Greybox Program Synthesis: A New Approach to Attack Obfuscation

Robin David

2021-04-08

Contents

I. Introduction

Program obfuscation is getting more and more exposure, and consequently approaches to defeat them as well. While control-flow obfuscation attracted attention the last few years, fewer approaches focused on dataflow. Program synthesis appears to be a promising approach to target this kind of obfuscation. This whitepaper aims first at introducing broad principles and its application to obfuscation. Then we will show all advances made at Quarkslab on our approach since the publication of the first results[1] at the Binary Analysis Workshop 2020 *(collocated with NDSS)*. That will be the opportunity to get more into details, on the implementation and experiments performed since to improve the algorithm, and especially, its performances and scalability.

More specifically, a focus will be given to our "Greybox Synthesis" algorithm combining a pure black-box I/O based synthesis (relying on precomputed expressions) with a search algorithm on the semantic (white-box aspect). The semantic is obtained through symbolic execution thanks to the Triton framework. Various extensions like linearization or learning will be presented as well as updated benchmarks results.

We will then discuss how to deobfuscate various cases embedding various obfuscation like MBA[2] or VM using QSynthesis. Then an overview of the IDA integration will be shown with an end-to-end deobfuscation starting from obfuscated code up to to reassembled clear instructions.

II. Background

a. Program Synthesis Primer

Program Synthesis is the mean of generating a program given the expected behavior via a specification of it. Such approach found multiple usages[3], especially in optimization[4] and deobfuscation (our use-case). As shown on Figure 1, in the context of

[1]QSynth - A Program Synthesis based approach for Binary Code Deobfuscation Robin David (Quarkslab), Luigi Coniglio (University of Trento), Mariano Ceccato (University of Verona) https://archive.bar/pdfs/bar2020-preprint9.pdf

[2]Yongxin Zhou, Alec Main, Yuan Xiang Gu, Harold Johnson: Information Hiding in Software with Mixed Boolean-Arithmetic Transforms. WISA 2007: 61-75

[3]Sumit Gulwani, Oleksandr Polozov, Rishabh Singh Program Synthesis https://www.microsoft.com/en-us/research/wp-content/uploads/2017/10/program_synthesis_now.pdf

[4]S. Bansal and A. Aiken, “Automatic generation of peephole superopti-mizers,” in Proceedings of the 12th International Conference on Archi-tectural Support for Programming Languages and

program synthesis the specification **is the semantic of the program itself** that we usually want to preserve through synthesis. The only additional specification parameter is the intended goal e.g. a faster program, a cleaner program that we will call the *fitness function* for the rest of the article. So a synthesizer is a program generating or rewriting new programs.

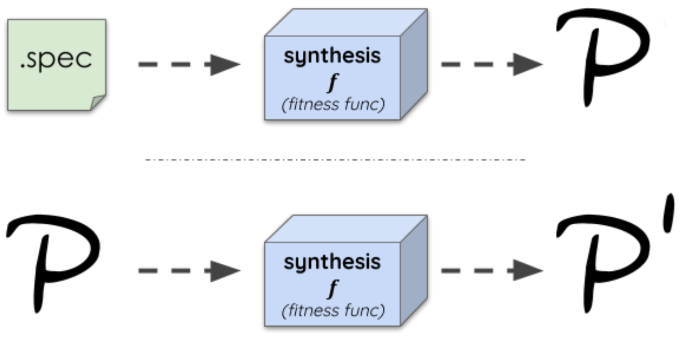
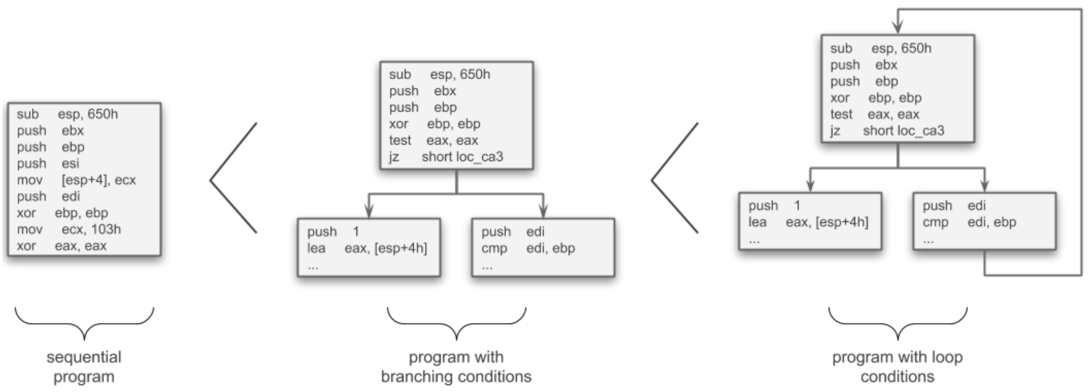


Figure 1: Abstract functional view of synthesis.

Programs considered in this research are not programs as we are used to manipulate and to execute but a more abstract definition of it. The Figure 2 shows some elementary "classes" of programs that incrementally bring a load of complexity to handle and thus to be synthesized.



```
sub     esp, 650h
push    ebx
push    ebp
push    esi
mov     [esp+4], ecx
push    edi
xor     ebp, ebp
mov     ecx, 103h
xor     eax, eax
```

```
sub     esp, 650h
push    ebx
push    ebp
xor     ebp, ebp
test    eax, eax
jz      short loc_ca3

push    1
lea     eax, [esp+4h]
...

push    edi
cmp     edi, ebp
...
```

```
sub     esp, 650h
push    ebx
push    ebp
xor     ebp, ebp
test    eax, eax
jz      short loc_ca3

push    1
lea     eax, [esp+4h]
...

push    edi
cmp     edi, ebp
...
```

sequential program
program with branching conditions
program with loop conditions

Figure 2: Schematic classes of program structure complexity.

The input program can be of arbitrary complexity, but the synthesis is usually bound to a certain class which is usually sequential programs also referred as "loop-free

Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006, 2006, pp. 394–403. https://theory.stanford.edu/~aiken/publications/papers/asplos06.pdf programs"[5]. Thus, input programs essentially require to be of the same class to be successfully synthesized *(it is rather improbable to synthesize a sequential program if the original contains branching conditions as the semantic would not be preserved)*. Consequently we don't consider the whole program but usually sequential sub-parts of it.

A sequential program, is a set of instructions *(not necessarily contiguous)* which link inputs to outputs through the semantic of instructions. A dataflow expression is thus obtained for each output which is usually represented as a first-order logic expression on bitvectors. An expression is structured as an AST (Abstract-Syntax-Tree). For the rest of the article, we will refer to them as "expressions".

To summarize, the synthesizer will intend to synthesize expressions with some objective functions *(or "fitness" function)*. Applied on deobfuscation the fitness function will be to obtain expressions as tight as possible *(fitness seems really appropriate in this context :))*.

For a broader view of synthesis, existing approaches and application, there are other interesting publication in the literature[6][7][8]. Next section dives in previous research applied on obfuscation.

b. Previous Research

Applied on optimization, a reference publication is Souper[9], but let's focus here on binary-level synthesis applied to reverse-engineering and more specifically deobfuscation. A pioneer in this field is Rolf Rolles[10]. He presented an enumerative approach for CPU emulator synthesis and peephole deobfuscation, as well as a template-based approach for metamorphic extraction. All three were inspired from

[5] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. In ACM SIGPLAN Notices, volume 46, pages 62–73. ACM, 2011.

[6] https://www.cs.utexas.edu/~bornholt/post/synthesis-explained.html

[7] https://courses.cs.washington.edu/courses/cse507/14au/index.html https://www.youtube.com/watch?v=KpDyuMIb_E0&ab_channel=ClojureTV

[8] https://fitzgeraldnick.com/2020/01/13/synthesizing-loop-free-programs.html

[9] Raimondas Sasnauskas and Yang Chen and Peter Collingbourne and Jeroen Ketema and Gratian Lup and Jubi Tanejaand John Regehr Souper: A Synthesizing Superoptimizer, 2017 https://arxiv.org/pdf/1711.04422.pdf

[10] Rolf Rolles, Program synthesis in reverse engineering, in No Such Conference Paris, France, 2014 https://www.msreverseengineering.com/s/Program-Synthesis-in-Reverse-Engineering.pdf

earlier academic work[11][12][13].

Lately, Tim Blatzyko proposed Syntia[14], a synthesis approach based on a stochastic search using Monte-Carlo-Tree-Search (MCTS) to find expressions with equivalent I/O behavior given a set of I/O pairs. The algorithm is deriving an expressions up until finding one having the exact same I/O behavior. For a given program (expression) the synthesizer result is thus essentially boolean on whether a program is found or not.

These two synthesizers are targeting various dataflow based obfuscations which are also the target of our study and our synthesizer. Many other interesting publications discuss synthesis but not specifically applied on obfuscation[15].

On our side, preliminary results have been presented on Quarkslab blog[16] in the context of our dynamic trace analysis framework analysis. Later, academic results were presented at BAR[17]. Since, it once again changed significantly as we explored new simplification strategies and multiple other improvements.

III. Synthesis Algorithm

An accurate description of the greybox synthesis algorithm is the following:

> Offline enumerative synthesis approach guided by an AST semantic search strategy.

Behind this enigmatic definition lays two main components:

[11] S. Bansal and A. Aiken, "Automatic generation of peephole superopti-mizers," in Proceedings of the 12th International Conference on Archi-tectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006, 2006, pp. 394–403. https://theory.stanford.edu/~aiken/publications/papers/asplos06.pdf

[12] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. In ACM SIGPLAN Notices, volume 46, pages 62–73. ACM, 2011.

[13] Patrice Godefroid and Ankur Taly. Automated synthesis of symbolic instruction encodings from i/o samples. In ACM SIGPLAN Notices, volume 47, pages 441–452. ACM, 2012. http://theory.stanford.edu/~ataly/Papers/pldi12.pdf

[14] Tim Blazytko and Moritz Contag and Cornelius Aschermann and Thorsten Holz Syntia: Synthesizing the Semantics of Obfuscated Code 26th USENIX Security Symposium, USENIX Security 2017, Vancouver

[15] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh Program Synthesis https://www.microsoft.com/en-us/research/wp-content/uploads/2017/10/program_synthesis_now.pdf

[16] https://blog.quarkslab.com/exploring-execution-trace-analysis.html

[17] QSynth - A Program Synthesis based approach for Binary Code Deobfuscation Robin David (Quarkslab), Luigi Coniglio (University of Trento), Mariano Ceccato (University of Verona) https://archive.bar/pdfs/bar2020-preprint9.pdf

- The **offline enumerative synthesis** approach which given an expression provides a synthesized expression using a standard I/O based synthesis. The peculiarity is the offline enumeration which means that all possible programs will be derived *(up to a given size/depth)* once and for all. They are thus precomputed when it comes to synthesizing an expression. Syntia has to perform the derivation for each expression it has to simplify. We call it "online" in opposition to precomputed approach which is "offline". All precomputed expressions are stored in a database described hereafter.
- The **AST semantic search strategy** which given a program dataflow expression represented as an AST will first try to synthesize the root node. If unsuccessful, it will iterate the AST to opportunistically trying to synthesize sub-ASTs. This provides a good trade-off with the boolean aspect of the I/O synthesis. It is the whitebox aspect of the algorithm as it manipulates the semantic of the expression. A shortcoming is that the search is impacted by the syntactic complexity of the AST and thus the obfuscation itself. To address this issue, various search strategies are described below.

Original Search Strategy

The original search strategy was an iterative random-walk BFS search iterating the AST up until finding synthesizable sub-tree. Every time a sub-tree is successfully synthesized it is replaced by a placeholder node acting as a new variable in the expression. The simplification search was repeated up until the AST has been reduced to a single node or no replacement took place during an iteration. After that, all placeholder variables were replaced by the effective AST yielding the synthesized expression.

A graph animation of the algorithm is available: here.

This approach appeared to be the most efficient one in terms of AST sizes reduction. But it implies always restarting from the root node and all editions of the AST comes at some costs *(due to internal Triton representation)*. On large ASTs the complexity quickly raises making the algorithm very slow. Even if providing optimal results in terms of expression sizes, the runtime complexity was a strong limitation.

Top-Down Search Strategy

This approach is the more instinctive as it iterates the whole AST once in a DFS manner and substitute any sub AST that can be synthesized. This algorithm is thus truly $\mathcal{O}(n)$ with n the number of AST nodes.

A graph animation of the algorithm is available: here.

Experimentally, results have shown not to be optimal as it cannot perform recursive simplification. Also, it does not take advantage that parts of the expression might be identical *(where some temporary abstraction can be made as in previous strategy)*.

Top-Down & Bottom-Up Strategy

This strategy is a variant of the first one trying to reduce some complexity bottlenecks while keeping some synthesis potency. In this algorithm, the search does not restart from the root node each iteration. Instead, the search is implemented as a BFS from the root node followed by a BFS from leaves to the root node.

A graph animation of the algorithm is available: here.

The intuition is the following, if a node get synthesized, one of its parents nodes might become synthesizable by means of reducing the number of variables involved in the computation. For example, if an expression contains 4 variables but precomputed tables used only contain 3 variables no lookup can be performed. But if the synthesis reduces the expression to 3 variables then it enables synthesizing the node.

Visualizing Simplification

For a given synthesized expression, it is somewhat difficult to assess the implication of the AST search. Indeed, it is difficult to know whether a single sub-AST has been simplified or multiple smaller ones. Such question does not arise for black-box synthesis as only the root node can be simplified. To answer this question we made a simple tool enabling visualizing the search on the obfuscated expression AST and simplifications that takes place. The Figure 3 below shows the synthesis search on 3 different expressions ranging from a few hundred nodes to more than 100k nodes. One can see that even though the root node cannot be synthesized all at once the search and the synthesizer enables simplifying many sub-graphs in the expression resulting in more understandable expressions! Green nodes are synthesized sub-ASTs at the end of the process.

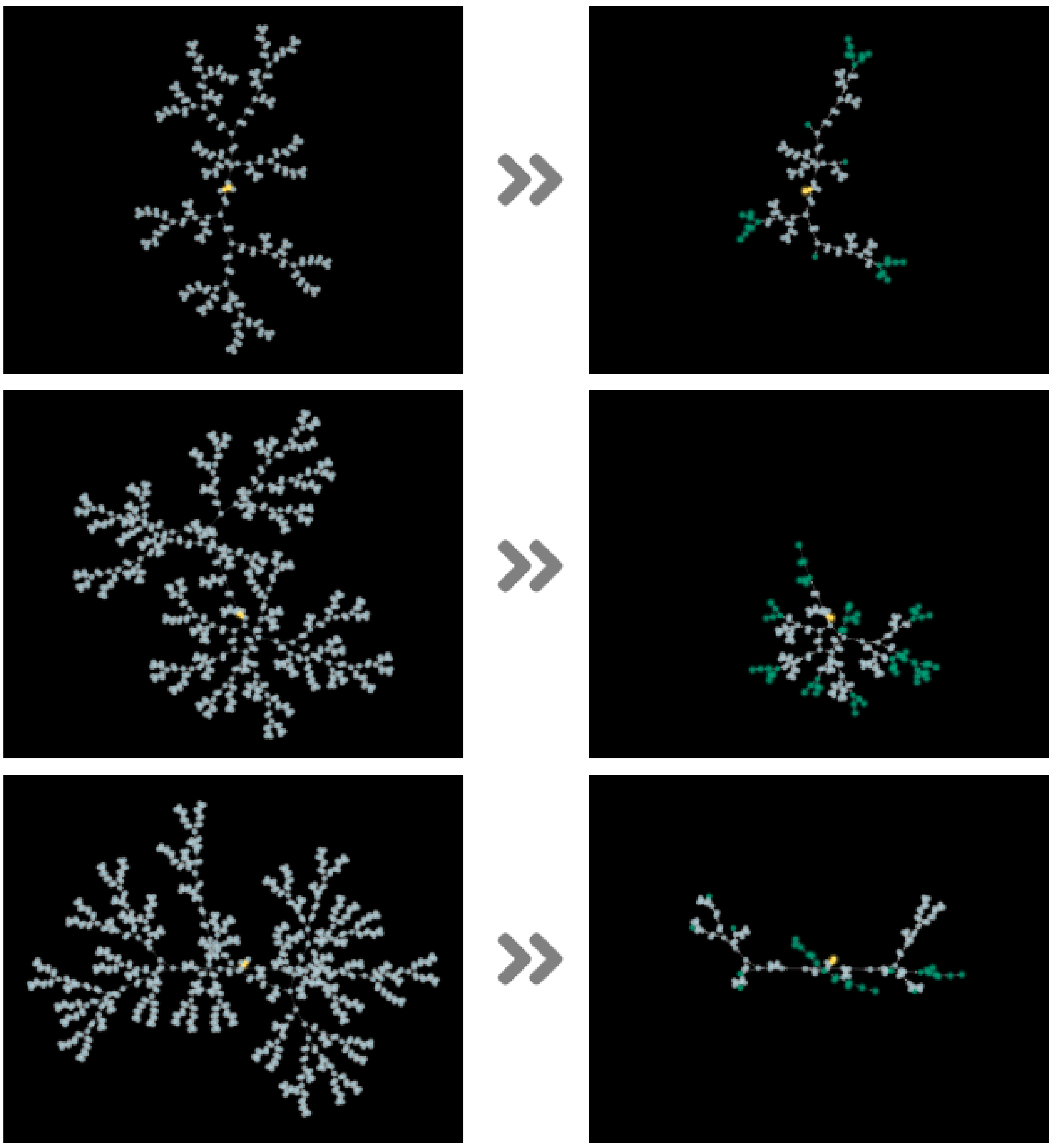


Figure 3: Visualization of ASTs before and after synthesis *(on three expressions)*.

Beyond Dynamic And Symbolic Execution

Our previous work retrieved obfuscated expressions through Dynamic Symbolic Execution, and thus on execution trace. At a given location, it enables computing easily the expression by backtracking backward on its dependencies up to the first instruction executed. Nonetheless, the synthesis algorithm is not bounded to this context. As the input of the algorithm is an expression AST, it can be computed from any analysis technique symbolic execution or not, it can originate from another intermediate represent or be computed entirely statically.

In our context, it can also be performed on static path which emancipate the algorithm from having to run the target concretely and set it free from reaching the given location. It implies to perform fake loading and initialization of the Triton symbolic state. However it works fairly well, especially on this kind of obfuscation where most instructions are computational ones (not calling external APIs doing memory operations). The main difference is that every register or memory area firstly read *(without having been previously written)* will be symbolized inducing an overhead of symbolic variables.

IV. Table Generation

Pre-computed tables are the cornerstone of this approach, they have to be as diversified and representative as possible. Even though the generation has to be performed only once, we tried to make generation efficient, compact on disk, and more than everything, keeping logarithmic accesses in the database. This section will drive you through the process of building a table design that scale for both generation and lookup.

From an abstract point of view, tables provide the mapping between N variables of a given size to a program (an expression) of a given size. Let's call **domain** the type of input variables and the associated output size. This signature of a table can be defined by: $\{N^a, N^b..\} \mapsto \mathcal{N}$ with a set of input variables of various sizes (a, b) and yield an expression of size $\mathcal{N}$. As such, an expression to synthesize belongs to a specific domain and can only be resolved with a table of the same domain. For most of the following experiments, we consider the $3.N^{64} \mapsto N^{64}$ domain containing up to 3 variables of size 64 yielding a result on 64 bits.

Given a set of variables and a set of operators (+, -, *, ..), tables are derived by applying recursively all operators on all combinations of variables[18]. Each expression generated is then evaluated on a set of inputs (Y valuations of all variables) which provide a vector of output values *(array of integers)*. This array is then hashed and this hash forms the key to access such expression. Any other expression generated the same by means of evaluating it on the same inputs would be considered semantically equivalent. The whole database is generated from the smallest to the

[18]The complexity of that process is very boldly exponential.

biggest expression ensuring optimality of the size (wrt. to variables and operators). As such, during the generation process, if an expression generates the same hash as a previous one it will be dropped as it is necessarily bigger.

Table generation is a very intensive process as it implies to evaluate an expression on a vector of input values. Thus the generation memoize intermediate results to never re-evaluating a given sub-AST twice. For instance, if (a+b) and (c-b) generate respectively the output vectors $V1$ and $V2$. Once combined with another operator let's say +, each vector items will be evaluated two by two ($V1[i] + V[i]$) yielding the output vector associated to (a+b)+(c-b). That evaluation is the most time-consuming operation but the memoization makes it very efficient.

JITTing Expression Evaluation

Table generation is made in Python, which is an order of magnitude slower than compiled languages and not trivially simulate low-level arithmetic operations on bounded integers *(with overflows, etc.)*. We used `dragonffi` an awesome jitting engine backed by LLVM available in open-source[19] . That will enable us to perform native integer operations and performs all vector operations in native to limit goings and comings from Python and native world. The snippet below shows how to perform a function that will increment all uint64 of an array by one and how to call it from Python.

```
FFI = pydffi.FFI()
N = 10
ArTy = FFI.arrayType(FFI.ULongLongTy, N)

ar = ArTy()  # values are random

CU = FFI.compile('''
#include <stdio.h>
#include <stdint.h>
void inc(uint64_t* buf, size_t n)
{
    for(int i=0; i < n; i++) {
      buf[i] = buf[i] + 1;
    }
}
''')

CU.funcs.inc(pydffi.ptr(ar), 10)
```

[19] https://github.com/aguinet/dragonffi

For generation, all operators are thus jitted with the appropriate arrays length and integer size. The table below shows an example for generating a database of 1.535.467 entries jitting without and with array. Pure python implementation is even slower.

	JIT	JIT-array
RAM (16GB)	27.6%	7.4%
Time (in s)	1m29s	30s

Generation Throughput

At the moment the largest table generated contains **374,726,312 entries** and the generation took 244m27s on a machine with 235GB of RAM. The average throughput is thus **25,548 entries per second**. It only considers the generation time. Writing the table on disk takes a bit of time.

Google Level-DB Against Pony-ORM

Tables were first pickle files, but as they have grown we switched to ponyorm[20] a fast and lightweight ORM for Python to store entries using the hash as a primary key. However it turned out not to scale to such a high number of entries as insertion seems linear in the number of entries. Not making hashes as key makes the lookup slow as hell. As tables are basically key-value pairs where the key is the hash of the output vector and value is the expression string, we focused on key-value databases. Multiple of them exists like Berkley-DB[21], Level-DB[22] by Google or RocksDB[23] by Facebook. Among them we choose Level-DB.

Such database store keys as tries *(or similar)* which satisfy our needs to have a logarithmic lookup *(in the number of entries)*. The time taken to query an entry between PonyORM SQLite (with hashes as primary keys)[24] and level-DB is the following:

[20] https://ponyorm.org/
[21] https://github.com/berkeleydb/libdb
[22] https://github.com/google/leveldb
[23] https://github.com/facebook/rocksdb
[24] Hashes as primary keys does not scale at all, as the insertion is linear in the number of entries. Generating tables with such mechanisms basically does not scale for large tables.

- PonyORM: 565 µs
- Level-DB: **122 µs**

With this database architecture, the cost of the pure I/O synthesis algorithm is equal to the one of the lookup, thus ~100µs. Also Level-DB automatically caches results thus querying the same hash twice *(which happen often in a single binary)* lower the lookup to 62.9 µs. In terms of disk size, 370M entries weight approximately 14GB which represent a ~27M entries per GB in average.

Expression Linearization

We also considered using expression linearization[25] to represent expressions as normalized equations when they used supported operators (+, -, *). The possibility to linearize every expression before adding them to the table was added to the generation process. For that, SymPy[26] a Python library for symbolic mathematics, has been chosen.

Original	Linearized
(a-(c-a))	2*a-c
((a-b)-(a+a))	-a-b
(-b-(b+c))	-2*b-c
((a+a)--a)	3*a
(cˆ(b+c))	cˆb+c
(-a-(a+a))	-3*a
(b-((c-b)-(b+b)))	4*b-c
(-a-((b+b)+(b+c)))	-a-3*b-c
(-a-((a+a)+(b+b)))	-3*a-2*b
(-c-((c+c)+(c+c)))	-5*c
…	…

As these examples show, it also introduces constants to expressions! That aspect lacks in our derivation mechanism and is an issue in synthesis in general. On

[25] https://www.wikiwand.com/en/Linearization
[26] https://www.sympy.org/en/index.html

expressions it managed to linearize, the gain in node size is ~34% and the string representation ~63% *(as it reduces parentheses)*.

In terms of time, linearizing expressions during table generation completely annihilates all performances as Sympy is not designed to be as computation intensive and is purely in python (65h instead of 3h). In terms of results because of operators like (|, &, ˆ) very few expressions can be linearized. The impact on synthesized expressions is thus rather low. Still, it allowed synthesizing function of the Syntia benchmark to their most compact form:

```
((((((0xE640327FE72F517E + ((((((((0xABEA5477E23EB83 + (((((((0x6F648D9353ED62EA
- ((((((d * 0xABEA5477E23EB83)) + 0xCDFE6C00C685741)) * 0x17C34B8B445AFB2B))))
| ((((((((d * 0xABEA5477E23EB83)) + 0xCDFE6C00C685741)) * 0x17C34B8B445AFB2B))
+ 0x909B726CAC129D15))) * 0xABEA5477E23EB83)) + ((((d * 0xABEA5477E23EB83))
+ 0xCDFE6C00C685741)))))) + ((0xABEA5477E23EB83 + (((((((0x6F648D9353ED62EA
- ((((((d * 0xABEA5477E23EB83)) + 0xCDFE6C00C685741)) * 0x17C34B8B445AFB2B))))
| ((((((((d * 0xABEA5477E23EB83)) + 0xCDFE6C00C685741)) * 0x17C34B8B445AFB2B))
+ 0x909B726CAC129D15))) * 0xABEA5477E23EB83)) + ((((d * 0xABEA5477E23EB83))
+ 0xCDFE6C00C685741)))))))) - ((0xABEA5477E23EB83 * (((0x6F648D9353ED62EA
- ((((((d * 0xABEA5477E23EB83)) + 0xCDFE6C00C685741)) * 0x17C34B8B445AFB2B))))
| ((((0x2F86971688B5F656 * (((((((0x6F648D9353ED62EA - ((((((d *
0xABEA5477E23EB83)) + 0xCDFE6C00C685741)) * 0x17C34B8B445AFB2B))))
| ((((((((d * 0xABEA5477E23EB83)) + 0xCDFE6C00C685741)) * 0x17C34B8B445AFB2B))
+ 0x909B726CAC129D15))) * 0xABEA5477E23EB83)) + ((((d * 0xABEA5477E23EB83))
+ 0xCDFE6C00C685741)))))) + 0x2136E4D958253A2C))))))) + ((((((d *
0xABEA5477E23EB83)) + 0xCDFE6C00C685741)) + (((((0x6F648D9353ED62EA
- ((((((d * 0xABEA5477E23EB83)) + 0xCDFE6C00C685741)) * 0x17C34B8B445AFB2B))))
| ((((0x2F86971688B5F656 * (((((((0x6F648D9353ED62EA - ((((((d *
0xABEA5477E23EB83)) + 0xCDFE6C00C685741)) * 0x17C34B8B445AFB2B))))
| ((((((((d * 0xABEA5477E23EB83)) + 0xCDFE6C00C685741)) * 0x17C34B8B445AFB2B))
+ 0x909B726CAC129D15))) * 0xABEA5477E23EB83)) + ((((d * 0xABEA5477E23EB83))
+ 0xCDFE6C00C685741)))))) + 0x2136E4D958253A2C))) * 0xABEA5477E23EB83))))))))
* 0x17C34B8B445AFB2B)) + 0x909B726CAC129D15))
```

To:

```
0x3 * d
```

Besides the low number of expressions that can be linearized with, SymPy also tends to introduce power expression (e.g.: x**2) however this operator is not supported by SMT and thus Triton (and most ASM semantics). Hence, when it happens it has to be expanded to its unrolled form, generating often suboptimal results. Better, leveraging symbolic reasoning as done by SymPy to better improve synthesis is left as future work.

Expression Learning

What if an expression to synthesize is smaller than the one synthesized by the I/O synthesizer? It can occur because the generation process is bounded, it generates expression with limited variables/operators, plus it does not involve any constants. If that happens, we can replace the expression in the database by the one given in input which is smaller. In this case, the expression will not be synthesized but it will improve the database.

> We implemented a learning mechanism enabling to improve the synthesizer by learning new smaller expressions.

The table below shows multiple examples taken from the benchmark binaries where table content was improved with new expressions learned.

Expression in DB	Learned expr. from binary
~(c - e) - (-c | e)	e + (~e | -c)
((e * e) + e) + (~e)	(e * e) - 0x1
(~e + e) + (~(-e))	0xFFFFFFFFFFFFFFFE + e
(e - d) - (~d)) | ~d	~d | (e + 0x1)
(~b + b) + (~(-b))	b - 0x2
(~b + b) + (e & b)	b + (~b | e)
(~a + a) + (~(-a))	~(0x1 - a)
(~d + d) + (a & d)	(~d | a) + d
((d * d) - d) - (~d)	~(0xFFFFFFFFFFFFFFFE - (d * d))
(~(-c)) + (~(c + c))	0xFFFFFFFFFFFFFFFE - c
(~b + b) + (d ^ b)	(b ^ d) - 0x1
(-d) - (d * d)	~d * d
(~d + d) + (-d & c)	((d - 0x1) | c) - d

Expression in DB	Learned expr. from binary
(~(-a) + (~(a + a))	~(a + 0x1)
…	…

Most expressions learned involve constants[27]. This learning mechanism is thus a step forward to address the constant issue in synthesis. Generalizing such learning mechanism would greatly improve synthesis potency and maybe enable learning other classes of programs (e.g. with branches).

V. Benchmarks

To assess the algorithm Syntia benchmark and custom Tigress 2.1[28] based benchmarks were used. Each benchmark was composed of a single binary containing 500 obfuscated functions with some defined Tigress passes. Test benchmarks are:

- Syntia: EncodeArithmetic *(MBA and arithmetic diversification)* and EncodeData *(data encoding)*;
- custom_EA: EncodeArithmetic *(but larger expressions to obfuscate)*;
- custom_VR_EA: Virtualize *(virtualization of the CFG)* and EncodeArithmetic;
- custom_EA_ED: EncodeArithmetic EncodeData (but way larger expressions).

These benchmarks are open-source and available on github[29] *(as part of the academic publication)*. After publication the first thing performed was switching to Triton 0.8 which brought various improvements and especially with regard to performances[30]. Without changing anything to synthesis time benchmarks on custom_EA switched from:

> Time DSE:0m36s Synthesis:1m9s Total:106.42s (Mean:0.21s/func)

To:

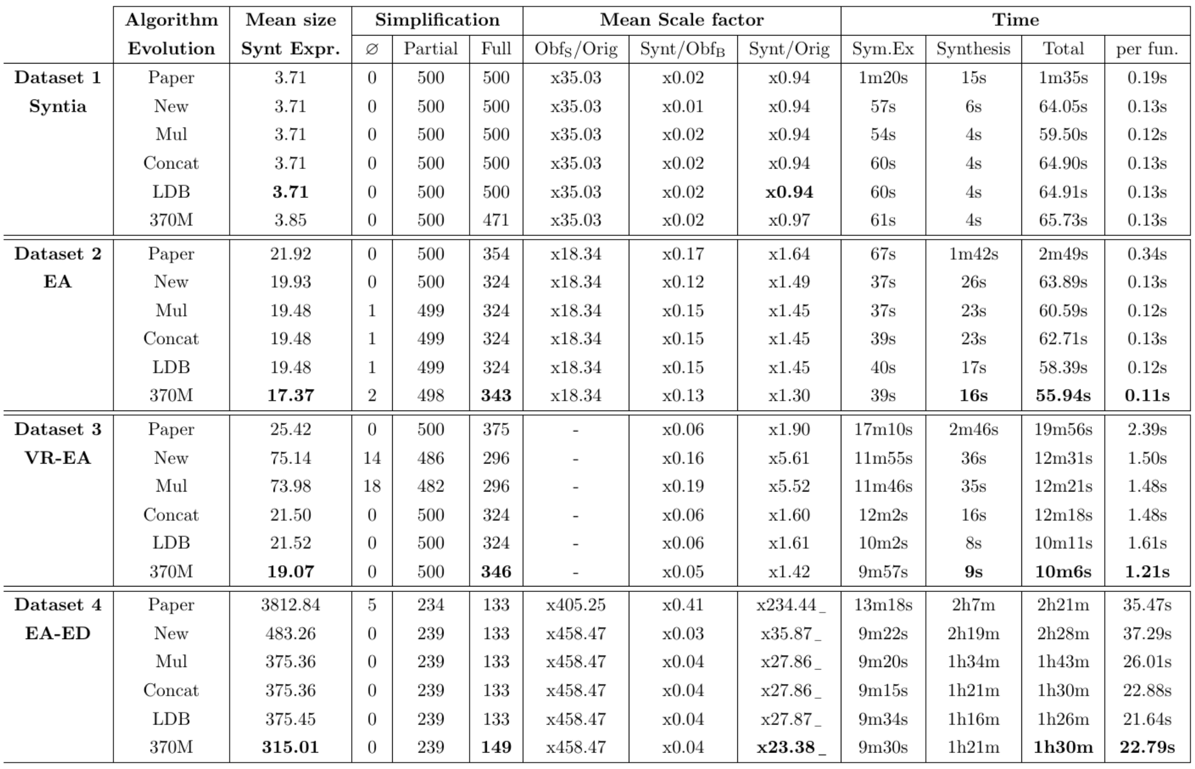
> Time DSE:0m39s Synthesis:0m28s Total:68.39s (Mean:0.14s/func)

[27]other expressions shows that optimality of generation is fallible
[28]http://tigress.cs.arizona.edu/
[29]https://github.com/werew/qsynth-artifacts
[30]https://blog.quarkslab.com/triton-v08-is-released.html

In addition, various improvements were brought to Triton and the synthesis algorithm improving overall performances and widening the gap with paper results. Two of them have to do with the low-level SMT representation of formulas by Triton. Thus these improvements profits to all Triton users by reducing SMT expressions sizes. Detailed improvements are:

- **New**: New implementation and migration to Triton 0.8. Use Top-Down search strategy and same tables as in the paper *(Pickle files at that time)*;
- **Mul**: Triton AST semantic improvement on MUL operations to avoid extracts of (un)signed extensions (PR#909);
- **Concat**: Improve folding of constant concatenation (Issue #907);
- **LDB**: Switch to Level-DB database mechanism *(using the same tables as the paper, thus 2,412,513 entries)*;
- **370M**: Switch to a 370 million entries database *(x153 more)*.

The table below shows performances starting from paper results up to the latest improvements all combined *(each line includes improvements of previous lines)*.



	Algorithm	**Mean size**	**Simplification**			**Mean Scale factor**			**Time**			
	Evolution	**Synt Expr.**	∅	Partial	Full	Obf$_S$/Orig	Synt/Obf$_B$	Synt/Orig	Sym.Ex	Synthesis	Total	per fun.
Dataset 1 Syntia	Paper	3.71	0	500	500	x35.03	x0.02	x0.94	1m20s	15s	1m35s	0.19s
	New	3.71	0	500	500	x35.03	x0.01	x0.94	57s	6s	64.05s	0.13s
	Mul	3.71	0	500	500	x35.03	x0.02	x0.94	54s	4s	59.50s	0.12s
	Concat	3.71	0	500	500	x35.03	x0.02	x0.94	60s	4s	64.90s	0.13s
	LDB	**3.71**	0	500	500	x35.03	x0.02	**x0.94**	60s	4s	64.91s	0.13s
	370M	3.85	0	500	471	x35.03	x0.02	x0.97	61s	4s	65.73s	0.13s
Dataset 2 EA	Paper	21.92	0	500	354	x18.34	x0.17	x1.64	67s	1m42s	2m49s	0.34s
	New	19.93	0	500	324	x18.34	x0.12	x1.49	37s	26s	63.89s	0.13s
	Mul	19.48	1	499	324	x18.34	x0.15	x1.45	37s	23s	60.59s	0.12s
	Concat	19.48	1	499	324	x18.34	x0.15	x1.45	39s	23s	62.71s	0.13s
	LDB	19.48	1	499	324	x18.34	x0.15	x1.45	40s	17s	58.39s	0.12s
	370M	**17.37**	2	498	**343**	x18.34	x0.13	x1.30	39s	**16s**	**55.94s**	**0.11s**
Dataset 3 VR-EA	Paper	25.42	0	500	375	-	x0.06	x1.90	17m10s	2m46s	19m56s	2.39s
	New	75.14	14	486	296	-	x0.16	x5.61	11m55s	36s	12m31s	1.50s
	Mul	73.98	18	482	296	-	x0.19	x5.52	11m46s	35s	12m21s	1.48s
	Concat	21.50	0	500	324	-	x0.06	x1.60	12m2s	16s	12m18s	1.48s
	LDB	21.52	0	500	324	-	x0.06	x1.61	10m2s	8s	10m11s	1.61s
	370M	**19.07**	0	500	**346**	-	x0.05	x1.42	9m57s	**9s**	**10m6s**	**1.21s**
Dataset 4 EA-ED	Paper	3812.84	5	234	133	x405.25	x0.41	x234.44 _	13m18s	2h7m	2h21m	35.47s
	New	483.26	0	239	133	x458.47	x0.03	x35.87 _	9m22s	2h19m	2h28m	37.29s
	Mul	375.36	0	239	133	x458.47	x0.04	x27.86 _	9m20s	1h34m	1h43m	26.01s
	Concat	375.36	0	239	133	x458.47	x0.04	x27.86 _	9m15s	1h21m	1h30m	22.88s
	LDB	375.45	0	239	133	x458.47	x0.04	x27.87 _	9m34s	1h16m	1h26m	21.64s
	370M	**315.01**	0	239	**149**	x458.47	x0.04	**x23.38 _**	9m30s	1h21m	**1h30m**	**22.79s**

Figure 4: Incremental benchmarks results from original publication to latest optimizations.

These results deserve some explanations. First, accuracy drops from paper implementation to the new one as we used the Top-Down to the benefits of computation time which is divided by 2.6 on custom_EA. Paper algorithm was more aggressively replacing sub-AST at the cost of computation time. To our opinion the gain in computation time is worth it. Optimizations **Mul** and **Concat** does not improve simplification but as AST are reduced to a more canonic representation processing them takes less time. This is especially true for `VR-EA` where **Concat** divide by two synthesis time. Similarly, **Mul** almost divide by two synthesis time for `EA-ED` reducing from 2h19m to 1h34m. One may notice that these two simplifications make the simplification potency to slightly drop. The reason is that synthesis was simplifying such unoptimized expressions while after plugging them in symbolic execution they come out already simplified from Triton. Consequently synthesis does not simplify any further. The switch from pickle tables loaded in memory to Level-DB does bring a significant improvement of the synthesis time *(which is surprising as we could expect objects in RAM would be accessed faster)*. For `VR-EA` synthesis time is divided by two. Then as expect a table of 370M entries brings an improvement on the simplification *(but not with a huge margin)*. The average size of synthesized expression drops on all our benchmarks and full synthesis improves for some of them.

VI. Implementation in QSynthesis

The implementation made in Python called QSynthesis is also integrated with your in-house time-travel debugger called QtraceDB described in a previous blogpost[31]. Thus it can equally work with or without trace support as long as instruction can be symbolically executed by Triton. The input of the synthesizer is basically a Triton AST regardless of where it comes from.

QSynthesis is now open-source and available on Github.

Reassembly

The ultimate goal of deobfuscation is being able to regenerate a runnable unobfuscated program. To this end we need to regenerate assembly from deobfuscated triton ASTs. Thankfully, Arybo[32] support converting triton ASTs to its internal rep-

[31] https://blog.quarkslab.com/exploring-execution-trace-analysis.html
[32] https://github.com/quarkslab/arybo

resentation which itself can be translated to LLVM-IR. At the top of it, it provides utility functions to reassemble LLVM-IR to assembly. Under the hood it is using the `llvmlite`[33] Python binding. As such, Qsynthesis uses Arybo as a dependency for reassembly. There are few limitations e.g.: only expression involving register can be synthesized (namely memory can't be used).

The example below shows an end-to-end simple example taken from YANSOllvm where we synthesize a function executed symbolically (without trace), back to reassembly and then patched back in the ELF using LIEF[34].

The video is available: here.

IDA Integration

Qsynthesis has also been integrated in IDA, to take benefit of all its features. It has been integrated in Qtrace-IDA our Time-Travel-Debugger to visualize traces, but it has also been designed to work as a standalone plugin. In the latter case, it does not take advantage of any trace information. Solely the later has been published as our dynamic tracing framework is not open-source[35]. That plugin allows having a fine grain control on the synthesis as it enables synthesizing a given operand just by selecting it. It is also possible to visualize an expression dependencies as an AST tree using IDA graph features. We also can directly reassemble synthesized expressions back in the binary and having a direct overview of the result. The video below shows a simple example where we reassemble a synthesized function into a fresh one.

The video is available: here.

VII. Real-World Examples

YANSOLLVM

While all benchmarks were made on Tigress we decided to focus on YANSOllvm[36] which has been released last year, and is another derivative of Obfuscated LLVM. It includes a bunch of protection and especially `ObfuscateConstant` and `VM` that makes

[33] https://github.com/numba/llvmlite
[34] https://lief.quarkslab.com/
[35] Hopefully it will be at some point.
[36] https://github.com/emc2314/YANSOllvm

uses of MBAs to replace obfuscate atomic operations like (+, -, ˆ). For instance `ObfuscateConstant` replaces zeroes by:

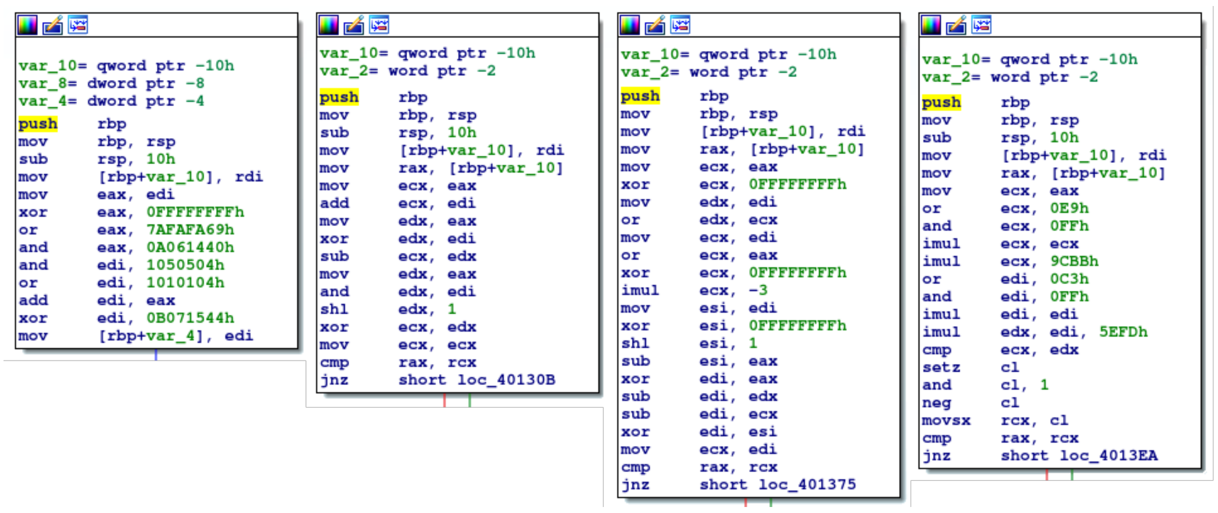
Obfuscated zero constant
((~x | 0x7AFAFA69) & 0xA061440) + ((x & 0x1050504) | 0x1010104) == 185013572
p1*(x|any)**2 != p2*(y|any)**2
x + y = xˆy + 2*(x&y)
x ˆ y = (x|~y) - 3*(~(x|y)) + 2*(~x) - y

Similarly `VM` replaces basic operations by the obfuscated ones:

Operator	Obfuscated expression
x + y	(x|~y) + (~x&y) - (~(x&y)) + (x|y)
x - y	x + ~y + 1
x « y	same
x >a y	same
x >l y	same
x & y	-(~(x&y)) + (~x|y) + (x&~y)
x | y	(xˆy) + y - (~x&y)
x ˆ y	x + y - ((x&y)«1)

The first obfuscation `ObfuscateConstant`[37], transform zeroes with MBAs but also apply constant splitting which value is determined with an arithmetic operation on two pseudo-random values. This later transformation is very effective in masking constants used for instance in the first scheme. As shown below without splitting schemes are rather straightforward to recognize:

[37] https://github.com/emc2314/YANSOllvm/blob/master/lib/Transforms/Obfuscate/ObfuscateConstant.cpp



```
var_10= qword ptr -10h
var_8= dword ptr -8
var_4= dword ptr -4

push    rbp
mov     rbp, rsp
sub     rsp, 10h
mov     [rbp+var_10], rdi
mov     eax, edi
xor     eax, 0FFFFFFFFh
or      eax, 7AFAFA69h
and     eax, 0A061440h
and     edi, 1050504h
or      edi, 1010104h
add     edi, eax
xor     edi, 0B071544h
mov     [rbp+var_4], edi
```

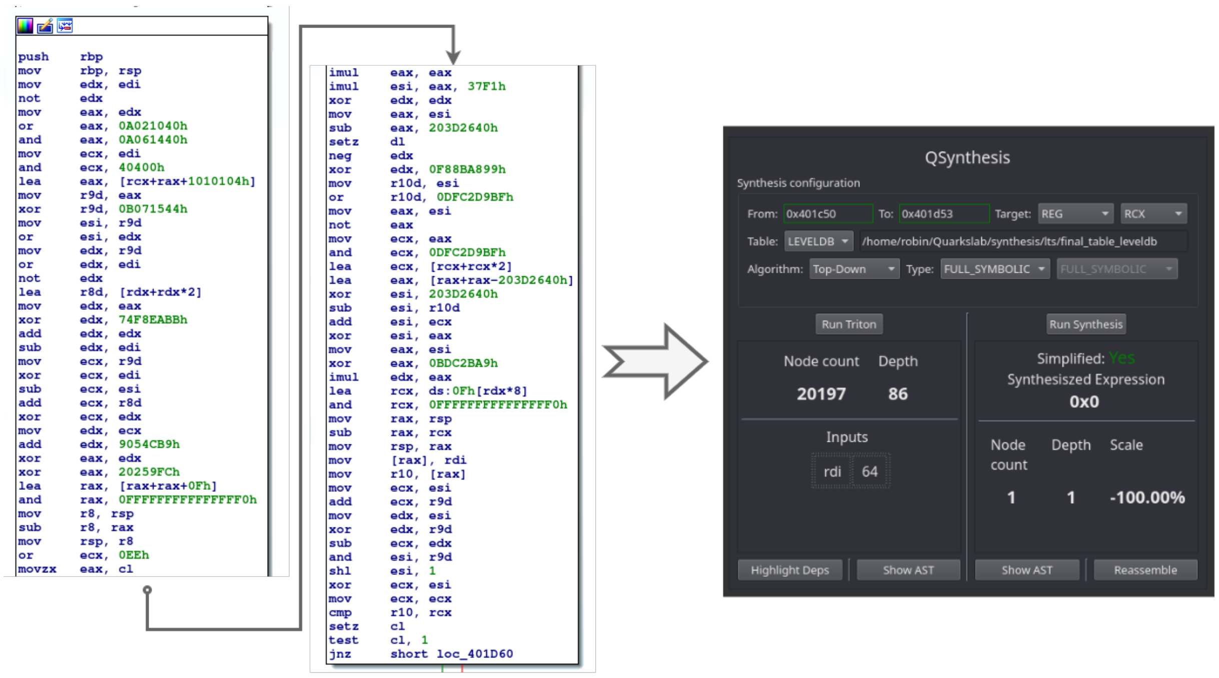
```
var_10= qword ptr -10h
var_2= word ptr -2

push    rbp
mov     rbp, rsp
sub     rsp, 10h
mov     [rbp+var_10], rdi
mov     rax, [rbp+var_10]
mov     ecx, eax
add     ecx, edi
mov     edx, eax
xor     edx, edi
sub     ecx, edx
mov     edx, eax
and     edx, edi
shl     edx, 1
xor     ecx, edx
mov     ecx, ecx
cmp     rax, rcx
jnz     short loc_40130B
```

```
var_10= qword ptr -10h
var_2= word ptr -2

push    rbp
mov     rbp, rsp
mov     [rbp+var_10], rdi
mov     rax, [rbp+var_10]
mov     ecx, eax
xor     ecx, 0FFFFFFFFh
mov     edx, edi
or      edx, ecx
mov     ecx, edi
or      ecx, eax
xor     ecx, 0FFFFFFFFh
imul    ecx, -3
mov     esi, edi
xor     esi, 0FFFFFFFFh
shl     esi, 1
sub     esi, eax
xor     edi, eax
sub     edi, edx
sub     edi, ecx
xor     edi, esi
mov     ecx, edi
cmp     rax, rcx
jnz     short loc_401375
```

```
var_10= qword ptr -10h
var_2= word ptr -2

push    rbp
mov     rbp, rsp
sub     rsp, 10h
mov     [rbp+var_10], rdi
mov     rax, [rbp+var_10]
mov     ecx, eax
or      ecx, 0E9h
and     ecx, 0FFh
imul    ecx, ecx
imul    ecx, 9CBBh
or      edi, 0C3h
and     edi, 0FFh
imul    edi, edi
imul    edx, edi, 5EFDh
cmp     ecx, edx
setz    cl
and     cl, 1
neg     cl
movsx   rcx, cl
cmp     rax, rcx
jnz     short loc_4013EA
```

Figure 5: Main MBAs used by YANSOLLVM

With constants splitted *(which is systematically used)* MBAs are less recognizable but we still break all the MBA easily as the semantic does not change and it is still a zero constant. In this context, synthesis is more generic and allows synthesizing complex expressions but obfuscated constants can also be broken with symbolic execution. In this latter case the principle is finding its value v by evaluation and determining wether or not other values are possible with a constraint of the form $expr \land \not{v}$.



```
push    rbp
mov     rbp, rsp
mov     edx, edi
not     edx
mov     eax, edx
or      eax, 0A021040h
and     eax, 0A061440h
mov     ecx, edi
and     ecx, 40400h
lea     eax, [rcx+rax+1010104h]
mov     r9d, eax
xor     r9d, 0B071544h
mov     esi, r9d
or      esi, edx
mov     edx, r9d
or      edx, edi
not     edx
lea     r8d, [rdx+rdx*2]
mov     edx, eax
xor     edx, 74F8EABBh
add     edx, edx
sub     edx, edi
mov     ecx, r9d
xor     ecx, edi
sub     ecx, esi
add     ecx, r8d
xor     ecx, edx
mov     edx, ecx
add     edx, 9054CB9h
xor     eax, edx
xor     eax, 20259FCh
lea     rax, [rax+rax+0Fh]
and     rax, 0FFFFFFFFFFFFFFF0h
mov     r8, rsp
sub     r8, rax
mov     rsp, r8
or      ecx, 0EEh
movzx   eax, cl
```

```
imul    eax, eax
imul    esi, eax, 37F1h
xor     edx, edx
mov     eax, esi
sub     eax, 203D2640h
setz    dl
neg     edx
xor     edx, 0F88BA899h
mov     r10d, esi
or      r10d, 0DFC2D9BFh
mov     eax, esi
not     eax
mov     ecx, eax
and     ecx, 0DFC2D9BFh
lea     ecx, [rcx+rcx*2]
lea     eax, [rax+rax-203D2640h]
xor     esi, 203D2640h
sub     esi, r10d
add     esi, ecx
xor     esi, eax
mov     eax, esi
xor     eax, 0BDC2BA9h
imul    edx, eax
lea     rcx, ds:0Fh[rdx*8]
and     rcx, 0FFFFFFFFFFFFFFF0h
mov     rax, rsp
sub     rax, rcx
mov     rsp, rax
mov     [rax], rdi
mov     r10, [rax]
mov     ecx, esi
add     ecx, r9d
mov     edx, esi
xor     edx, r9d
sub     ecx, edx
and     esi, r9d
shl     esi, 1
xor     ecx, esi
mov     ecx, ecx
cmp     r10, rcx
setz    cl
test    cl, 1
jnz     short loc_401D60
```

QSynthesis

Synthesis configuration

From: 0x401c50 To: 0x401d53 Target: REG RCX

Table: LEVELDB /home/robin/Quarkslab/synthesis/lts/final_table_leveldb

Algorithm: Top-Down Type: FULL_SYMBOLIC FULL_SYMBOLIC

Node count	Depth
20197	86

Inputs

rdi	64

Simplified: Yes

Synthesiszed Expression

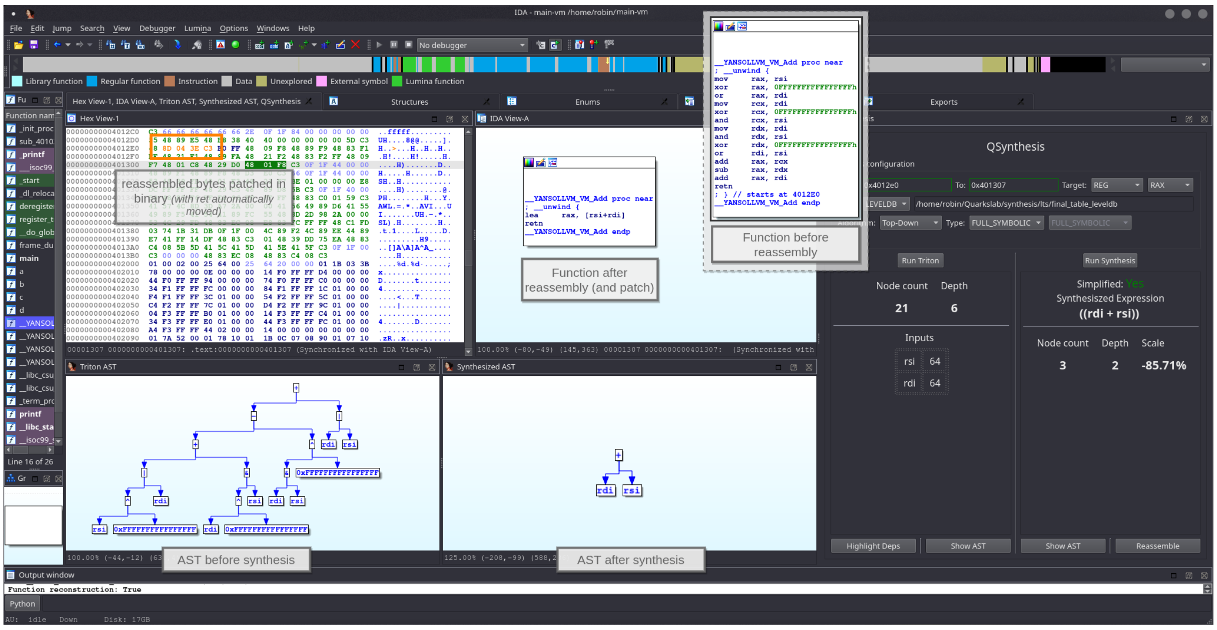
0x0

Node count	Depth	Scale
1	1	-100.00%

Figure 6: Obfuscated constant code and the result in QSynthesis

The VM transformation pass works by opportunistically replacing arithmetic operations mentioned here above by a call to a function doing the operation. The dispatching method is thus systematically a call, no switch, no virtual program counter (VPC), no bytecode per se. In that sense the VM is rather simple in comparison to the possibilities offered by Tigress [38] or other obfuscators. Still, VM obfuscation breaks down complex operations into simple units performing simple operations. We successfully synthesize VM handlers obfuscated with MBAs into their true original operation. The image shown below shows the reassembled and patch version of the ADD handler of a test program.

[38]https://tigress.wtf/virtualize.html


reassembled bytes patched in binary (with ret automatically moved)
Function after reassembly (and patch)
Function before reassembly
AST before synthesis
AST after synthesis
QSynthesis
Run Triton
Run Synthesis
Highlight Deps
Show AST
Reassemble

```
__YANSOLLVM_VM_Add proc near
; __unwind {
mov     rax, rsi
xor     rax, 0FFFFFFFFFFFFFFFFh
or      rax, rdi
mov     rcx, rdi
xor     rcx, 0FFFFFFFFFFFFFFFFh
and     rcx, rsi
mov     rdx, rdi
and     rdx, rsi
xor     rdx, 0FFFFFFFFFFFFFFFFh
or      rdi, rsi
add     rax, rcx
sub     rax, rdx
add     rax, rdi
retn
; } // starts at 4012E0
__YANSOLLVM_VM_Add endp
```

```
__YANSOLLVM_VM_Add proc near
; __unwind {
lea     rax, [rsi+rdi]
retn
__YANSOLLVM_VM_Add endp
```

```
Node count  Depth
21          6
Inputs
rsi  64
rdi  64

Simplified: Yes
Synthesized Expression
((rdi + rsi))
Node count  Depth  Scale
3           2      -85.71%

Function reconstruction: True
```

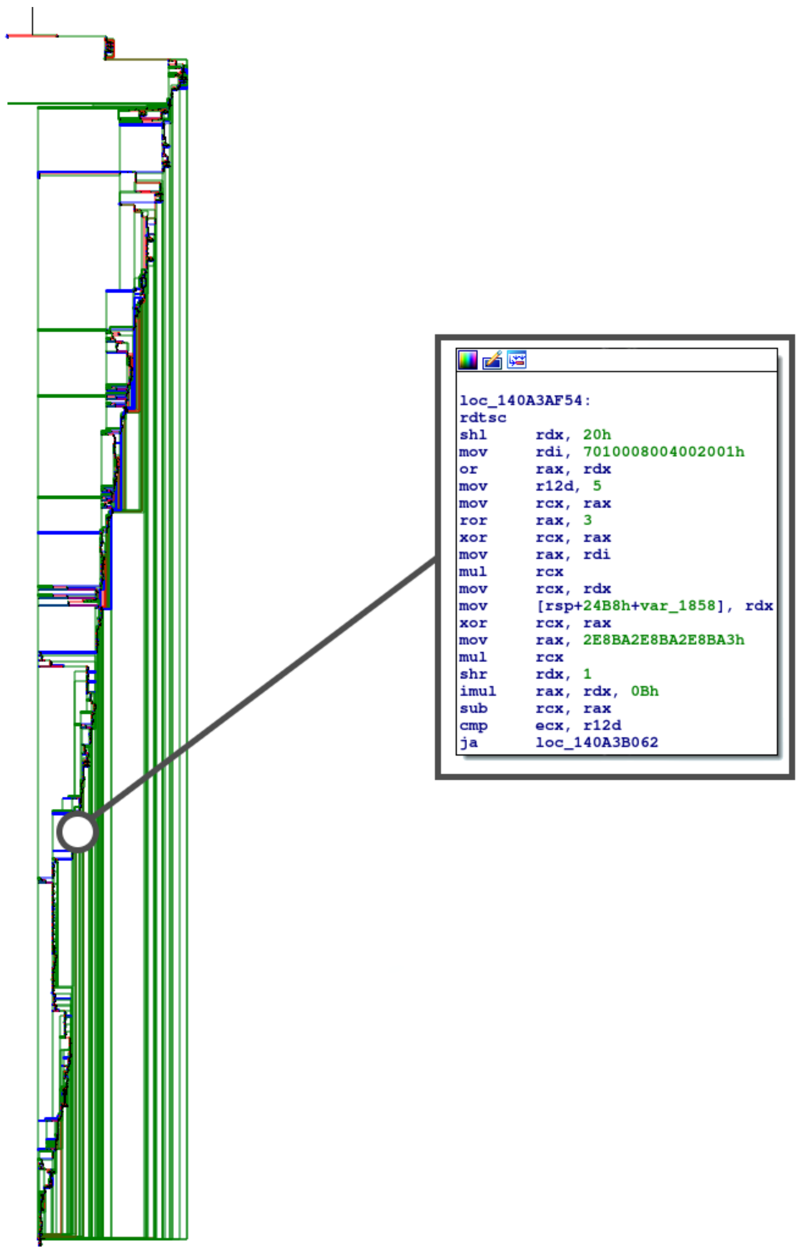
Figure 7: Qsynthesis IDA integration example with full reassembly.

While these two obfuscations do not hold against synthesis, YANSOllvm is a very decent obfuscator and all transformations combined together makes it rather difficult to identify locations where to perform the synthesis.

Warbird Framework

Microsoft is using a framework called Warbird to obfuscate various components and notably KPP. Alex Ionescu discussed this obfuscator at Ekoparty[39], and Airbus Seclab studied its VM engine[40]. A colleague pinpointed me the `PatchGuardInit` function of the Windows kernel for being somewhat obfuscated. The function is rather big with 5231 basic blocks where many of them seem to contain overly complex arithmetic operations on pseudo-random values generated by `rtdsc`. We do not know the exact obfuscation nature nor the exact purpose of these blocks but operations seem at least diversified and purposely expanded.

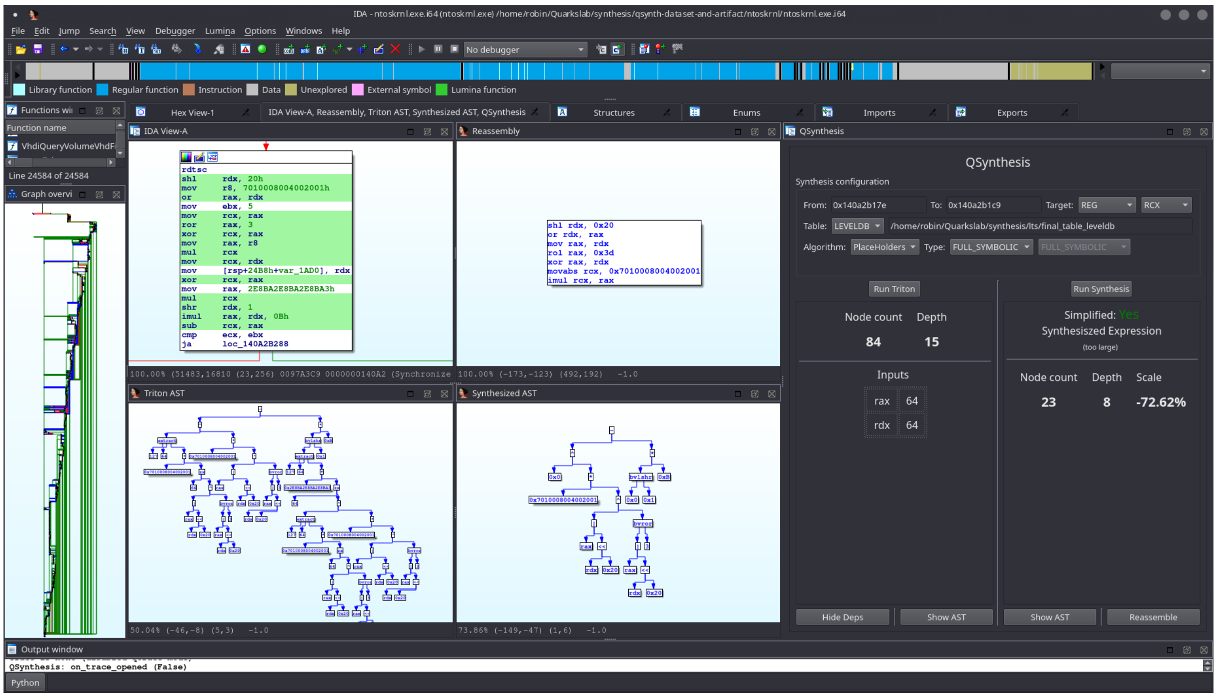
[39] https://youtu.be/gu_i6LYuePg?t=987
[40] https://github.com/airbus-seclab/warbirdvm#id11



```
loc_140A3AF54:
rdtsc
shl     rdx, 20h
mov     rdi, 7010008004002001h
or      rax, rdx
mov     r12d, 5
mov     rcx, rax
ror     rax, 3
xor     rcx, rax
mov     rax, rdi
mul     rcx
mov     rcx, rdx
mov     [rsp+24B8h+var_1858], rdx
xor     rcx, rax
mov     rax, 2E8BA2E8BA2E8BA3h
mul     rcx
shr     rdx, 1
imul    rax, rdx, 0Bh
sub     rcx, rax
cmp     ecx, r12d
ja      loc_140A3B062
```

Figure 8: Basic block sample in `PatchGuardInit`.

Synthesis somewhat works on these basic blocks and simplifies the whole function. In addition to anti-debugging schemes other obfuscations seems to have been applied as it also seems unrolled. We did not investigated further. Arithmetic diversification seem's rather easy to address in comparison to the VM protection that Warbird also provides. Still, it shows the efficiency of synthesis to clean a bit the instructions.



```
rdtsc
shl     rdx, 20h
mov     r8, 7010008004002001h
or      rax, rdx
mov     ebx, 5
mov     rcx, rax
ror     rax, 3
xor     rcx, rax
mov     rax, r8
mul     rcx
mov     rcx, rdx
mov     [rsp+24B8h+var_1AD0], rdx
xor     rcx, rax
mov     rax, 2E8BA2E8BA2E8BA3h
mul     rcx
shr     rdx, 1
imul    rax, rdx, 0Bh
sub     rcx, rax
cmp     ecx, ebx
ja      loc_140A2B288
```

```
shl rdx, 0x20
or rdx, rax
mov rax, rdx
rol rax, 0x3d
xor rax, rdx
movabs rcx, 0x7010008004002001
imul rcx, rax
```

QSynthesis

Synthesis configuration

From: 0x140a2b17e To: 0x140a2b1c9 Target: REG RCX
Table: LEVELDB /home/robin/Quarkslab/synthesis/lts/final_table_leveldb
Algorithm: PlaceHolders Type: FULL_SYMBOLIC FULL_SYMBOLIC

Node count	Depth
84	15

Inputs

rax	64
rdx	64

Simplified: Yes
Synthesiszed Expression
(too large)

Node count	Depth	Scale
23	8	-72.62%

```
QSynthesis: on_trace_opened (False)
```

Figure 9: Synthesis result on the basic block of `PatchGuardInit`.

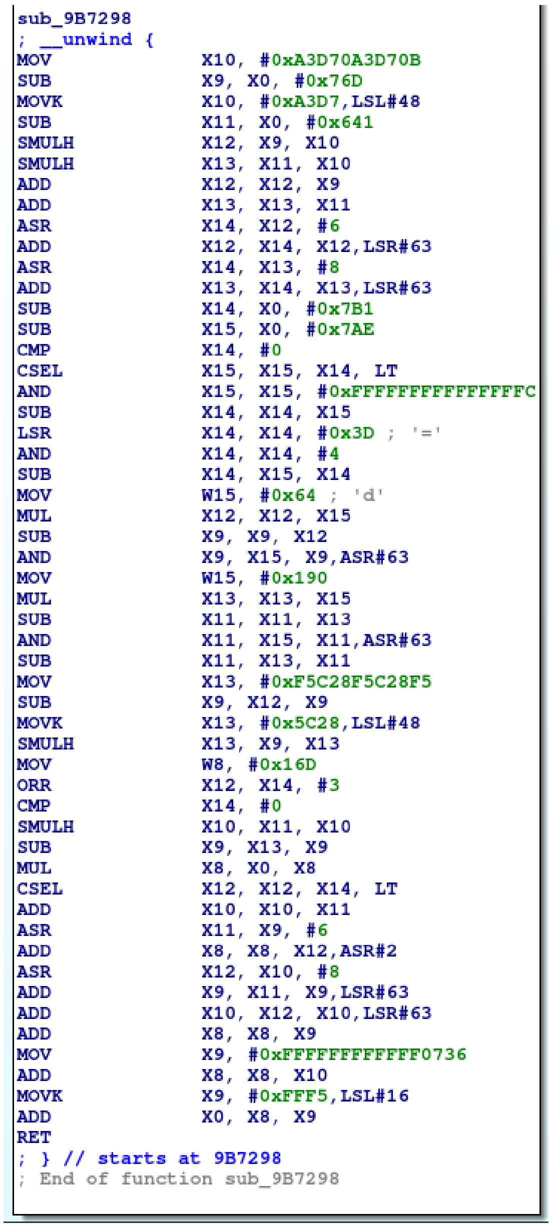
One can observe that the reassembly further simplified the synthesized AST by removing parts which were constant. When reassembling we thus also takes advantage of compiler optimizations to simplify even more the expressions. Note that we could have done it by applying a post-processing pass on the AST.

Messaging Application

Messaging application are well known for being obfuscated. Previous research has been made on one of them by another researcher[41],[42]. We looked at the messaging application and especially `libclient.so` in version 10.85.5.0 of the app in ARM64. Previous works have shown that it makes use of dataflow obfuscation and more specifically MBAs. One of them is an obfuscated version of `0x99DB8D4C50945260 ^ bss_val1 & ~0x3` containing 165 nodes. The synthesis algorithm managed to divide its size by two (~75 nodes) but not to recover the original expression. The reason is the presence of the constant which hinders the synthesis. Dedicated approaches like Arybo managed to solve it[43]. By crawling the code, we found similar structures

[41] https://hot3eed.github.io/2020/06/18/snap_p1_obfuscations.html
[42] https://hot3eed.github.io/2020/06/22/snap_p2_deobfuscation.html
[43] https://pastebin.com/xBvSF05J

which if not MBA are at least purposely complex. On the example given in the figure below, we were able to reduce the expression of `X0` on the return address by 24%. Yet, we did not recovered the original expression if any.

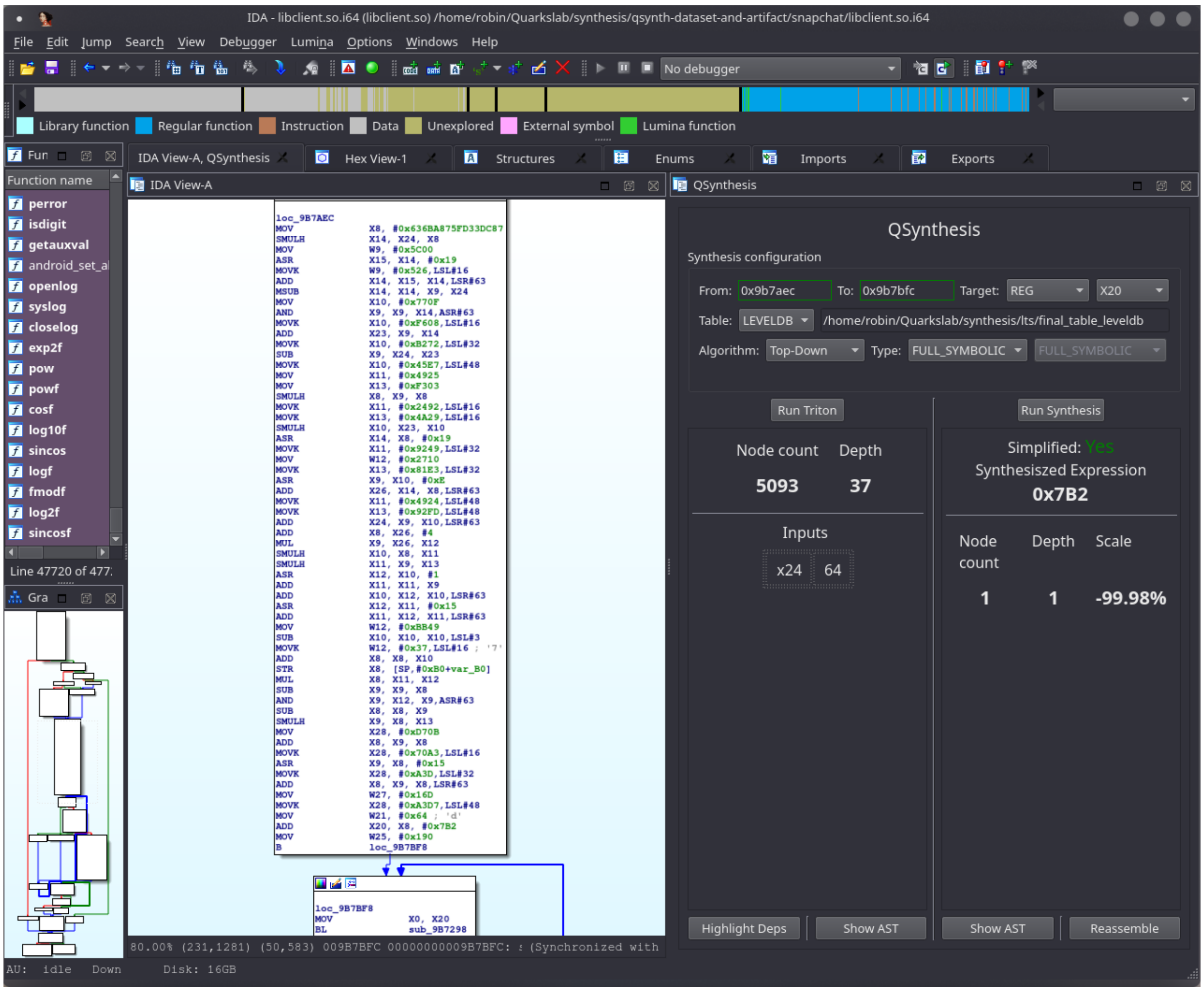


```
sub_9B7298
; __unwind {
MOV                X10, #0xA3D70A3D70B
SUB                X9, X0, #0x76D
MOVK               X10, #0xA3D7,LSL#48
SUB                X11, X0, #0x641
SMULH              X12, X9, X10
SMULH              X13, X11, X10
ADD                X12, X12, X9
ADD                X13, X13, X11
ASR                X14, X12, #6
ADD                X12, X14, X12,LSR#63
ASR                X14, X13, #8
ADD                X13, X14, X13,LSR#63
SUB                X14, X0, #0x7B1
SUB                X15, X0, #0x7AE
CMP                X14, #0
CSEL               X15, X15, X14, LT
AND                X15, X15, #0xFFFFFFFFFFFFFFFC
SUB                X14, X14, X15
LSR                X14, X14, #0x3D ; '='
AND                X14, X14, #4
SUB                X14, X15, X14
MOV                W15, #0x64 ; 'd'
MUL                X12, X12, X15
SUB                X9, X9, X12
AND                X9, X15, X9,ASR#63
MOV                W15, #0x190
MUL                X13, X13, X15
SUB                X11, X11, X13
AND                X11, X15, X11,ASR#63
SUB                X11, X13, X11
MOV                X13, #0xF5C28F5C28F5
SUB                X9, X12, X9
MOVK               X13, #0x5C28,LSL#48
SMULH              X13, X9, X13
MOV                W8, #0x16D
ORR                X12, X14, #3
CMP                X14, #0
SMULH              X10, X11, X10
SUB                X9, X13, X9
MUL                X8, X0, X8
CSEL               X12, X12, X14, LT
ADD                X10, X10, X11
ASR                X11, X9, #6
ADD                X8, X8, X12,ASR#2
ASR                X12, X10, #8
ADD                X9, X11, X9,LSR#63
ADD                X10, X12, X10,LSR#63
ADD                X8, X8, X9
MOV                X9, #0xFFFFFFFFFFFF0736
ADD                X8, X8, X10
MOVK               X9, #0xFFF5,LSL#16
ADD                X0, X8, X9
RET
; } // starts at 9B7298
; End of function sub_9B7298
```

Figure 10: MBA example

As another example, the basic block shown below, compute the register `X0` given as argument to the function `sub_9B7298` at the bottom. The algorithm synthesized it as the constant `0x7b2` which appear to be true as the register `X8` at instruction `MOV`

X20, X8, #0x7B2 gets synthesized to 0. Thus whole computation seems essentially useless. On this use-case synthesis is very effective. Yet, obfuscation is greatly applied on this library, deobfuscating the whole app would require addressing the other kinds of obfuscations and the various blends performed. Doing so would require far more work.


IDA - libclient.so.i64 (libclient.so) /home/robin/Quarkslab/synthesis/qsynth-dataset-and-artifact/snapchat/libclient.so.i64

```
loc_9B7AEC
MOV             X8, #0x636BA875FD33DC87
SMULH           X14, X24, X8
MOV             W9, #0x5C00
ASR             X15, X14, #0x19
MOVK            W9, #0x526,LSL#16
ADD             X14, X15, X14,LSR#63
MSUB            X14, X14, X9, X24
MOV             X10, #0x770F
AND             X9, X9, X14,ASR#63
MOVK            X10, #0xF608,LSL#16
ADD             X23, X9, X14
MOVK            X10, #0xB272,LSL#32
SUB             X9, X24, X23
MOVK            X10, #0x45E7,LSL#48
MOV             X11, #0x4925
MOV             X13, #0xF303
SMULH           X8, X9, X8
MOVK            X11, #0x2492,LSL#16
MOVK            X13, #0x4A29,LSL#16
SMULH           X10, X23, X10
ASR             X14, X8, #0x19
MOVK            X11, #0x9249,LSL#32
MOV             W12, #0x2710
MOVK            X13, #0x81E3,LSL#32
ASR             X9, X10, #0xE
ADD             X26, X14, X8,LSR#63
MOVK            X11, #0x4924,LSL#48
MOVK            X13, #0x92FD,LSL#48
ADD             X24, X9, X10,LSR#63
ADD             X8, X26, #4
MUL             X9, X26, X12
SMULH           X10, X8, X11
SMULH           X11, X9, X13
ASR             X12, X10, #1
ADD             X11, X11, X9
ADD             X10, X12, X10,LSR#63
ASR             X12, X11, #0x15
ADD             X11, X12, X11,LSR#63
MOV             W12, #0xBB49
SUB             X10, X10, X10,LSL#3
MOVK            W12, #0x37,LSL#16 ; '7'
ADD             X8, X8, X10
STR             X8, [SP,#0xB0+var_B0]
MUL             X8, X11, X12
SUB             X9, X9, X8
AND             X9, X12, X9,ASR#63
SUB             X8, X8, X9
SMULH           X9, X8, X13
MOV             X28, #0xD70B
ADD             X8, X9, X8
MOVK            X28, #0x70A3,LSL#16
ASR             X9, X8, #0x15
MOVK            X28, #0xA3D,LSL#32
ADD             X8, X9, X8,LSR#63
MOV             W27, #0x16D
MOVK            X28, #0xA3D7,LSL#48
MOV             W21, #0x64 ; 'd'
ADD             X20, X8, #0x7B2
MOV             W25, #0x190
B               loc_9B7BF8

loc_9B7BF8
MOV             X0, X20
BL              sub_9B7298
```

```
80.00% (231,1281) (50,583) 009B7BFC 00000000009B7BFC: : (Synchronized with
AU:  idle   Down      Disk: 16GB
```

QSynthesis

Synthesis configuration

From: 0x9b7aec To: 0x9b7bfc Target: REG X20
Table: LEVELDB /home/robin/Quarkslab/synthesis/lts/final_table_leveldb
Algorithm: Top-Down Type: FULL_SYMBOLIC FULL_SYMBOLIC

Node count	Depth
5093	37

Inputs: x24 64

Simplified: Yes
Synthesiszed Expression
0x7B2

Node count	Depth	Scale
1	1	-99.98%

Figure 11: QSynthesis synthesized X20 which is the constant value 0x7B2.

VIII. From Deobfuscation to Software Diversification

So far this whole research was turned toward reducing the size of expressions manipulated to generate optimized instructions. The goal of the synthesis oracle *fitness function* is returning a new expression solely if it was smaller than the one in input. However, this function can represent any objective. As such, we can turn instantly the synthesizer into an obfuscating program synthesizer by flipping the comparison being made by the oracle. We can thus recursively and infinitely diversify programs. The exhaustiveness of tables, entirely determines the reversibility of the process.

IX. Conclusion

This whitepaper intended to provide a quick introduction of program synthesis, how we do use it at Quarkslab, and how we are improving it incrementally to make it more and more efficient. We explained in greater details *(than the paper)* our "greybox synthesizer" inner working, its search strategies, lookup tables working and various extensions like linearization, static way to work or else the learning mechanism. We have also shown efforts made to make it scale and we have shown how it performs on real targets. Yet, the approach is not perfect, there are many roadblocks remaining, and we still have various experiments to perform to continue improving it. Among them, constants is a huge one that we want to address by coupling synthesis with other approaches and by circumventing techniques[44]. Also, at the moment all input variables are of the same size and output is also of the same size. We would like to synthesize expressions with heterogeneous sizes.

Still, we show through this implementation and integration one of the first implementations that recovers deobfuscated instructions reassembled back in the code as an end-to-end process. It has now gained a level a maturity allowing us to use it in day-to-day reverse tasks (hopefully facilitating Quarkslab mates life ;)).

[44] https://blog.regehr.org/archives/1636