Code Obfuscation with LLVM

Matus Mrekaj

24 May 2023

Surreptitious software and why is it useful. Concentrating on implementing various transformation passes in LLVM for code obfuscation making static analysis of a binary harder but not unfeasible.

1 Computer Security

When we talk about computer security, we most often talk about how to secure our data and communications from being read by a malicious actor. This could be achieved by having a secret key shared between two parties that the third party does not know, this branch of security is known as cryptography.

Computer security is a vast topic in itself and has many branches that focus on different areas. While cryptography addresses the problem of secure communication and encryption of data, it is based on the fundamental assumption that the secret key is securely stored.

Consider this scenario, you have a software product packaged in a binary and you decide to distribute it to your customers who paid for it. These binaries may contain secrets that you don't want anyone else to have except the customer who paid for your product. These secrets may be keys, unique identification numbers or anything that you consider confidential. These secrets that are distributed with the binary are then used when communicating with you.

What happens when a 3rd party gets hold of one or more of the binaries and decide to re-distribute them, perhaps with a cheaper price, some changes to the features etc. The malicious actor can do whatever he wants as there is no limit as to what can and can't be done when he gets hold of the binary. This is where Surreptitious software can be useful.

1.1 Surreptitious software

Surreptitious software is a term that describes a branch of computer security that works with techniques from many branches of computer security, namely cryptography, watermarking, reverse engineering, steganography and compilers [1].

You may be wondering how we could have avoided the scenario just described using these techniques. Watermarking could be used to track illegal copies, tamperproofing could be used to prevent tampering with the binary, obfuscation could be used to protect algorithms.

As with cryptography, where the secret keys used have different lifetimes and will not prevent a third party from reading the data forever, Surreptitious Software does not guarantee that these techniques will work forever or be unbroken. The goal is to slow down the malicious actor so that breaking these algorithms takes more time than they are willing to invest and potentially give up [1].

You may be asking yourself why would you want to store secrets in your binary that could be exploited. There is no judgement to this question, whatever the case is its good to know what can be used to improve the security when such a use cases arises.

1.1.1 Examples of Surreptitious software

A quick glance at the following website https://patents.google.com/ with keywords, watermarking, obfuscation, tamperproofing etc... will tell us which companies use Surreptitious software. For example, Microsoft Inc. owns several patents

related to obfuscation, watermarking [10], Apple Inc. owns patents on obfuscation [3], and there are many more such examples.

These are only those that are public publicly accessbile, The use-cases for Surreptitious software can range from simply preventing Games from being cracked to protecting Military equipment [1].

2 Code Obfuscation

In this paper, the focus was only on part of Surreptitious software for making static analysis of a program harder, namely code obfuscation. Several obfuscation methods were implemented as LLVM [1] transformation passes.

The obfuscation passes work on the IR level in LLVM, this allows for the transformations to be language independent. Any language that has an implementation for the LLVM frontend namely Ada, C, C++, D, Delphi, Fortran, Haskell, Julia, Objective-C, Rust, and Swift should theoretically work with the transformation passes, it may be the case that some of these languages use LLVM constructs which weren't considered when implementing the passes and thus may break. The passes were tested with C, C++.

Further, we also achieve platform independence as the modified IR code can then be passed to the LLVM backends, such as x86, PowerPC, ARM, and SPARC, to then generate code for the given type of CPU [9].

2.1 Related work

There are many commercial available code obfuscators, such as Tigress, that support various languages. From the open-source scene the more known one is *ollvm* which is described in [11], Since it ended with LLVM version 4.0 other projects started, [6, 7, 12] that were build upon the codebase and added new transformation passes.

[1] LLVM is a set of compiler and toolchain technologies `https://llvm.org/`

2.2 My Contribution

The aim was to implement common obfuscation methods, with some modifications, which are described in [2]. The LLVM passes were implemented for LLVM v15.0.7, which was the stable version at the time of start of the implementation.

The following methods were implemented

1. Basic Block Extraction
2. Operation Substitution
3. Strings Obfuscation
4. Opaque Predicates
5. Function Merging
6. Introduce Jump into Loop
7. Control Flow Flattening
8. Integer Literals Obfuscation
9. Function Call Obfuscation
10. Branch Function Obfuscation
11. Bogus Control Flow

3 Transformation Passes

3.1 Basic Block Extraction

Basic Block extraction is a very simple transformation pass that given a basic block in a function it extracts that basic block into a new function, a call instruction to the new function is then inserted in-place of the basic block. The changes are illustrated by Figure 1 on the top of the following page.

By itself the block extraction doesn't add much confusion to the control flow graph, if however combined with other transformation passes, which further obfuscate the newly created function for the extracted block, it can result in strong control flow obfuscation.

3.2 Operation Substitution

Operation substitution is another simple transformation pass that substitutes an instruction into a sequence of instructions which ultimately end lup with the same result.

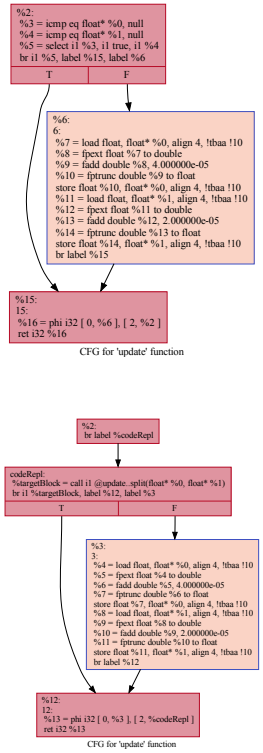


Figure 1: Block Extraction Transformation Pass

This method is also known as Mixed Boolean-Arithmetic expressions. The MBA expressions were collected from [5, 11] with some variations.

The used substitutions are listed below where A is the result B, C are the operators of the expresion and R is a random value.

AND Substitution $a = b \wedge c$
$a = (b \oplus \neg c) \wedge b$
$a = \neg(\neg b \vee \neg c) \wedge (r \vee \neg r)$
$a = (\neg b \vee c) - (\neg b)$
XOR Substitution $a = b \oplus c$
$a = (b \oplus r) \oplus (c \oplus r)$
$a = (\neg b \wedge c) \vee (b \wedge \neg c)$
$a = (b \vee c) - (b \wedge c)$
$a = (\neg b \wedge r \vee b \wedge \neg r) \oplus (\neg c \wedge r \vee c \wedge \neg r)$

OR Substitution $a = b \vee c$
$a = (b \wedge c) \vee (b \oplus c)$
$a = (b \wedge \neg c) + c$
$a = [(\neg b \wedge r) \vee (b \wedge \neg r) \oplus (\neg c \wedge r) \vee (c \wedge \neg r)] \vee$ $[\neg(\neg b \vee \neg c) \wedge (r \vee \neg r)]$
Sub Substitution $a = b - c$
$a = b + (-c)$
$a = b + r;\ a = a - c;\ a = a - r$
$a = b - r;\ a = a - c;\ a = a + r$
Add Substitution $a = b + c$
$a = b - (-c)$
$a = -(-b + (-c))$
$a = b + r;\ a = a + c;\ a = a - r$
$a = b - r;\ a = a + c;\ a = a + r$
$a = (b \wedge c) + 2 * (b \oplus c)$
$a = (b \wedge c) + (b \vee c)$

In addition to the *Add substitution* there is also a special substitution for 8-bit integers.

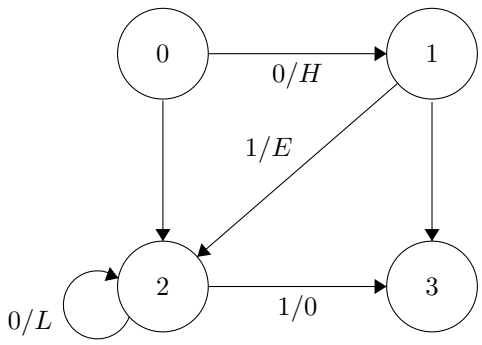
$$\text{a} = (((b \oplus c) + 2 * (b \wedge c)) * 39 + 23) * 151 + 111$$

3.3 String Obfuscation

When obfuscating strings, you can't avoid reconstructing the original string because at some point you will have to work with it, for example before comparing it to other strings, printing it via print, etc. Thus, string obfuscation plays an important role in static analysis, but it can be broken by watching the execution of the program until the transformation is reversed and working with the original string.

This transformation pass is implemented by making use of Mealy machines as described in Chapter 4.5.3 of [2]

Consider the string *Hello* for which the following state machine was build.



In the binary it would be replace with the sequence 01001, when decoding the sequence into the original string we need some information to reconstruct the string from the state machine, for this we use two arrays.

```
// transitions to nodes
int next[][2] = {
    {1, 2},
    {3, 2},
    {2, 3}
};
// printable chars
char out [][2] = {
    {'H', 'q'},
    {'t', 'E'},
    {'L', 'O'}
};
```

We always start from node 0, for the transiftion from that node we look at the outgoing edges and take the next bit from the generated sequence which is also 0. $next[0][0]$ this yields the next node which is 1 and the character $out[0][0]$ which is 'H'. This is repeated until the bit sequence is consumed.

3.4 Opaque Predicates

Most of the implemented transformation passes in this paper rely on opaque predicates. As described in Chapter 4.3.1 in [2] opaque predicates are simply expressions that are either opaquely true or opaquely false.

The predicates used as the condition shouldn't be to easy to break. For example this opaquely true

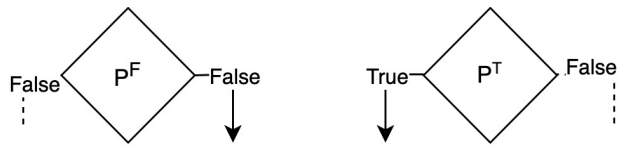


Figure 2: Opaque Predicates

predicate $x \oplus x == 0$ is easy to break and most decompilers (and also optimazation passes) wouldn't have a problem deducing the result.

Thus in the transformation pass when inserting predicates we decided to use more numerical expressions which work $\forall x \in \mathbb{Z}$

Opaquely True predicates
$(x \wedge 1 == 0) \vee [3 * (x^2 + x)] \bmod 2 == 0$
$(x \wedge 1 == 1) \vee (x^2 + x) \bmod 2 == 0$
$(2x + 2) * (2x) \bmod 4 == 0 \vee (x^2 + x) \bmod 2 == 0$
$3 * (x^2 + x) \bmod 2 == 0 \wedge (x^2 + x) \bmod 2 == 0$
$(x^2 + x) \bmod 2 == 0 \wedge (2x + 2)(2x) \bmod 4 == 0$
$(x + x^3) \bmod 2 == 0 \wedge (2x + 2)(2x) \bmod 4 == 0$

These predicates are then used with the combination of the substitution pass to obfuscate the Loop conditions and add bogus control flow by splitting a basic block at a random instruction as depicted in Figure 3.

The transformation pass chooses at random whether the new block has random instructions that don't affect the program or uses the predicate $x \bmod 2 == 0$ to clone the block such that both block gets executed at random and in one of them obfuscate the

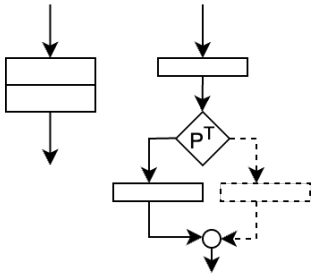


Figure 3: Opaque Predicates Bogus Flow

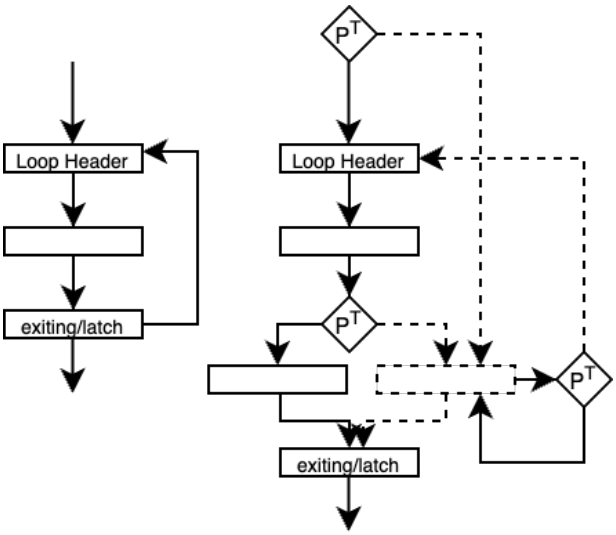


Figure 4: Jump To Loop

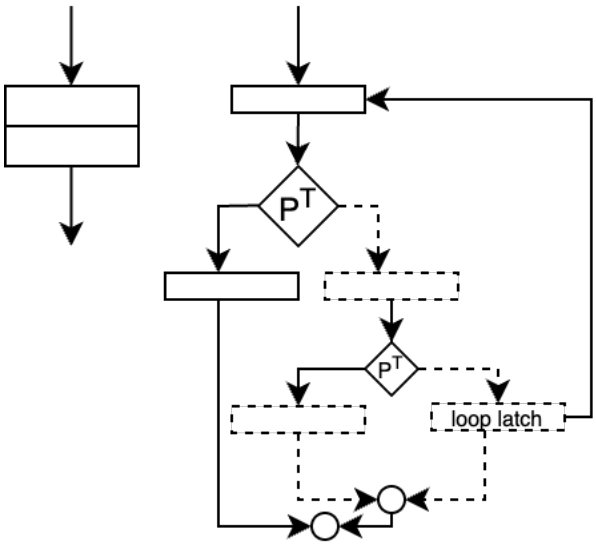


Figure 5: Bogus Control Flow

instructions by either adding random instructions or using the substitution pass.

This transformation pass will only transform basic blocks for which a reachable integer [2] exists.

3.5 Jump to Loop

Jump to Loop further builds upon the opaque predicates by introducing a jump into a randomly chosen basic block that is part of a Top-Level [3] Loop without nested loops and introduces a new nested loop.

The jumped to basic block is split at a random instruction and the newly inserted basic block filled with bogus instructions. The transformation is depicted by Figure 4.

3.6 Bogus Control Flow

Another transformation pass that builds further upon opaque predicates. Given a basic block it first applies the transformation described in Figure 3 it then chooses a random block from the two split blocks continues to add bogus operations into it and proceeds to performer the same split transformation again on that chosen block

One of the latter split blocks will be chosen as the loop latch and a branch to the former split basic block will be added, forming a loop from the created transformations, this process is depicted in Figure 5.

3.7 Function Merging

Generally, programmers write functions to abstract away logic of a program into a single unit that can be looked at independently from other logic of that program, making it easier to understand and read the code.

Function merging, as described in Chapter 4.6.1 of [2], is the exact opposite of that, we merge functions into a single function and replace all occurrences of the merged function from the program and replaced them with the new function, this is known as abstraction breaking.

To illustrate the transformation pass consider this program.

[2] A reachable integer is an integer variable that was defined up until the point when control flow reaches the basic block.

[3] Top Level Loop is the Root of the Tree of Loops nested inside it. `https://llvm.org/docs/LoopTerminology.html#important-notes`

```
string rand_string() {
    return to_string(rand());
}

int add(int first, int second) {
    return first + second;
}

int main() {
    int result = add_two(1, 2);
    cout << result << '\n';

    string str = rand_string();
    cout << str << '\n';
}
```

After applying the transformation pass the program will look as follows

```
void merged(int case, string *res,
int first, int second, int* res2) {
    switch (case) {
     case 0:
         *res = to_string(rand());
     case 1:
         *res2 = first + second;
    }
}

int main() {
    int result;
    merged(1, NULL, 1, 2, &result);
    cout << result << '\n';

    string str;
    merged(0, &str, 0, 0, NULL);
    cout << str << '\n';
}
```

3.8 Integer Literals Obfuscation

As with string literals, integer literals occur very frequently in a binary and information can be deduced by understanding in what context that given integer literal is being used. Obfuscating integer literals is a bit harder as we are working with compile-time constants and thus further optimization passes after our transformation pass could "optimize" the obfuscation away.

In [12] I've stumbled upon an idea, that the constant obfuscation pass in that project is using, which I've liked and thus decided to base my solution on it as well.

The idea is that we replace the integer constant with a multiplication of two carefully chosen values. The first is a randomly chosen odd integer and the second is the modular inverse of that odd integers multiplied with the constant we want to obfuscate.

Finding the modular inverse of an odd number can be achieved with this function, explanations can be found in [4, 8]

```
uint64_t mod_inv(uint64_t x) {
    uint64_t y = (3 * x) ^ 2;
    y = y * (2 - y * x);
    y = y * (2 - y * x);
    y = y * (2 - y * x);
    y = y * (2 - y * x);
    return y;
}
```

The multiplication $odd * (mod\_inv(odd) * constant)$ will then at runtime yield the original constant. The operands of the multiplication have been further obfuscated by XORing them with random operations that cancel out and yield the original operand.

The obfuscation of a constant after this transformation pass have the following structure

$(A \oplus ... \oplus odd \oplus ...) * (B \oplus ... \oplus mod\_inv(odd) * constant \oplus ...)$

3.9 Branch Function Obfuscation

The idea in Chapter 4.3.5 in [2], was that unconditional branch instructions would be replaced by a call to a "branch function" that would overwrite the return address on the stack by computing a new return address from values stored in an array.

The implementation of this would be a bit tricky at the IR level in LLVM, as the assembly instructions may not be compatible for every platform, I've thus opted out and modified the implementation.

For every function an array will be created where the addresses of each basic block within that function

will be stored. Then, two "branch functions" will be generated that calculate the return address based on the provided arg.

```
int8* bf(int32 *n) {
  int64 idx = map((int64)(*n));
  return &T[idx];
}

int64 map(int64 n) {
  // R is a random generated number
  return R ^ n;
}
```

At the start of the function, the created arrays will be populated by the addresses of the basic blocks within that function.

```
// This is done for every
// Basic Block in that function.
K = R ^ idx;
T[map(K)] = &BlockAddress;
```

The value of K, will be generated by the transformation pass and will be a constant.

Then the replacement of branch instructions is done with an indirect branch instruction that receives the address to jump to by calling the $bf()$ function. The argument to that function depends on whether the branch instruction being replaced is conditional or not.

If it's conditional the branch instruction

```
br cond ? &BlockAddress_1:
          &BlockAddress_2;
```

Will be replaced by

```
K_1 = R ^ idx_1;
K_2 = R ^ idx_2;
N = cond ? K_1 : K_2;
br bf(N ^ (K_1 ^ K_2));
```

The XOR operations will cancel out and only the index into the array will be left.

Unconditional branch instructions are replaced with conditional branch instructions by using an opaquely true predicate. The true case continue with the original Basic block and the false case has a back reference to the block itself forming a loop. This conditional branch instruction is then modified with the same steps as above.

At a higher abstraction the changes can be imagined as

```
int f(int64 n) {
  if (n % 2 == 0) {
    return 2;
  } else if (n % 3 == 0) {
    return 3;
  }
  return 0;
}

int f(int64 n) {
  T[map(R ^ 0)] = &if-part;
  T[map(R ^ 1)] = &else-part;
  T[map(R ^ 2)] = &endif;
  T[map(R ^ 3)] = &ret;
  int result;
  goto bf(R ^ 0);
if-part:
  if (n % 2 == 0)
    result = 2;
    N = cond ? (R^3) : (R^0);
    goto bf(N^(R^0)^(R^3));
else-part:
  else if (n % 3 == 0)
    result =  3;
    N = cond ? (R^3) : (R^1);
    goto bf(N^(R^1)^(R^3));
endif:
    result = 0;
    N = cond ? (R^3) : (R^2);
    goto bf(N^(R^2)^(R^3));
ret:
    return result;
}
```

3.10 Function Call Obfuscation

Function call obfuscation works on the same principle as the branch function but instead of replacing branch instructions call instructions are replaced.

The following example illustrates the transformation pass at a higher abstraction.

```
int add(int a, int b) {
  return a + b;
}
int f(int64 n) {
  int a = add(1, 2);
  int b = add(3, 4);
  printf("%d\n", a);
  printf("%d\n", b);
  return a + b;
}

int f(int64 n) {
  T[map(R ^ 0)] = (int*)(&add);
  T[map(R ^ 1)] = (int*)(&add);
  T[map(R ^ 2)] = (int*)(&printf);
  T[map(R ^ 3)] = (int*)(&printf);

  // retype and call
  int a =
  ((int (*)(int, int)))(T[map(R ^ 0)])
  (1, 2);

  int b =
  (int (*)(int, int))(T[map(R ^ 1)])
  (3, 4);

  // retype and call
  (int (*)(const char*, ...))
  (T[map(R ^ 2])("%d\n", a)

  (int (*)(const char*, ...))
  (T[map(R ^ 3)])("%d\n", b)

  return a + b;
}
```

3.11 Control Flow Flattening

This transformation pass implements two solutions for control flow flattening. The first is based on of [11] also described in Chapter 4.3.2 of [2], using a switch to decide which block within the switch statement should be executed next.

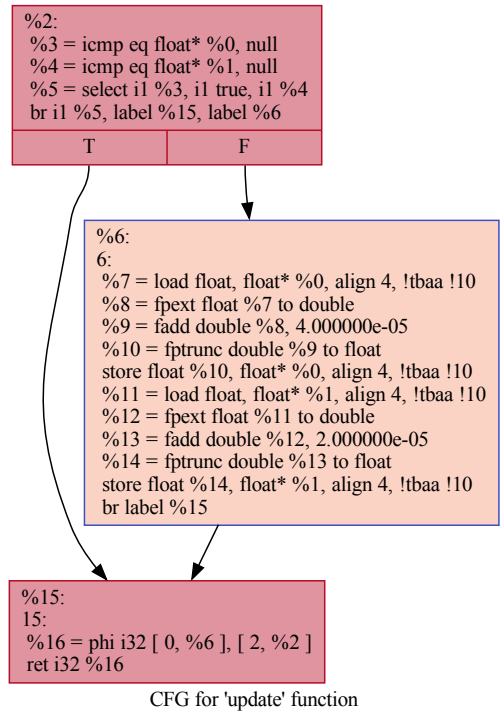


Figure 6: Original Function Flow

The second is purely based on indirect branching, using aliasing to obfuscate the control flow.

The idea of control flow flattening is to destroy the control flow of a function. Any branching, looping or jumping is flattened to make the original control flow hard to follow. The transformation pass is best illustrated by looking at the transformation in Figure 7 and Figure 8 and comparing them to the orignal flow in Figure 6.

The former version uses an lookup array from which calculations are made to get the next block to be executed, addition, subtraction and modulus is used when calculating the switch case number.

The latter version uses a lookup table where addresses are stored, it calculates the index of into that lookup table to get the address to which it should

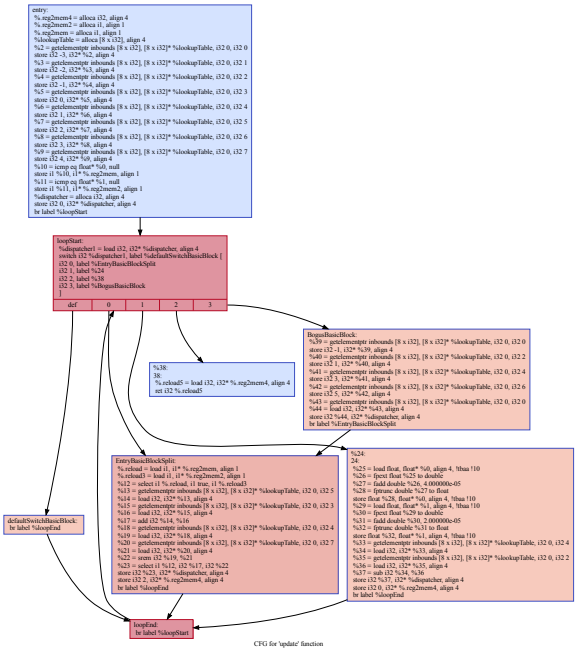


Figure 7: Control Flow Flattening Switch

jump next.

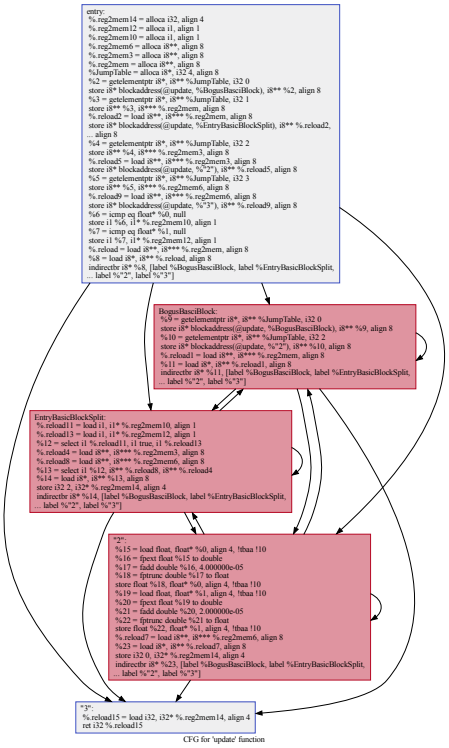


Figure 8: Control Flow Flattening Aliasing

4 Performance Comparison

All of the described transformation passes in the previous section have been applied on a simple program that renders a mesh to the terminal. The result of the benchmark are shown in Figure 9.

Overall the time it took to render a single frame was **13,13** times slower then the original program without obfuscations. The program was executed on an Apple M1 chip.

The program was run via *lli*, without any further optimization passes, which is the LLVM IR code interpreter.

The IR file was generated from compiling the C program, available under the *demo* folder in the repository with $-emit-llvm$ flag.

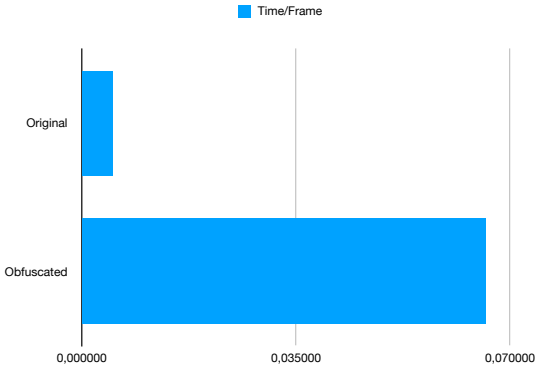


Figure 9: Benchmarks

5 Future work

The implemented transformation passes can be improved upon, for example in the call obfuscation further obfuscating the performed call instruction, or in the control flow flattening pass, calculating the which switch statement will be executed next could be done via constructing a specific array, as described in Chapter 4.4.2 [2] and calculating the value at runtime instead at compile time by inserting a function into the code.

Further transformation passes for destroying abstractions such as classes, preventing debugging or tamper-proofing can be added. The transformation passes have been implemented to the best of my knowledge of LLVM, it may happen at an unexpected construct that they may break, thus making sure the passes don't fail on unexpected LLVM constructs, that other frontend languages may use, and hardening the passes overall.

6 Conclusion

Most of the obfuscation methods described in Chapter 4 of [1] have been implemented, given that there already exist a few open-source obfuscation implementations for LLVM, that also implement some of the methods described in the book, effort was put in to distinguish the implementations in this paper from those already existing ones by implementing the method in a different manner, such as the indirect branching version of control flow flattening, or simply reworking the method to work differently but the core idea would still be there as changing it would yield a completely different transformation.

References

[1] Christian Collberg. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Addison-Wesley, 2009, pages 19–26. ISBN: 978-0321549259.

[2] Christian Collberg. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Chapter 4. Addison-Wesley, 2009, pages 297–428. ISBN: 978-0321549259.

[3] *Apple Obfuscation Patent*. URL: `https://patents.google.com/patent/US20110206285A1/en`.

[4] *Computing the inverse of odd integers*. URL: `https://lemire.me/blog/2017/09/18/computing-the-inverse-of-odd-integers/`.

[5] *Defeating MBA-based Obfuscation*. URL: `https://hal.science/hal-01388109/document`.

[6] *Hikari*. URL: `https://github.com/HikariObfuscator/Hikari`.

[7] *Hikari*. URL: `https://github.com/61bcdefg/Hikari-LLVM15`.

[8] *Integer multiplicative inverse via Newton's method*. URL: `https://marc-b-reynolds.github.io/math/2017/09/18/ModInverse.html#mjx-eqn%3Arr`.

[9] *LLVM Backend*. URL: `https://llvm.org/docs/WritingAnLLVMBackend.html#introduction`.

[10] *Microsoft Watermarking Patent*. URL: `https://patents.google.com/patent/US20110314550A1/en`.

[11] *OLLVM*. URL: `https://crypto.junod.info/spro15.pdf`.

[12] *YANSOllvm*. URL: `https://github.com/emc2314/YANSOllvm`.