Combining Obfuscation and Optimizations in the Real World

Serge Guelton
Quarkslab
13 rue Saint-Ambroise
75011 Paris
Email: sguelton@quarkslab.com

Adrien Guinet
Quarkslab
13 rue Saint-Ambroise
75011 Paris
Email: aguinet@quarkslab.com

Pierrick Brunet
Quarkslab
13 rue Saint-Ambroise
75011 Paris
Email: pbrunet@quarkslab.com

Juan Manuel Martinez
Quarkslab
13 rue Saint-Ambroise
75011 Paris
Email: jmmartinez@quarkslab.com

Fabien Dagnat, Nicolas Szlifierski
IMT Atlantique, IRISA, Université Bretagne Loire
Brest, France
Email: first.last@imt-atlantique.fr

***Abstract*—Code obfuscation is the *de facto* standard to protect intellectual property when delivering code in an unmanaged environment. It relies on additive layers of code tangling techniques, white-box encryption calls and platform-specific or tool-specific countermeasures to make it harder for a reverse engineer to access critical pieces of data or to understand core algorithms.**

The literature provides plenty of different obfuscation techniques that can be used at compile time to transform data or control flow in order to provide some kind of protection against different reverse engineering scenarii.

Scheduling code transformations to optimise a given metric is known as the *pass scheduling* problem, a problem known to be NP-hard, but solved in a practical way using hard-coded sequences that are generally satisfactory. Adding code obfuscation to the problem introduces two new dimensions. First, as a code obfuscator needs to find a balance between obfuscation and performance, *pass scheduling* becomes a multi-criteria optimisation problem. Second, obfuscation passes transform their inputs in unconventional ways, which means some pass combinations may not be desirable or even valid.

This paper highlights several issues met when blindly chaining different kind of obfuscation and optimisation passes, emphasising the need of a formal model to combine them. It proposes a non-intrusive control language as an evolution to traditional, sequential pass management techniques. The language is validated on real-world scenarii gathered during the development of an industrial-strength obfuscator on top of the LLVM compiler infrastructure.

I. Introduction

Code obfuscation is the process of transforming a program, either in source, intermediate or binary form, in order to make it difficult for a reverse engineer to access some kind of secret hidden in the program. Secrets can include specific data like cryptographic keys, copyrighted assets etc. They can also consist in proprietary algorithms, encryption methods, protocols etc.

Although perfect obfuscation has been proved impossible [1], and *indistinguishable obfuscation* [2] is far from practical efficiency [3], research and industry have continuously produced a large number of obfuscation techniques to protect against different threats. These threats range from automatic tools such as static or dynamic analyzers to highly specialized engineers called *reversers*. The main practical objective being to make it difficult to understand an obfuscated program. Difficult means costly in terms of time and engineering efforts. Surveys of common obfuscation techniques has been proposed in [4], [5]. Classical techniques include control flow graph flattening [6], mixed boolean-arithmetic expressions [7] or virtualization [8]. When trying to defeat static or dynamic analysis, a valid approach is to rely on NP-hard problems that a code analyzer is unlikely to solve efficiently [9]. Relying on architectural specificities and tool limitations is also an interesting, yet short-term, solution [10].

Each of the numerous proposed obfuscation techniques is efficient only on a small number of threats. Furthermore, they often require some preconditions on the input program and have a large impact on the size of the program, its performance, its memory and energy consumption... In order to build a generic code obfuscator that defeats multiple threats, one needs to be able to combine these obfuscations with each other, and possibly with existing code optimisations. The sequence of obfuscations must also keep the execution time and the various resource consumption of the obfuscated binary within reasonable bounds.

This paper makes the following contributions:

- Identify and illustrate pass management issues specific to obfuscation.
- Formalize function property management in the context of pass management.

It is divided in five parts: the first part examines the simple scenario of trying to protect a reference sha256 computation with a combination of Tigress [11] and O-LLVM [12], showcasing the need of a finer grain control than provided by industrial compilers. The second part investigates the possible

```
1  static
2  void sha256(unsigned char* input, size_t len)
3  {
4    unsigned char hash[SHA256HashSize];
5    int i;
6
7    SHA256Context sha256;
8    SHA256Reset(&sha256);
9    SHA256Input(&sha256, input, len);
10   SHA256Result(&sha256, hash);
11   for(i = 0; i < SHA256HashSize; i++)
12   {
13       printf("%02x", hash[i]);
14   }
15 }
```

Listing 1. Sample `sha256` API usage.

interactions between typical obfuscations and optimisations, using the example of Tigress and O-LLVM to illustrate the limitations of a naive chaining approach. The third part proposes a control language to enable correct, fine-grain optimisations and obfuscations composition, while the fourth part validates the approach through the industrial experience of building an obfuscating compiler. The last part discusses related work and concludes.

II. Case study: obfuscate a cryptographic hash algorithm

In this section, we analyse a simple scenario: a developer wants to protect a simple algorithm that contains both *magic value* (data) and a specific algorithm (code): the reference sha256 computation, as specified in RFC 6234 [13]. The goal consists in preventing static and dynamic analyses to retrieve them. To achieve that goal, and illustrate the collaboration between different obfuscation engines, two obfuscating compilers are used: Tigress[1] and O-LLVM[2].

A. Code description

Our example program, SHA256, is a classical cryptographic hash function whose reference implementation is partly given in listing 1 (the "`SHA1FinalBits`" call is omitted). It consists in three main steps, `SHA256Reset`, `SHA256Input` and `SHA256Result` (in lines 8,9 and 10) that initialize the context, perform the actual computation and dump the hash, respectively.

The core of the algorithm is located in the auxiliary function `SHA224_256ProcessMessageBlock` called by `SHA256Input`. This function uses an auxiliary constant array "`K`" used to perform substitutions. The control flow is also typical of iterated cryptographic hash algorithms processing blocks of messages, with a fixed-size loop to process a data block, and an outer loop making repetitive calls to `SHA224_256ProcessMessageBlock`. Both the substitution and the block processing are typical of this family of algorithms. A good obfuscation must be able to hide the data and code structure to a reverser.

B. Tools presentation

Industrial strength obfuscators are generally not freely available and thus cannot be studied by researchers. Fortunately, a few tools are still available. O-LLVM provides an open source code base with a limited set of obfuscations and enough material to study the general approach. The binary code of Tigress is also freely available and shipped with enough documentation to understand its pass management options.

1) O-LLVM: It is an LLVM bitcode obfuscator that provides, in its open source version, three simple obfuscations:

- *instruction substitution* transforms explicit patterns of instructions into other patterns of instructions generally more complex;
- *bogus control flow insertion* adds bogus basic blocks and uses opaque predicates to guard them and prevent their execution;
- *control flow graph flattening* replaces the control flow of a function by a lookup in a transition table.

The user can use compiler options to control various obfuscation parameters:

- the number of times a substitution is applied;
- the probability to apply bogus control flow insertion on a basic block;
- whether basic blocks should be split before applying control flow graph flattening;

The obfuscation is applied in the middle of the optimisation pipeline, only once (even if the flag is repeated). It is possible to control which functions are obfuscated or not using function attributes.

2) Tigress: It is a source-to-source C code obfuscator that provides, among others, three main high-level transformations:

- *code virtualization* compiles a function into a custom bytecode that is interpreted by a custom virtual machine;
- *jitting* generates the function machine code at runtime;
- *dynamic jitting* dynamically generates different versions of the function machine code at runtime.

It also provides more common obfuscations like control flow flattening or data encoding. Obfuscations are scheduled using a list of obfuscation and function pairs, stating which obfuscation to apply to which function.

C. Experimentation

A naive way to obfuscate the SHA256 computation would be to apply a set of obfuscations to each `SHA256*` function. Unfortunately, there is no equivalent to generic optimisation flags like `-On` for obfuscations. Without expert knowledge on each obfuscation and the way they interact with each other, one can only resort on chaining them on the targeted functions with the hope that piling up obfuscations raises the security level, as illustrated in Listing 2.

[1] http://tigress.cs.arizona.edu/

[2] https://github.com/obfuscator-llvm/obfuscator

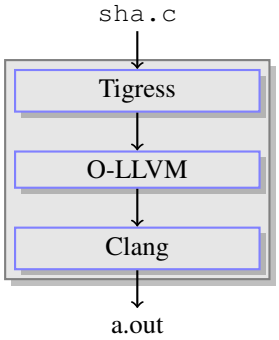

Fig. 1. Obfuscation pipeline combining Tigress, O-LLVM and clang optimisations.

```
> tigress sha.c -O2
  --Transform=Flatten \
  --Functions=SHA224_256ProcessMessageBlock \
  --Transform=Virtualize \
  --Functions=SHA224_256ProcessMessageBlock
```

Listing 2. Example of `tigress` obfuscation CLI.

Note that in this case, the optimisations triggered by the `-O2` option are applied *after* the obfuscations.

As Tigress is a source-to-source obfuscator, it is perfectly possible to combine it with the obfuscation of O-LLVM. For example, one can apply the CFG flattening of O-LLVM on the source obfuscated by Tigress, named `tigress_output.c` in Listing 3.

```
> clang tigress_output.c -O2 -mllvm -fla
```

Listing 3. Example of O-LLVM obfuscation CLI.

In that scenario, the sources are first obfuscated by Tigress, then optimised by LLVM and finally obfuscated by O-LLVM, as illustrated in Figure 1.

However, the obfuscator's user must find a trade off between the obfuscation quality and the execution speed of the obfuscated binary. In order to assess the performance regression, we studied three obfuscation schemes, namely *virtualize*, *virtualize* followed by *flattening*, and *flattening* followed by *virtualize*. The benchmark uses the `clang` compiler from O-LLVM as a back-end compiler in all cases, measuring the hash speed on a 10 MB file using the SHA256 function. Figure 2 presents the execution time of the same SHA kernel compiled with Tigress *virtualize* protection using ten different seeds (the x-value), measured by ten executions per seed (each giving a point).

The first striking result is the non-reproducibility of performances across different seeds. Indeed, obfuscators usually rely on a random engine to choose different patterns or obfuscation sub-strategies, in order to improve the diversity of the generated code. However it appears this is done at the cost of non-reproducible performance. This lack of reproducibility is a real problem when trying to find a balance between obfuscation and optimisation, as the result is highly instable.

Once the seed is fixed to an arbitrary value (1 in that case), we observe that:

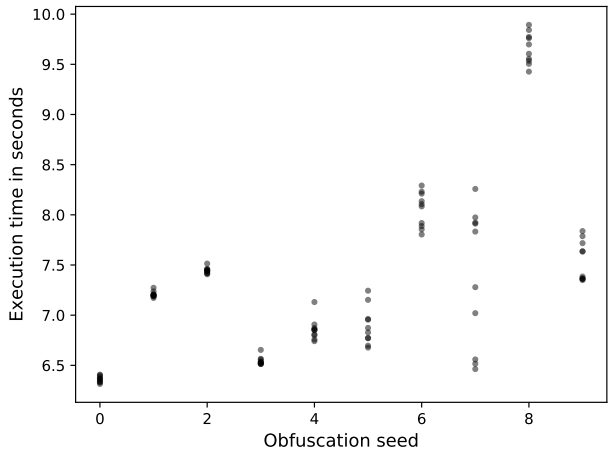

Fig. 2. Distribution of execution time of an obfuscated SHA256 kernel with different seeds.

Obfuscation Setup	Average execution time (s)
reference	0.08
virtualization	5.37
flattening	0.11
virt. + flat.	20.35
flat. + virt.	> 600

TABLE I
AVERAGE TIME TAKEN TO HASH A 10 MB FILE USING A SHA256 KERNEL OBFUSCATED USING DIFFERENT TIGRESS OBFUSCATION COMBINATIONS.

- The version obfuscated by Tigress alone, without the extra substitution, runs 80 times slower than the original version in average.
- The version obfuscated by Tigress then O-LLVM runs 83 times slower than the original one.

Table I summarizes the average execution time of the same kernel compiled with Tigress, with five different configurations: no obfuscation, virtualization, CFG flattening, virtualization then flattening, and flattening then virtualization. The results illustrate the non-predictability of the performance impact of the combination of obfuscations, even for only two of them. In terms of performance impact, the combination of two obfuscations is not commutative. Furthermore, some obfuscations may generate many candidates for a further obfuscation, implying a huge performance cost, as in the flattening followed by virtualization setup.

Additionally, from a security perspective, only a single function is obfuscated by the *virtualize* obfuscation, so the generic algorithm scheme, including the block processing, is still apparent. To hide that part, one could first *inline* the various `SHA*` calls, *e.g.* using a source-to-source compiler framework like PIPS [14].

D. Summary

This introductory experimentation raises several questions:

1) As a naive obfuscator user, choosing which obfuscation to pick is difficult. Is it possible to describe a generic obfuscation scheme for non expert users, in a similar manner to `-On` switches?
2) Applying inlining before some obfuscations increases security. Applying optimisations *before* virtualization decreases the size of the obfuscated functions, thus certainly improves its speed. What are the possible interactions between different obfuscations and code transformations?
3) What kind of considerations should be taken into account when designing an obfuscation that relies on a random engine to perform its transformation to avoid performance reproducibility issues?

III. Inspecting interactions between passes

In order to keep the execution time of the obfuscated binary within reasonable bounds, it is still necessary to optimise it using regular code optimisations. This raises the issue of the interactions between obfuscation and optimisation.

A. Optimization before obfuscation

Intuitively, one of the reasons for the important performance penalty implied by the use of the virtualizer may be caused by the fact that it handles unoptimised C code as input, and once transformed into the virtualized bytecode, there is not much optimisation opportunity available as the optimiser cannot read through the virtualized bytecode.

To assert this idea, we considered the rotation macros `SHA256_ROTL` and `SHA256i_ROTR`. Once optimised by clang, they turn into a single `ror` or `rol`. But once virtualized, they are split into several operations which can no longer be optimised. By optimising them manually to call intrinsics and *then* calling Tigress, we already get a $\times 1.16$ acceleration factor compared to the original Tigress run. Applying the same strategy to the `SHA256_SIGMA*` functions leads to a $\times 1.40$ acceleration factor.

Intuitively, well-crafted obfuscations prevent further code transformations from understanding the code and optimise it as well as they would on the original code. Some obfuscations even rely on the inability of the compiler to solve complex problems, *e.g.* alias analysis [9].

B. Optimization after obfuscation

This section compares the time taken to hash a given random $10\,\mathrm{MB}$ file with the SHA256 program, using different obfuscation and optimisation flags. The results are given in table II. The average reference execution time without obfuscation for various optimisation levels are given in the second line. The third line illustrates the behavior from the virtualize obfuscation of Tigress. It shows that obfuscating compilers focus primarily on producing code difficult to reverse rather than code that runs fast.

The effect of optimisation on the output of Tigress is significant. However, Tigress is a source-to-source obfuscator, so it does not take optimised code as input, which leaves a lot of room for further optimisation.

Optimization level	`-O0`	`-O1`	`-O2`
Execution time, no obfuscation (s)	0.96	0.44	0.45
Execution time, Tigress virtualization (s)	14.7	8.46	6.95

TABLE II
AVERAGE TIME TAKEN TO HASH A $10\,\mathrm{MB}$ FILE USING A SHA256 KERNEL, FOR DIFFERENT POST-OBFUSCATION OPTIMISATION LEVELS.

pre \ post	`-O0`	`-O1`	`-O2`
`-O0`	4.44	1.70	1.32
`-O1`	2.46	1.05	1.08
`-O2`	2.92	0.99	1.02

TABLE III
AVERAGE TIME IN SECONDS TAKEN TO HASH A $50\,\mathrm{MB}$ FILE USING A SHA256 KERNEL OBFUSCATED USING O-LLVM CONTROL FLOW GRAPH FLATTENING, FOR DIFFERENT PRE AND POST OPTIMISATION LEVELS.

We led a similar experiment using the control flow graph flattening feature of O-LLVM. In this case, two optimisation phases happen: one *before* and one *after* obfuscation. Table III reports the results of hashing a $50\,\mathrm{MB}$ file while varying the levels of the pre-obfuscation and post-obfuscation optimisations.

The fastest obfuscated codes are generated when the *pre* and *post* optimisation levels are above zero. It seems always valid to optimise after obfuscation, at least for that obfuscation. Interestingly, optimising at the `-O2` level before obfuscation and `-O0` after yields slower code than optimising at the `-O1` level before obfuscation and `-O0` after. This is due to a more complex control-flow graph generated by `-O2` to handle vectorization. This further illustrates the difficult task of choosing a sequence of obfuscations and optimisations.

C. Optimizations and a de-obfuscation Tool

Let us now consider the substitution obfuscation as implemented by O-LLVM. The effect of an example of such an obfuscation is illustrated on Figure 3 where $a + b$ is replaced by $b - (-a)$.

Unsurprisingly, applying the optimisations implied by `-O2` on this output completely reverts the obfuscation. However, the *bogus control flow* obfuscation introduces an opaque predicate $(x-1) \times x \mod 2 = 0$ which is always true but that LLVM cannot simplify as the x variable is declared as a global variable, as showcased in Listing 4.

Given that several reversing tools now use compiler optimisation as a basic de-obfuscation engine [15], it seems reasonable to ensure that every obfuscation should resist basic optimisations, or be followed by an obfuscation that itself resists optimisations and hardens the previous obfuscation. For instance, the opaque predicate used by *bogus control flow* could be combined with the substitution to turn the `add`

```
define i32 @add(i32 %a, i32 %b) {
entry:
  %add = add nsw i32 %b, %a
  ret i32 %add
}

define i32 @add(i32 %a, i32 %b) {
entry:
  %0 = sub i32 0, %a
  %1 = sub i32 %b, %0
  ret i32 %1
}
```

Fig. 3. Result of a substitution applied by O-LLVM on a simple LLVM function.

```
@x = common local_unnamed_addr global i32 0

define i32 @foo(i32 %a) {
entry:
  %0 = load i32, i32* @x
  %1 = add i32 %0, -1
  %2 = mul i32 %1, %0
  %3 = and i32 %2, 1
  %4 = icmp eq i32 %3, 0
  br i1 %4, label %true_br, label %false_br
```

Listing 4. Example of an opaque predicate generated by O-LLVM.

function into the one from Listing 5. Which is not simplified back to the original code by the optimisations from `-O2`.

D. Optimization and randomness

The example studied in Section II shows that different seed parameters may have an influence on the execution time of the obfuscated program. Indeed, depending on the choice made based on the random seed, obfuscated code can have different performance behavior. Let us consider the two functions from Listing 6. Both always return zero, but `slow_zero` involves a division, so it is likely to be much slower than `fast_zero`. Depending on the *hotness* of the code section where these opaque predicates are used, randomly choosing one over the other introduces great performance variability.

The interaction with further optimisations can also introduce non-reproducible performance. Suppose one of the functions above is used to hide the zero in Figure 4, i.e. the second argument of the call to `bar` inside the loop on line 4.

```
define i32 @add(i32 %a, i32 %b) {
entry:
  %0 = sub i32 %a, 1
  %1 = mul i32 %0, %a
  %2 = urem i32 %1, 2
  %3 = sub i32 %2, %a
  %4 = sub i32 %b, %3
  ret i32 %4
}
```

Listing 5. Result of substitution combined with an opaque predicate applied by O-LLVM on a simple LLVM function.

```
uint32_t slow_zero(uint32_t x) {
    uint32_t y = x & 0xFFFFFFE;
    return y / (y + 1);
}

uint32_t fast_zero(uint32_t x) {
    return (x * 3) == (x | 1);
}
```

Listing 6. Two possible opaque predicates functions.

Depending on the free variable from the context that is chosen as an argument for the call to `zero`, the code of this line may be transformed into one of the codes on the right part of the figure. The first choice, `x`, does not depend on the loop so the optimiser is going to move the call to `zero` outside the loop. The second choice, `i`, depends on the loop and the call cannot be moved outside the loop. The performance of the final code is therefore strongly affected by the random choices (here, the variable and the zero function).

E. Chaining obfuscations

An obfuscation is typically crafted to protect against one or different threats. For instance, control flow graph flattening is a good technique against static analysis, but it falls short in front of dynamic analysis [16]. Mixed Boolean-Arithmetic expressions provides a good protection against some dynamic analysis tools [17]. Thus combining the two techniques leads to a better protection. In a similar manner, an anti-debug technique [18] would benefit from further obfuscation to hide or diversify a pattern that would be too obvious to an experienced reverser otherwise.

From a defender point-of-view, the concept can be summarized as follows. As reverse engineers use a large variety of tools: static analyzer, dynamic instrumentation, virtual machines, debuggers, etc. A complete binary protection must combine together means of defeating each approach.

From a software engineering point of view, decoupling obfuscations to be able to chain them also holds good properties. For instance the bogus control flow obfuscation used in O-LLVM actually involves two obfuscations: the first inserts random false branches in the code, and the second turns the false predicates into opaque predicates. The latter could be used in other obfuscations, *e.g.* in the substitution obfuscation from O-LLVM.

IV. Enforcing pre and post conditions

Let us consider the case of two different obfuscation techniques:

- *random code injection* is a code obfuscation that inserts random assembly guarded by an opaque predicate into existing control flow. It increases the amount of code to handle during static disassembling and may break some disassemblers.
- *breakpoint detection* scans loaded executable for particular opcodes used to implement software breakpoint

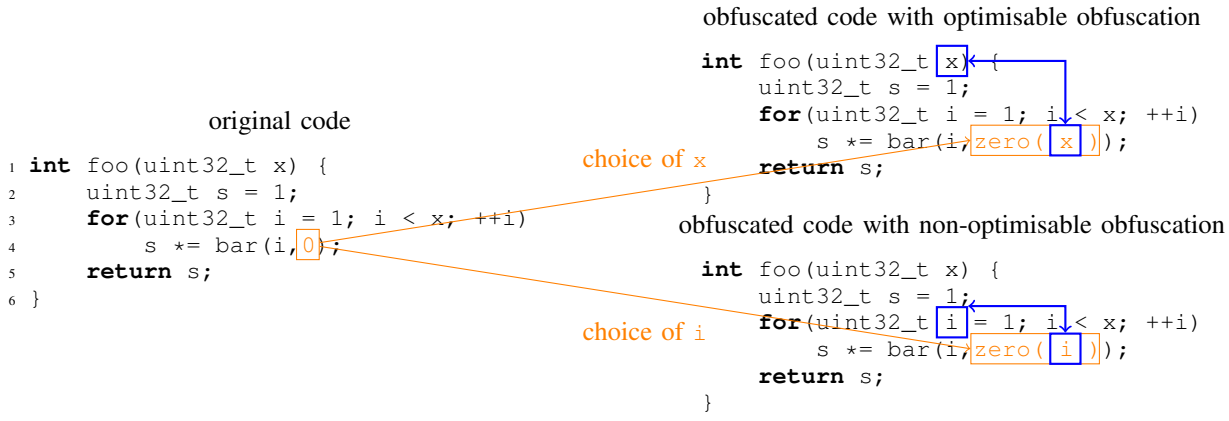

Fig. 4. Obfuscation vs optimisation

(typically `int3` on x86 architecture). It protects against debugging.

Combining the two obfuscations leads to a situation where a breakpoint opcode may be generated by *random code injection* and then detected by *breakpoint detection*, leading to the incorrect runtime statement that the code is being debugged.

Similar situations appear when using signal handlers to detect the presence of a debugger like `gdb`. Indeed there could be a signal handler already installed by the original application for its own use. There could also be a signal handler installed by an obfuscation to perform signal based control flow obfuscation. Both situations would conflict with the addition of new signal handlers to detect debuggers. A static analysis can detect the first situation (the legitimate use of a signal handler). But the second situation of conflicts should be prevented by the obfuscation tool. It should prevent the simultaneous use of the signal handlers detection and of an obfuscation relying on signal handlers. This section explores the use of a simple formalism to enforce the following properties:

1) The code is indeed obfuscated.
2) Invalid code is not generated.
3) Generated code meets performance expectations.

A. Property-based compilation of obfuscated programs

Let us model a source code as a set of functions $F \in \mathcal{P}(\mathbb{F})$ and consider the set of properties $\mathbb{P}$. A function $f \in \mathbb{F}$ can be associated to a set of properties $P_f \in \mathcal{P}(\mathbb{P})$ to form a qualified function $f' = (f, P_f) \in \mathbb{F}'$. In a similar manner, a qualified code is a code where all functions are qualified.

A code transformation $t \in \mathbb{T}$ is a function from $\mathcal{P}(\mathbb{F})$ to $\mathcal{P}(\mathbb{F})$, that produces a set of functions by transforming an input set of functions.

An obfuscation is modeled as a code transformation parameterized by a random engine $e \in \mathbb{E}$. It is a function transforming a source code and random engine into another source code and random engine.

Any code transformation (*e.g.* an optimisation) that does not use any random engine still fits in the model by returning the random engine it received. Making the random engine explicit is important to capture the randomness of an obfuscation, and potentially modify the distribution of the random engine to achieve reproducibility.

Finally, the set of code transformations is:

$$\mathbb{T} = (\mathcal{P}(\mathbb{F}) \times \mathbb{E}) \to (\mathcal{P}(\mathbb{F}) \times \mathbb{E}) \tag{1}$$

Any code transformation t from $\mathbb{T}$ is associated with a qualified code transformation t' from $\mathbb{T}'$ which takes into consideration the properties associated to each function, where:

$$\mathbb{T}' = (\mathcal{P}(\mathbb{F}') \times \mathbb{E}) \to (\mathcal{P}(\mathbb{F}') \times \mathbb{E}) \tag{2}$$

To define a qualified code transformation, we are going to use a helper function. Such a function builds a qualified code transformation from a code transformation, a function to update the properties and a predicate over the properties. The helper function $h(t, p, \phi)$ applies the transformation t on every function f whose properties P_f satisfy ϕ. When transforming a function, its properties are updated using the function p:

$$h : (\mathbb{T} \times (\mathcal{P}(\mathbb{P}) \to \mathcal{P}(\mathbb{P})) \times (\mathcal{P}(\mathbb{P}) \to \{0,1\})) \to \mathbb{T}'$$

$$(t, p, \phi) \mapsto \left((f, e), P_f \mapsto \begin{cases} t(f,e), p(P_f) & \text{if } \phi(P_f) \\ (f,e), P_f, & \text{otherwise} \end{cases} \right)$$

We propose to split the obfuscation process into three phases:

1) Initialization: turn the developer code into a qualified code. This can be done by associating a set of properties to each function, through manual annotations or based on the result of some static analysis of the program.
2) Transformation: apply qualified transformations to the qualified program
3) Postlude: verify an assertion over the qualified functions of the transformed program to assert some properties of the obfuscated program.

Note that this process only relies on qualified code transformations, so it can also be used to enhance a regular optimisation process.

B. Obfuscation flow example

Some compilers already have a mechanism similar to property management through function attributes. For instance both GCC and Clang make it possible to annotate existing code to control inlining through `noinline` and `always_inline`, which respectively prevents and forces inlining of the considered functions. GCC does print a warning if both attributes are used on the same function, but Clang does not, which shows the limits of an *ad hoc* approach to property management and motivates a formal approach.

The model proposed in Section IV-A can be used to formalize such behaviors, and makes it easy to propose solutions to the problems envisioned in Sections II and III.

1) Reducing compilation time: Let's consider the following properties:

$$\mathbb{P} = \{\texttt{to\_obf}, \texttt{opt}\}$$

Where `to_obf` means that obfuscation is required, and `opt` means that optimisation has been applied.

Suppose that we have two code transformations o_{opt} and o_{obf} that respectively optimise a function and obfuscate it. We can define the corresponding qualified code transformations by their helper functions:

$$o'_{\mathsf{opt}} = h\begin{pmatrix} o_{\mathsf{opt}}, \\ P_f \mapsto P_f \cup \{\texttt{opt}\}, \\ P_f \mapsto \texttt{opt} \notin P_f \end{pmatrix}$$

$$o'_{\mathsf{obf}} = h\begin{pmatrix} o_{\mathsf{obf}}, \\ P_f \mapsto P_f \setminus \{\texttt{to\_obf}, \texttt{opt}\}, \\ P_f \mapsto \texttt{to\_obf} \in P_f \end{pmatrix}$$

Then we combine these two transformations in the following order:

$$o'((f,e),P_f) = o'_{\mathsf{opt}}(o'_{\mathsf{obf}}(o'_{\mathsf{opt}}((f,e),P_f)))$$

The behavior of this code transformation can be described by the table of figure 5. Functions that are not optimised and should not be obfuscated are optimised only once (first row). Functions already optimised are left unmodified (second row). All functions marked to be obfuscated are obfuscated and are optimised after obfuscation (third and fourth row). This pattern avoids unnecessary re-optimisation of functions that have not been touched by obfuscation while it allows to optimise before and after obfuscation.

2) Making obfuscated hot functions performance more reproducible: As seen in Section II-C, obfuscation can have a large impact on performance, and this impact may vary importantly when changing the obfuscation seed. Therefore, to mitigate this problem, it is possible to use a fixed seed on functions that have a large impact on performance (the so-called *hot* functions).

For this, let us consider the property `hot` and the following qualified obfuscation for the obfuscation o:

$$o'_{\mathsf{hot}} = h\begin{pmatrix} f, \_ \mapsto o(f,0), \\ P_f \mapsto P_f, \\ P_f \mapsto \texttt{hot} \in P_f \end{pmatrix}$$

$$o'_{\mathsf{cold}} = h\begin{pmatrix} o, \\ P_f \mapsto P_f, \\ P_f \mapsto \texttt{hot} \notin P_f \end{pmatrix}$$

$$o'((f,e),P_f) = o'_{\mathsf{hot}}(o'_{\mathsf{cold}}((f,e),P_f))$$

The combined qualified obfuscation o' leads to the obfuscation of every function (because either a function has the property `hot` or not). Functions not marked `hot` are obfuscated normally, possibly using the random generator state. However, in functions marked as `hot` the random generator used is fixed by a constant generator, in order to get reproducible results, independent from the previous random state.

This annotation `hot` is an example of qualification that could come from other sources than user annotation, typically from profiling information.

3) Enforcing pre- and post-conditions: Let us consider the following properties `has_bp` and `forbid_bp`, which respectively state that the function code does contain a breakpoint, and that the function must not contain a breakpoint. In most cases, all functions start without these two properties.

Now consider the two obfuscations described in Section IV and their qualified versions.

Random code injection, denoted $o_{\mathsf{rand\_code}}$:

$$o'_{\text{rand\_code}} = h\begin{pmatrix} o_{\mathsf{rand\_code}}, \\ P_f \to P_f \cup \{\texttt{has\_bp}\}, \\ P_f \to \texttt{forbid\_bp} \notin P_f \end{pmatrix}$$

initial		after first o'_{opt}		after o'_{obf}		final	
function	properties	function	properties	function	properties	function	properties
f	{}	$o_{\text{opt}}(f)$	{opt}	$o_{\text{opt}}(f)$	{opt}	$o_{\text{opt}}(f)$	{opt}
f	{opt}	f	{opt}	f	{opt}	f	{opt}
f	{to_obf}	$o_{\text{opt}}(f)$	{to_obf, opt}	$o_{\text{obf}}(o_{\text{opt}}(f))$	{}	$o_{\text{opt}}(o_{\text{obf}}(o_{\text{opt}}(f)))$	{opt}
f	{opt, to_obf}	f	{opt, to_obf}	$o_{\text{obf}}(f)$	{}	$o_{\text{opt}}((o_{\text{obf}}(f))$	{opt}

Fig. 5. Combined code transformation $o'_{\text{opt}} \circ o'_{\text{obf}} \circ o'_{\text{opt}}$ on a function f depending on its properties.

Breakpoint detection, denoted $o_{\text{detect\_bp}}$:

$$o'_{\text{detect\_bp}} = h \left(\begin{array}{l} o_{\text{detect\_bp}}, \\ P_f \rightarrow P_f \cup \{\texttt{forbid\_bp}\}, \\ P_f \rightarrow \texttt{has\_bp} \notin P_f \end{array} \right)$$

Using the qualified versions, it becomes impossible to combine the original obfuscations and to build an incorrect program due to an invalid interaction between the two passes: applying the first pass generates a property that triggers a different behavior of the second pass.

Moreover, by changing the initial properties of the program, it becomes possible to inform the compiler that a given function will be debugged (*e.g.* during the development process) and the obfuscation adding the detection of breakpoints will skip that particular function.

4) Formalization of existing practices, optnone: Since version 3.4, LLVM adds the function attribute `optnone` to all functions compiled with `-O0`, in order to prevent their optimisation at link time. This can be modeled using the application:

$$o'_{\text{optnone}} = h(o, P_f \rightarrow P_f, P_f \rightarrow \texttt{optnone} \notin P_f)$$

The advantage of formalizing this behavior as part of an external declaration is that it does not clutter the code of each optimisation with the managment of property interactions.

V. Application to an industrial compiler: Epona

A. Using properties to shedule passes within Epona

During the development of the closed-source LLVM bitcode obfuscator named Epona by QuarksLab [19], we found out a subtle change in the definition of what a "semantic preserving" code transformation is, in the context of obfuscation.

By nature, some obfuscations rely on the structure of the code. For instance the addition of integrity checks could verify the depth of the call-stack at different points of the program. This kind of introspection on the code content is not part of the semantics of the program, so a compiler is free to ignore it. In the case of the call stack, this does not prevent transformations that modify it, like inlining and outlining, but then these transformations would break the obfuscation because they would make the call stack larger or smaller. The property system described in this paper proved to be a simple yet efficient solution to the problem.

Obfuscations from the Epona compiler all interact with the property system. It has been used in different situations:

- The `to_obf` and `opt` qualifiers presented in Section IV-B1 have been introduced to reduce compilation time in case of local obfuscation, making the second optimisation round a no-op in many cases. This optimisation has significantly reduced the test suite runtime in several scenarii.
- Properties similar to `forbid_bp` and `has_bp` used in the definition of $o'_{\text{rand\_code}}$ and $o'_{\text{detect\_bp}}$ have been used to enforce the legality of obfuscation compositions.
- A dedicated property `epona_generated` has been used to change the obfuscation behavior on code injected by the compiler. The compiler can inject debugger detection code and raise a warning if it ends up not being obfuscated.

B. Integration within source code

Both Tigress and O-LLVM use command line arguments to specify the obfuscation pipeline. Additionally, O-LLVM can use function attributes. In addition to these approaches, Epona uses a directive mechanism to specify the required obfuscations, the functions they apply to and their order, as in a stack. Without loss of genericity, we also extended the obfuscations to code blocks, where decorated blocks are first outlined to single functions so that they fit in the model.

Considering the example in Listing 7, the order of execution of the obfuscation is the following:

1) `a[i] ^= 42` in function `xoring1` is outlined to a temporary function,
2) `MBA` is applied on `a[i] ^= 42` in the temporary function,
3) the temporary function is inlined into `xoring1`,
4) the entire loop in function `xoring1` is outlined to a temporary function,
5) `OpaquePredicates` is applied on the entire loop in the temporary function,
6) the temporary function is inlined into `xoring1`,
7) `ControlFlowGraphFlattening` is applied on `xoring1`,

```
#pragma obfuscate DuplicateBasicBlocks(ratio=.5)
#pragma obfuscate ControlFlowGraphFlattening
void xoring1(int a[8]) {
    #pragma obfuscate OpaquePredicates
    for(int i = 0; i < 8; ++i)
        #pragma obfuscate MBA
        a[i] ^= 42;
}
```

Listing 7. Composition of obfuscation directives.

8) `DuplicateBasicBlocks` is applied on function `xoring1` with a ratio of 0.5,

A dual view of these directives can also be handled by the obfuscator through a configuration file that describes the location of each directive in terms of function name. That way the original code is untouched and the obfuscation process is separated from the original source code. This does not include the possibility to mark blocks of instructions though.

Note that in order to integrate this dynamic pass scheduling within LLVM, it is not possible to entirely rely on the static pass pipeline used by LLVM. As the default optimisation scheme is static, a new pass has been inserted after the first optimisation round. This pass launches a new pass manager that applies obfuscation passes based on command line arguments, function directives and/or a dedicated file content. It then adds a second optimisation round and finally a symbol stripping pass.

C. About randomness

The reproducibility of performance has also been a recurring problem. One way to tackle this issue in the context of MBA was to group obfuscations by category, each category having the same performance impact. That way randomness still provides diversity, while not significantly affecting performance reproducibility. Note that this does not prevent performance divergence if a later obfuscation can only be applied to a specific pattern. Figure 6 illustrates the dispersion obtained for different obfuscation levels: within each level, there is a low variability in the number of generated lines, and each level maps to a different group.

VI. Discussion

The concept of specific control language to manipulate the compilation of a program has been studied for generating programs with better performances and/or with more flexibility and ease of development. To describe image processing pipelines, which contains dozens of loop nests and are difficult to optimise with traditional loop optimisation algorithms, Halide [20] splits the pipeline into an algorithm (what is computed) and a schedule (how it is computed). An image processing pipeline is described in Halide as a composition of side-effect free functions. Each function computes the color of a pixel given its coordinates. This way a Halide program describes the computing algorithm for the value but not when and where these values are computed and stored.

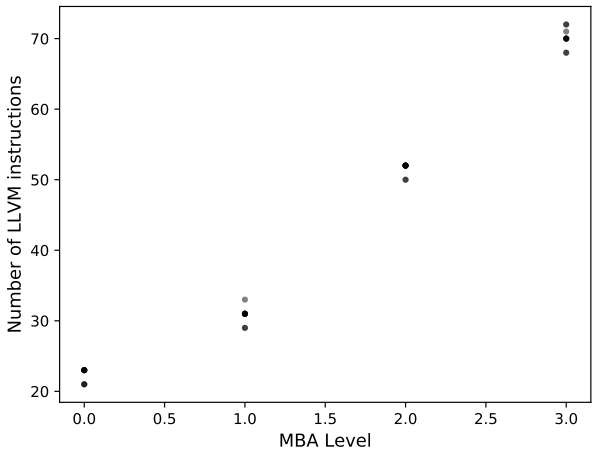

Fig. 6. Number of generated lines after Mixed Boolean Arithmetic on a simple add function `int foo(int x, int y) { return x + y;}`, for different obfuscation levels, using a different seed for each run.

The scheduling for each function is described separately, and the compiler generates the pipeline according to the schedule.

To evaluate the best-performing composition of loop optimisations, CHiLL [21] uses empirical optimisation and transformation scripts that contain sequences of loop transformations, each with a set of parameters. A decision algorithm derives a set of transformation scripts with unbound parameters. These scripts are then used as input of a search engine that binds the parameters and are compiled and executed to determine the best-performing sequences and parameters.

PyPS [22] is a pass manager API for the PIPS source-to-source parallelizing compiler framework. It offers more flexibility in the description of pass composition and to dynamically manipulate the pass manager. PyPS provides high-level compilation abstractions such as passes, functions or loops, which can be used in a Python script to describe the compilation flow with a lower granularity. The use of a programming language also offers high-level constructions to drive the compilation process, such as conditional and loop control structures. Theoretically, one could gain advantage of a general purpose language to implement the model proposed in this article, but nothing is formalized that way in PyPS, which focuses on scripting pass management.

The concept of *Obfuscation Executive* is studied in detail in [23]. The concept is close to a dynamic pass manager associated to a terminating condition, generally a set of metrics to fulfill. In the paper, each obfuscation is associated to a *cost*, a *potency* and a set of pre/post requirements and pre/post suggestions. The former share similarities with our property system, the latter is used to help an automatic process to build a finite state machine that should be able to fulfill the terminating condition, based on the accumulated cost and potency. To guide the search within the FSM, the paper assumes determinism in the application of each obfuscation through a

fixed potency and cost, ignoring the intrinsic randomness of the application of some obfuscation. It also does not try to model interaction with regular optimisations in terms of costs and properties.

The pass ordering problem applied to obfuscation has been studied in [24]. In this paper, the authors focus on the impact of obfuscation order, emphasising the impact of the ordering on the potency of the obfuscation. The authors also mention the impact of optimisation on the resulting binary, stating that the tool used to attack the obfuscated binary, KLEE, first performs a generic optimisation round. Finally, they did get some variability in the measurements, which could be linked to the random nature of the obfuscations, as seen in Section II. They conclude that an optimal order can be found for the combination of a limited number of passes, and within the limited scope of an attack based on KLEE.

VII. Conclusion and Future Works

Code obfuscation introduces a new aspect to the general problem of pass scheduling, because of the pseudo-random behavior of a large class of obfuscations, and because of the subtle interactions between obfuscations, and between obfuscation and generic optimisation. This paper details these issues through a series of case study and experiments, then proposes a formalism based on code properties manipulated by the obfuscations to enforce pre and post conditions and explicit manipulation of the random state. The formalism is validated in the implementation of the industrial compiler EPONA, where it efficiently solves the problems introduced by obfuscation, and also captures several existing behavior of optimising compilers.

This formalism helps to design correct obfuscation flows, but the task of building the pass order is still delegated to an expert. This can be a problem as this requires both knowledge from the application, to identify pieces of code to obfuscate, and a reverse-engineering background to identify the threats and produce an obfuscation executive that counter balances them. Extending the formalism to capture part of this knowledge is still a largely open field.

Acknowledgment

The initial research around an LLVM-based code obfuscator was funded by the French "Direction Générale de l'Armement" through a RAPID project.

The authors would like to thank the reverse engineers from QuarksLab for their valuable feedback during the development process of the tool, and Béatrice Creusillet, Marion Videau and Ninon Eyrolles for their careful reviews.

References

[1] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang, "On the (im)possibility of obfuscating programs," *J. ACM*, 2012.

[2] S. Garg, C. Gentry, S. Halevi, M. Raykova, A. Sahai, and B. Waters, "Candidate indistinguishability obfuscation and functional encryption for all circuits," in *FOCS'13*, 2013.

[3] B. Barak, "Hopes, fears, and software obfuscation," *Communications of the ACM*, 2016.

[4] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," Tech. Rep., 1997.

[5] I. You and K. Yim, "Malware obfuscation techniques: A brief survey," in *2010 International Conference on Broadband, Wireless Computing, Communication and Applications*, Nov 2010, pp. 297–300.

[6] T. László and Á. Kiss, "Obfuscating C++ programs via control flow flattening," in *Proceedings of the 10th Symposium on Programming Languages and Software Tools (SPLST 2007)*, Dobogókő, Hungary, 2007, p. 15–29.

[7] Y. Zhou, A. Main, Y. X. Gu, and H. Johnson, *Information Hiding in Software with Mixed Boolean-Arithmetic Transforms*, 2007.

[8] S. Banescu, C. S. Collberg, V. Ganesh, Z. Newsham, and A. Pretschner, "Code obfuscation against symbolic execution attacks," in *Proceedings of the 32nd Annual Conference on Computer Security Applications, ACSAC 2016, Los Angeles, CA, USA, December 5-9, 2016*, 2016, pp. 189–200. [Online]. Available: http://dl.acm.org/citation.cfm?id=2991114

[9] A. Majumdar, A. Monsifrot, and C. Thomborson, "On evaluating obfuscatory strength of alias-based transforms using static analysis," in *2006 International Conference on Advanced Computing and Communications*, Dec. 2006, pp. 605–610.

[10] B. Dang, A. Gazet, E. Bachaalany, and S. Josse, *Practical Reverse Engineering: X86, x64, ARM, Windows Kernel, Reversing Tools, and Obfuscation*, 1st ed. Wiley Publishing, 2014.

[11] C. Collberg, S. Martin, J. Myers, and J. Nagra, "Distributed application tamper detection via continuous software updates," in *Proceedings of the 28th Annual Computer Security Applications Conference*, ser. ACSAC '12. New York, NY, USA: ACM, 2012, pp. 319–328. [Online]. Available: http://doi.acm.org/10.1145/2420950.2420997

[12] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin, "Obfuscator-LLVM – software protection for the masses," in *SPRO'15*, B. Wyseur, Ed., 2015.

[13] T. Hansen and D. E. E. 3rd, "US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)," RFC 6234, May 2011. [Online]. Available: https://rfc-editor.org/rfc/rfc6234.txt

[14] M. Amini, C. Ancourt, F. Coelho, B. Creusillet, S. Guelton, F. Irigoin, P. Jouvelot, R. Keryell, and P. Villalon, "PIPS Is not (only) Polyhedral Software, adding gpu code generation in pips."

[15] F. Perriot, "Defeating polymorphism through code optimization," ser. Virus Bulletin Conference. The Pentagon, Abington, Oxfordshire, England: Virus Bulletin Ltd., Sep. 2003, pp. 142–159.

[16] S. K. Udupa, S. K. Debray, and M. Madou, "Deobfuscation: reverse engineering obfuscated code," in *12th Working Conference on Reverse Engineering (WCRE'05)*, Nov 2005, pp. 10 pp.–.

[17] N. Eyrolles, "Obfuscation with Mixed Boolean-Arithmetic Expressions : reconstruction, analysis and simplification tools," Theses, Université Paris-Saclay, Jun. 2017. [Online]. Available: https://tel.archives-ouvertes.fr/tel-01623849

[18] M. Gagnon, S. Taylor, and A. K. Ghosh, "Software protection through anti-debugging," *IEEE Security and Privacy*, 2007.

[19] S. Guelton, A. Guinet, J. M. Martinez, and P. Brunet, "Challenges when building an llvm bitcode obfuscator," https://llvm.org/devmtg/2017-10/, Oct. 2017.

[20] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," *ACM SIGPLAN Notices*, 2013.

[21] C. Chen, J. Chame, and M. Hall, "Chill: A framework for composing high-level loop transformations," Tech. Rep., 2008.

[22] S. Guelton, "Building source-to-source compilers for heterogenous targets," Ph.D. dissertation, Télécom Bretagne, 2011.

[23] K. Heffner and C. Collberg, "The Obfuscation Executive," in *Information Security*, ser. Lecture Notes in Computer Science, K. Zhang and Y. Zheng, Eds. Berlin, Heidelberg: Springer Berlin / Heidelberg, 2004, vol. 3225, ch. 36, pp. 428–440. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-30144-8_36

[24] W. Holder, J. T. McDonald, and T. R. Andel, "Evaluating optimal phase ordering in obfuscation executives," in *Proceedings of the 7th Software Security, Protection, and Reverse Engineering / Software Security and Protection Workshop*, ser. SSPREW-7. New York, NY, USA: ACM, 2017, pp. 6:1–6:12. [Online]. Available: http://doi.acm.org/10.1145/3151137.3151140