (and options) at runtime
- Creates and run classical `llvm::Pass`

Table of Contents

Pass ordering

Compilation flow (simplified)

- LLVM optimisations passes
- **One** LLVM pass that schedules **obfuscations**
- **One** LLVM pass that schedules **post-optimize**

$\Rightarrow$ We run pass managers within an LLVM pass!

Pass ordering

Compilation flow (simplified)

- LLVM optimisations passes
- **One** LLVM pass that schedules **obfuscations**
- **One** LLVM pass that schedules **post-optimize**

$\Rightarrow$ We run pass managers within an LLVM pass!

Is that really a good idea?

- Some optimisations rely on target dependant information (i.e.: `SimplifyLibCalls`)
- Where do they come from?

Pass manager inception: gotcha

Listing 1: clang/lib/CodeGen/BackendUtil.cpp:CreatePasses

```
Triple TargetTriple(TheModule->getTargetTriple());
std::unique_ptr<TargetLibraryInfoImpl> TLII(
    createTLII(TargetTriple, CodeGenOpts));
// [...]
  MPM.add(new TargetLibraryInfoWrapperPass(*TLII));
// [...]
  FPM.add(new TargetLibraryInfoWrapperPass(*TLII));
// [...]
  PMBuilder.populateFunctionPassManager(FPM);
  PMBuilder.populateModulePassManager(MPM);
```

Gotcha

- `TargetLibraryInfo` and `TargetTransformInfo` analyzes must be forwarded to the new pass managers
- APIs of these analyzes isn't meant for this...

Table of Contents


BUGS
BUGS EVERYWHERE
imgflip.com

Unit Testing

Randomness

To seed or not to seed?

Looking for invariant

`FileCheck` 4TW

Non-reversability

`-O2` as a minimal contract

Z3/Arybo to proove obfuscated substitutions

Fuzz Testing

CSmith

Bug 2545

```
long  aa = var_10 * long(1945964878U * var_41 >> var_1 );
int a = var_1 & aa;
unsigned u = (unsigned(aa) - aa) || !a;
```

Piping Obfuscations

```
[] fuzz(auto bitcode) {
    while(true) {
        auto obfuscation = get_random_obfuscation();
        obfuscation.run(bitcode);
    }
}
```

Testing in the wild

OSS validation

Each obfuscation, Maximum Level

- Lua (C)
- CMake (C++)
- OpenSSL (C)
- ZLib (C)
- libjpeg (C)
- petanque (C++) [2]

Esod Mumixam!

[2] From https://github.com/quarkslab/arybo

Pathological Testing

A.k.a. troublesome patterns

- Large object passed by value
- Weak Linkage
- Call in catch block
- Exceptions that traverse the call stack
- Variadic arguments
- Recursive calls

Finding the Origin

Common Sense

- Dump the seed
- Reproducible builds
- Save faulty builds
- Fix the seed?

Finding the origin

- **Tedious** Dichotomy on obfuscated location, from compilation unit to basic block
- **Brain Damaging CLI** Bugpoint

Meeting the limits

setjmp, vfork

Thank you Glibc

Bug 20382 - getcontext and setjmp should have `__attribute__((returns_twice))`

Meeting the limits

setjmp, vfork

Thank you Glibc

Bug 20382 - getcontext and setjmp should have `__attribute__((returns_twice))`

Generating Large Expression

Listing 3: ScheduleDAGRRList.cpp

```
assert(PredSU->NumSuccsLeft < UINT_MAX && "NumSuccsLeft will
    overflow!");
```

- Being stuck in register allocator for ages
- Hitting Valgrind max instruction per function

Ultimate Testing

Working at QuarksLab

Isn't that a security firm?

Ultimate Testing

Working at QuarksLab

Isn't that a security firm?

Speak with the reversers!

- Watch them work
- Read their report
- Internal Capture The Flag Challenges

Questions?

Introduction

LLVM bitcode obfuscations

Frankenstein obfuscation

Improved pass management

Tests and benchmarks

SECURING EVERY BIT OF YOUR DATA