QSynth - A Program Synthesis approach for Binary Code Deobfuscation

Binary Analysis Workshop - NDSS

Robin David <rdavid@quarkslab.com>
Luigi Coniglio <luigi.coniglio@studenti.unitn.it>
Mariano Ceccato <mariano.ceccato@univr.it>

February 23th, 2020 - San Diego, California
Talk Outline

Context:
- Need to address highly obfuscated binaries
- Few approaches address data obfuscation

Goal: deobfuscating expression *(obfuscated with data transformations)*
**Talk Outline**

**Context:**
- Need to address highly obfuscated binaries
- Few approaches address data obfuscation

**Goal:** deobfuscating expression *(obfuscated with data transformations)*

**Takeway**

We provide a synthesis approach addressing various obfuscations and that **supersede** the state-of-the-art in both **speed** and **accuracy**
Table of Contents

Background
  Software obfuscation
  Deobfuscation techniques

Our Synthesis Approach
  Goal & Contributions
  Approach steps

Experimental Benchmarks
  Experimental Setup
  Benchmarks

Conclusion
Obfuscation types

Control-Flow Obfuscation

Hiding the **logic** and algorithm of the program

- Virtualization, Opaque predicates,
- CFG-flattening, Split, Merge, Packing,
- Implicit Flow, MBA, Loop-Unrolling...

Example
Obfuscation types

**Control-Flow Obfuscation**

Hiding the **logic** and algorithm of the program

- Virtualization, Opaque predicates, CFG-flattening, Split, Merge, Packing, Implicit Flow, MBA, Loop-Unrolling...

**Data-Flow Obfuscation**

Hiding data, constants, strings, APIs, keys etc.

- Data encoding, MBA, Arithmetic Encoding, Whitebox, Array Split, Fold and Merge, Variable Splitting...

**Example**

\[ a + b \implies (((((a \land \neg b) + b) \ll 1) \land \neg((a \lor b) - (a \land b))) \ll 1) - (((a \land \neg b) + b) \ll 1) \oplus ((a \lor b) - (a \land b))) \]
Obfuscation types

<table>
<thead>
<tr>
<th>Control-Flow Obfuscation</th>
<th>Data-Flow Obfuscation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hiding the <strong>logic</strong> and algorithm of the program</td>
<td>Hiding data, constants, strings, APIs, keys etc.</td>
</tr>
<tr>
<td>Virtualization, Opaque predicates, CFG-flattening, Split, Merge, Packing, Implicit Flow, MBA, Loop-Unrolling...</td>
<td>Data encoding, MBA, Arithmetic Encoding, Whitebox, Array Split, Fold and Merge, Variable Splitting...</td>
</tr>
</tbody>
</table>

**Example**

\[
\begin{align*}
\text{Example} \\
\quad a + b & \implies (\neg(((((a \land \neg b) + b) << 1) \land \neg((a \lor b) - (a \land b))) << 1) - (((((a \land \neg b) + b) << 1) \oplus ((a \lor b) - (a \land b))))
\end{align*}
\]

**Problem:** Reverting an obfuscating transformation is hard.
Deobfuscation

Let’s focus on two deobfuscation techniques:

Dynamic Symbolic Execution

Program Synthesis
Symbolic Execution

Definition

Mean of executing a program using **symbolic values** *(logical symbols)* rather than real values *(bitvectors)* in order to obtain an **in-out relationship of a path**

Source Code (C)

```c
int f(int a, int b) {
    if (a < 10) {
        if (a > b) {
            printf("OK");
        }
    }
}
```

Formula:

`a < 10 ∧ a > b`

Solution:

`a=5, b=1`

(using SMT solvers)
Symbolic Execution

Definition

Mean of executing a program using **symbolic values** (**logical symbols**) rather than real values (**bitvectors**) in order to obtain an **in-out relationship of a path**

**Source Code (C)**

```c
int f(int a, int b) {
    if (a < 10) {
        if (a > b) {
            printf("OK");
        }
    }
}
```

**Formula:**

\[ a < 10 \land a > b \]

**Solution:**

\[ a=5, b=1 \]

(Using SMT solvers)

Dynamic Symbolic Execution (a.k.a. concolic)

- **Properties:** work on **dynamic paths** and use runtime values
- **Advantages:** path sure to be **feasible** and thwart various obfuscations
Symbolic Execution: Example

⇒ In this context used to extract symbolic expressions (e.g. b)

```c
1  if (a > 0){
2      b += (a | -1) - 1;
3      b -= ((~ a) & -1);
4  } else {
5      b += (a | -3) + 1;
6  }
7  b -= 1 + ((b * (b + 1)) % 2);
```
Symbolic Execution: Example

⇒ In this context used to extract symbolic expressions (e.g. b)

```java
1 if (a > 0){
  2       b += (a | -1) - 1;
  3       b -= ((~ a) & -1);
4 } else {
5       b += (a | -3) + 1;
6     }
7 b -= 1 + ((b * (b + 1)) % 2);
```

Symbolic State

\[ \phi_b = b \]
Symbolic Execution: Example

⇒ In this context used to extract symbolic expressions (e.g. \( b \))

```plaintext
1 if (a > 0) {
2     b += (a | -1) - 1;
3     b -= ((~ a) & -1);
4 } else {
5     b += (a | -3) + 1;
6 }
7 b -= 1 + ((b * (b + 1)) % 2);
```

**Symbolic State**

\[
\begin{align*}
\phi_b &= b \\
\phi_b &= b + (a | -1) - 1
\end{align*}
\]
Symbolic Execution: Example

⇒ In this context used to extract symbolic expressions (e.g. b)

```
1 if (a > 0){
2     b += (a | -1) - 1;
3     b -= ((~ a) & -1);
4 } else {
5     b += (a | -3) + 1;
6 }
7 b -= 1 + ((b * (b + 1)) % 2);
```

### Symbolic State

\[
\begin{align*}
\phi_b &= b \\
\phi_b &= b + (a \mid -1) - 1 \\
\phi_b &= b + (a \mid -1) - 1 - ((\sim a) \& -1)
\end{align*}
\]
Symbolic Execution: Example

In this context used to extract symbolic expressions (e.g. b)

```c
if (a > 0){
    b += (a | -1) - 1;
    b -= ((~ a) & -1);
} else {
    b += (a | -3) + 1;
}
b -= 1 + ((b * (b + 1)) % 2);
```

**Symbolic State**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \phi_b = b )</td>
<td>( \phi_b = b )</td>
</tr>
<tr>
<td>( \phi_b = b + (a</td>
<td>-1) - 1 )</td>
</tr>
<tr>
<td>( \phi_b = b + (a</td>
<td>-1) - 1 - ((\sim a) &amp; -1) - 1 + (((b + (a</td>
</tr>
</tbody>
</table>

**Question:** How to simplify the \( \phi_b \) expression?

(Knowing that the quality of the result depends on the syntactic complexity of the obfuscated expression)
Program Synthesis

Definition

Program synthesis consists in automatically deriving a program from:

- a high-level specification *(typically its I/O behaviour)*
- additional constraints:
  - Compilation: a faster program
  - Deobfuscation: a smaller or more readable program

Example

Obfuscated Program

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 2</td>
<td>3</td>
</tr>
<tr>
<td>2, 2</td>
<td>4</td>
</tr>
<tr>
<td>2, 3</td>
<td>5</td>
</tr>
</tbody>
</table>

⇒ `a + b`
Program Synthesis

**Definition**

Program synthesis consists in automatically deriving a program from:

- a high-level specification (*typically its I/O behaviour*)
- additional constraints:
  - Compilation: a faster program
  - Deobfuscation: a smaller or more readable program

**Example**

<table>
<thead>
<tr>
<th>Obfuscated Program</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Program Synthesis

Definition

Program synthesis consists in automatically deriving a program from:

▶ a high-level specification (typically its I/O behaviour)
▶ additional constraints:
  ▶ Compilation: a faster program
  ▶ Deobfuscation: a smaller or more readable program

Example

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 2</td>
<td>3</td>
</tr>
</tbody>
</table>

Obfuscated Program
Program Synthesis

Definition

Program synthesis consists in automatically deriving a program from:

- a high-level specification (typically its I/O behaviour)
- additional constraints:
  - Compilation: a faster program
  - Deobfuscation: a smaller or more readable program

Example

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 2</td>
<td>3</td>
</tr>
<tr>
<td>2, 2</td>
<td>4</td>
</tr>
</tbody>
</table>

Obfuscated Program
Program Synthesis

Definition

Program synthesis consists in automatically deriving a program from:

- a high-level specification (*typically its I/O behaviour*)
- additional constraints:
  - Compilation: a faster program
  - Deobfuscation: a smaller or more readable program

Example

**Obfuscated Program**

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 2</td>
<td>3</td>
</tr>
<tr>
<td>2, 2</td>
<td>4</td>
</tr>
<tr>
<td>2, 3</td>
<td>5</td>
</tr>
</tbody>
</table>
Program Synthesis

Definition

Program synthesis consists in automatically deriving a program from:

- a high-level specification *(typically its I/O behaviour)*
- additional constraints:
  - Compilation: a faster program
  - Deobfuscation: a smaller or more readable program

Example

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 2</td>
<td>3</td>
</tr>
<tr>
<td>2, 2</td>
<td>4</td>
</tr>
<tr>
<td>2, 3</td>
<td>5</td>
</tr>
</tbody>
</table>

⇒ \( a + b \)
Program Synthesis

Definition

Program synthesis consists in automatically deriving a program from:

▶ a high-level specification (typically its I/O behaviour)
▶ additional constraints:
  ▶ Compilation: a faster program
  ▶ Deobfuscation: a smaller or more readable program

Example

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 2</td>
<td>3</td>
</tr>
<tr>
<td>2, 2</td>
<td>4</td>
</tr>
<tr>
<td>2, 3</td>
<td>5</td>
</tr>
</tbody>
</table>

⇒ \( a + b \)

Obfuscated Program

Problem

Synthesizing programs (expressions) with complex behaviors is hard.
Table of Contents

Background
  Software obfuscation
  Deobfuscation techniques

Our Synthesis Approach
  Goal & Contributions
  Approach steps

Experimental Benchmarks
  Experimental Setup
  Benchmarks

Conclusion
Key Intuition

**Symbolic Execution**

- Capture full semantic
- Influenced by syntactic complexity
Key Intuition

**Symbolic Execution**
+ Capture full semantic
  - Influenced by syntactic complexity

**Program Synthesis**
+ Only influenced by semantic complexity
  - Black-box $\Rightarrow$ big search space
Key Intuition

**Symbolic Execution**
- Capture full semantic
- Influenced by syntactic complexity

**Program Synthesis**
- Only influenced by semantic complexity
- Black-box $\Rightarrow$ big search space

**Idea**: Using symbolic execution to *reduce the synthesis search space*
Contributions

A synthesis approach using an **Offline Enumerative Search** based on pre-computed lookup tables combined with an Abstract Syntax Tree **simplification algorithm** which **outperform** similar approach of the state-of-the-art (e.g. Syntia)
QSynth: Overview

Obfuscated program

Execution tracing (DBI)

Execution trace

Dynamic Symbolic Execution

Obfuscated expressions

Enumerative Synthesis Oracle

(generated once for all)

Simplification Strategy

(for each sub-expression)

inputs

outputs

equivalent expression

synthesized expressions
QSynth: Overview

**Execution tracing**

**Obfuscated program**

**Dynamic Symbolic Execution**

**Enumerative Synthesis Oracle**

(generated once for all)

**Simplification Strategy**

(for each sub-expression)

**Inputs**

**Outputs**

**Equivalent expression**

**Synthesized expressions**

**Tool:**

QBDI
Dynamic Binary Instrumentation

Using **QBDI**: Quarkslab Dynamic binary Instrumentation *(similar to Pin, DynamoRIO)*

- multi-architecture & platform
- no (direct) thread support

**Qtracer** (a qbditool like Pin “pintools”)

- gather instruction executed with their concrete state *(registers and memory)*
- Data are consolidated in database *(SQLite, PostgresSQL etc.)*

**Original**

```assembly
mov qword [0x000232c0], 8
mov r13, rax
test rax, rax
je 0x42a7                   
xor r8d, r8d          ...
xor edx, edx   ...
xor esi, esi   ...
```

**Instrumented**

```assembly
mov qword [0x000232c0], 8
; Some code ...
mov r13, rax
; Some code ...
test rax, rax
; Some code ...
test rax, rax
; Some code ...
je <patched address>  
xor r8d, r8d          ...
xor edx, edx          ...
xor esi, esi          ...
```

*[https://qbdi.quarkslab.com/](https://qbdi.quarkslab.com/)*
QSynth: Overview

Obfuscated program

Execution tracing (DBI)

Dynamic Symbolic Execution

Enumerative Synthesis Oracle
(generated once for all)

Simplification Strategy
(for each sub-expression)

inputs

outputs

equivalent expression

synthesized expressions

Tool: QBDI
QSynth: Overview

Execution tracing (DBI)

Dynamic Symbolic Execution

Enumerative Synthesis Oracle
(generated once for all)

Simplification Strategy
(for each sub-expression)

Obfuscated program

Execution trace

Obfuscated expressions

Enumerative Synthesis Oracle

Equivalent expression

Outputs

Synthesized expressions

Tool:
TRILLION

Dynamic Binary Analysis
Triton allows computing any **symbolic expression** along the trace by backtracking on data dependencies.

```plaintext
1 if (a > 0){
2     b += a - 1;
3 } else {
4     b += 2;
5 }
6 b -= 1;
```
Triton allows computing any **symbolic expression** along the trace by backtracking on data dependencies.
Triton allows computing any **symbolic expression** along the trace by backtracking on data dependencies.

```c
if (a > 0) {
    b += a - 1;
} else {
    b += 2;
}
b -= 1;
```
Triton allows computing any **symbolic expression** along the trace by backtracking on data dependencies.

\[
\varphi \triangleq (b + (a - 1)) - 1
\]

\(\varphi\) is the associated I/O oracle can be evaluated on different inputs.
QSynth: Overview

Execution tracing (DBI)

Obfuscated program

Dynamic Symbolic Execution

Enumerative Synthesis Oracle (generated once for all)

Simplification Strategy (for each sub-expression)

Tool: TRILION

Execution trace

Obfuscated expressions

inputs

outputs

equivalent expression

synthesized expressions
QSynth: Overview

Execution tracing (DBI)
Obfuscated program
Dynamic Symbolic Execution
Obfuscated expressions
Simplification Strategy (for each sub-expression)
Enumerative Synthesis Oracle (generated once for all)
inputs
outputs
equivalent expression
synthesized expressions
We call Synthesis Primitive any program $\mathcal{SP}$ taking as input parameters a black-box oracle $O_\varphi$ and a set of input parameters to the oracle $\mathcal{I}$, and returning, in case of success, a program $p$, such that for any $i \in \mathcal{I}$ then $p(i) = O_\varphi(i)$.

\[
\mathcal{SP}(O_\varphi, \mathcal{I}) \Rightarrow p \mid \forall i \in \mathcal{I}, p(i) \equiv O_\varphi(i)
\]

\[
\mathcal{SP}(O_\varphi, \mathcal{I}) \Rightarrow \emptyset
\]
Generate a set of programs based on a given grammar: (operators & variables)

\[ a + b, a - b, a + a, b + b, a + a - b, \ldots \]
Offline Enumerative Search (synthesis primitive $SP$)

Generate a set of programs based on a given **grammar**: (operators & variables)

\[ a + b, a - b, a + a, b + b, a + a - b, \ldots \]

and with a set of **inputs**: (pseudo-random)

\[ \text{vector } I = \{(1, 1), (1, 0), (2, 1)\} \]
Offline Enumerative Search (synthesis primitive $SP$)

Generate a set of programs based on a given grammar: (operators & variables)

\[ a + b, a - b, a + a, b + b, a + a - b, \ldots \]

and with a set of inputs: (pseudo-random)

\[ \text{vector } \mathcal{I} = \{(1, 1), (1, 0), (2, 1)\} \]

Evaluate all programs on $\mathcal{I}$ and create the synthesis oracle $SP : \text{outputs} \rightarrow p$
Generate a set of programs based on a given grammar: (operators & variables)

\[ a + b, a - b, a + a, b + b, a + a - b, \ldots \]

and with a set of inputs: (pseudo-random)

vector \( \mathcal{I} = \{(1, 1), (1, 0), (2, 1)\} \)

Evaluate all programs on \( \mathcal{I} \) and create the synthesis oracle \( \mathcal{SP} : \text{outputs} \rightarrow p \)

Example:

<table>
<thead>
<tr>
<th>Outputs</th>
<th>( p )</th>
</tr>
</thead>
<tbody>
<tr>
<td>2, 1, 3</td>
<td>( a + b )</td>
</tr>
</tbody>
</table>
Offline Enumerative Search (synthesis primitive $SP$)

Generate a set of programs based on a given grammar: (operators & variables)

$$a + b, a - b, a + a, b + b, a + a - b, \ldots$$

and with a set of inputs: (pseudo-random)

vector $I = \{(1, 1), (1, 0), (2, 1)\}$

Evaluate all programs on $I$ and create the synthesis oracle $SP : \text{outputs} \rightarrow p$

Example:

<table>
<thead>
<tr>
<th>Outputs</th>
<th>$p$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2, 1, 3</td>
<td>$a + b$</td>
</tr>
<tr>
<td>0, 1, 1</td>
<td>$a - b$</td>
</tr>
</tbody>
</table>
Offline Enumerative Search (synthesis primitive $SP$)

Generate a set of programs based on a given grammar: (operators & variables)

$$a + b, a - b, a + a, b + b, a + a - b, \ldots$$

and with a set of inputs: (pseudo-random)

vector $I = \{(1, 1), (1, 0), (2, 1)\}$

Evaluate all programs on $I$ and create the synthesis oracle $SP : outputs \rightarrow p$

Example:

<table>
<thead>
<tr>
<th>Outputs</th>
<th>$p$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2, 1, 3</td>
<td>$a + b$</td>
</tr>
<tr>
<td>0, 1, 1</td>
<td>$a - b$</td>
</tr>
<tr>
<td>2, 2, 4</td>
<td>$a + a$</td>
</tr>
</tbody>
</table>
Offline Enumerative Search \hspace{10px} (synthesis primitive $S\mathcal{P}$)

Generate a set of programs based on a given **grammar**: (operators & variables)

\[ a + b, a - b, a + a, b + b, a + a - b, \ldots \]

and with a set of **inputs**: (pseudo-random)

vector $I = \{(1, 1), (1, 0), (2, 1)\}$

Evaluate all programs on $I$ and create the synthesis oracle $S\mathcal{P} : \text{outputs} \rightarrow p$

**Example:**

<table>
<thead>
<tr>
<th>Outputs</th>
<th>$p$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2, 1, 3</td>
<td>$a + b$</td>
</tr>
<tr>
<td>0, 1, 1</td>
<td>$a - b$</td>
</tr>
<tr>
<td>2, 2, 4</td>
<td>$a + a$</td>
</tr>
<tr>
<td>\ldots</td>
<td>\ldots</td>
</tr>
</tbody>
</table>
Generate a set of programs based on a given **grammar**: (operators & variables)

\[ a + b, a - b, a + a, b + b, a + a - b, \ldots \]

and with a set of **inputs**: (pseudo-random)

\[ \text{vector } I = \{(1, 1), (1, 0), (2, 1)\} \]

---

**Bad**

- Expressions derived grows exponentially *(but can still easily achieve 10 nodes AST expressions)*
- This primitive is **unsound** *(it is only sound wrt. } \bar{I} \)

| \(2, 1, 0\) | \(a + b\) |
| \(0, 1, 1\) | \(a - b\) |
| \(2, 2, 4\) | \(a + a\) |
| \(\ldots\) | \(\ldots\) |
Generate a set of programs based on a given grammar: (operators & variables)

\[ a + b, a - b, a + a, b + b, a + a - b, \ldots \]

and with a set of inputs: (pseudo-random)

\[ \text{vector } I = \{(1, 1), (1, 0), (2, 1)\} \]

**Bad**

- Expressions derived grows exponentially *(but can still easily achieve 10 nodes AST expressions)*
- This primitive is **unsound** *(it is only sound wrt. } I \}

**Good**

Generated **only once** and usable on different obfuscations and across programs
QSynth: Overview

Execution tracing (DBI) → Dynamic Symbolic Execution → Obfuscated expressions → Simplification Strategy (for each sub-expression) → Enumerative Synthesis Oracle (generated once for all)

- Obfuscated program
- Execution trace
- Equivalent expression
- Inputs
- Outputs
- Synthesized expressions
QSynth: Overview

- Execution tracing (DBI)
- Dynamic Symbolic Execution
- Obfuscated expressions
- Simplification Strategy (for each sub-expression)
- Enumerative Synthesis Oracle (generated once for all)

Diagram:
- Obfuscated program
- Execution tracing (DBI)
- Execution trace
- Obfuscated expressions
- Inputs
- Outputs
- Equivalent expression
- Synthesized expressions
\[ \phi \triangleq ((((A \lor B) + (A \land B)) \land A) - (((A \lor B) + (A \land B)) \lor A)) \]

\[ I = \{(1, 1), (1, 0), (2, 1)\} \]
\[ \varphi \triangleq (((A \lor B) + (A \land B)) \land A) - (((A \lor B) + (A \land B)) \lor A) \]

\[ I = \{(1, 1), (1, 0), (2, 1)\} \]

\[ O_{\varphi}\text{outputs} = \{3, 0, 1\} \]

\[ SP[\text{outputs}] : \text{not found} \]
\[ \varphi \triangleq (((A \lor B) + (A \land B)) \land A) - (((A \lor B) + (A \land B)) \lor A) \]

\[ I = \{(1, 1), (1, 0), (2, 1)\} \]
\[ O_{\varphi, \text{outputs}} = \{3, 1, 3\} \]
\[ SP[\text{outputs}]: \text{not found} \]
\( \varphi \triangleq (((A \lor B) + (A \land B)) \land A) - (((A \lor B) + (A \land B)) \lor A) \)

\[ I = \{(1, 1), (1, 0), (2, 1)\} \]

\( O_{\varphi, \text{outputs}} = \{0, 1, 2\} \)

\( \text{SP}[\text{outputs}]: \text{not found} \)
AST simplification - Example

\[ \varphi \triangleq (((A \lor B) + (A \land B)) \land A) - (((A \lor B) + (A \land B)) \lor A) \]

\[ \mathcal{I} = \{(1, 1), (1, 0), (2, 1)\} \]

\[ O_{\varphi, \text{outputs}} = \{2, 1, 3\} \]

\[ SP[\text{outputs}]: \text{found} \Rightarrow A + B \]
\[ \varphi \triangleq (((A \lor B) + (A \land B)) \land A) - (((A \lor B) + (A \land B)) \lor A) \]

\[ I = \{(1, 1), (1, 0), (2, 1)\} \]
\[ O_{\varphi, \text{outputs}} = \{2, 1, 3\} \]
\[ SP[\text{outputs}] : \text{found} \Rightarrow A + B \]
$\varphi \triangleq (((A \lor B) + (A \land B)) \land A) - (((A \lor B) + (A \land B)) \lor A)$

\[ I = \{(1, 1), (1, 0), (2, 1)\} \]
\[ O_{\varphi, \text{outputs}} = \{2, 1, 3\} \]
\[ SP[\text{outputs}]: \text{found} \Rightarrow A + B \]
\( \varphi \triangleq (((A \lor B) + (A \land B)) \land A) - (((A \lor B) + (A \land B)) \lor A) \)

\[ \mathcal{I} = \{(1, 1), (1, 0), (2, 1)\} \]

\( O_{\varphi, \text{outputs}} = \{2, 1, 3\} \)

\( SP[\text{outputs}]: \text{found} \Rightarrow A + B \)
\[ \varphi \triangleq (((A \lor B) + (A \land B)) \land A) - (((A \lor B) + (A \land B)) \lor A) \]

\[ I = \{(1, 1), (1, 0), (2, 1)\} \]
\[ O_{\varphi, \text{outputs}} = \{2, 1, 3\} \]
\[ SP[\text{outputs}]: \text{found} \Rightarrow A + B \]
\[ \varphi \triangleq (((A \lor B) + (A \land B)) \land A) - (((A \lor B) + (A \land B)) \lor A) \]

\[ \mathcal{I} = \{(1, 1), (1, 0), (2, 1)\} \]

\[ O_{\varphi, \text{outputs}} = \{0, 1, 3\} \]

\[ SP[\text{outputs}]: \text{found} \Rightarrow V1 \oplus A \]
\( \varphi \triangleq (((A \lor B) + (A \land B)) \land A) - (((A \lor B) + (A \land B)) \lor A) \)
\( \varphi \triangleq (((A \lor B) + (A \land B)) \land A) - (((A \lor B) + (A \land B)) \lor A) \)
\[ \varphi \triangleq (((A \lor B) + (A \land B)) \land A) - (((A \lor B) + (A \land B)) \lor A) \]
\[ \varphi \triangleq (((A \lor B) + (A \land B)) \land A) - (((A \lor B) + (A \land B)) \lor A) \]

Result

**Obfuscated:**

\[ (((A \lor B) + (A \land B)) \land A) - (((A \lor B) + (A \land B)) \lor A) \]

\[ \downarrow \]

**Deobfuscated:**

\[ (A + B) \oplus A \]
Table of Contents

Background
  - Software obfuscation
  - Deobfuscation techniques

Our Synthesis Approach
  - Goal & Contributions
  - Approach steps

Experimental Benchmarks
  - Experimental Setup
  - Benchmarks

Conclusion
Datasets are built with Tigress 2.2 and the EncodeArithmetic (EA), EncodeData (ED) and Virtualization (VR).

In each dataset: 500 obfuscated functions (except 239 for EA-ED)
⇒ Datasets are built with Tigress 2.2 and the EncodeArithmetic (EA), EncodeData (ED) and Virtualization (VR).

⇒ In each dataset: 500 obfuscated functions (except 239 for EA-ED)

<table>
<thead>
<tr>
<th></th>
<th>Mean size $\phi$ (in node)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Original</td>
</tr>
<tr>
<td>#1: Syntia †</td>
<td>3.97</td>
</tr>
<tr>
<td>#2: EA</td>
<td>13.5</td>
</tr>
<tr>
<td>#3: VR-EA</td>
<td>13.5</td>
</tr>
<tr>
<td>#4: EA-ED</td>
<td>13.5</td>
</tr>
</tbody>
</table>

†use EA-ED (with 5 derivations max, other are 21 max)
Datasets are built with Tigress 2.2 and the EncodeArithmetic (EA), EncodeData (ED) and Virtualization (VR).

In each dataset: 500 obfuscated functions (except 239 for EA-ED)

<table>
<thead>
<tr>
<th></th>
<th>Original</th>
<th>Obfuscated</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1: Syntia †</td>
<td>3.97</td>
<td>203.19</td>
</tr>
<tr>
<td>#2: EA</td>
<td>13.5</td>
<td>131.56</td>
</tr>
<tr>
<td>#3: VR-EA</td>
<td>13.5</td>
<td>443.64</td>
</tr>
<tr>
<td>#4: EA-ED</td>
<td>13.5</td>
<td>9223.46</td>
</tr>
</tbody>
</table>

†use EA-ED (with 5 derivations max, other are 21 max)

# lookup table ($SP$): 3,358,709 expressions (14 sets of 3 vars & 5 operators each)

input vector size $\bar{I}$ (for $SP$): 15
## Simplification

<table>
<thead>
<tr>
<th></th>
<th>Mean expr. size</th>
<th>Simplification</th>
<th>Mean scale factor</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Orig</td>
<td>Obf&lt;sub&gt;B&lt;/sub&gt;</td>
<td>Synt</td>
</tr>
<tr>
<td>Syntia</td>
<td>/</td>
<td>/</td>
<td>/</td>
</tr>
<tr>
<td>QSynth</td>
<td>3.97</td>
<td>203.19</td>
<td>3.71</td>
</tr>
</tbody>
</table>

Orig, Obf<sub>S</sub>, Obf<sub>B</sub>, Synt are resp. original, obfuscated (source, binary level) and synthesized exprs.
Simplification

<table>
<thead>
<tr>
<th>Mean expr. size</th>
<th>Simplification</th>
<th>Mean scale factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Orig</td>
<td>Obf_B</td>
<td>Synt</td>
</tr>
<tr>
<td>Syntia</td>
<td>/</td>
<td>/</td>
</tr>
<tr>
<td>QSynth</td>
<td>3.97</td>
<td>203.19</td>
</tr>
</tbody>
</table>

Orig, Obf_S, Obf_B, Synt are rsp. original, obfuscated (source, binary level) and synthesized exprs

Accuracy & Speed

<table>
<thead>
<tr>
<th>Semantic</th>
<th>Time</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sym.Ex</td>
<td>Synthesis</td>
</tr>
<tr>
<td>Syntia</td>
<td>/</td>
<td>/</td>
</tr>
<tr>
<td>QSynth</td>
<td>500</td>
<td>1m20s</td>
</tr>
</tbody>
</table>
## Simplification

<table>
<thead>
<tr>
<th>Dataset 2</th>
<th>EA</th>
<th>Mean expr. size</th>
<th>Simplification</th>
<th>Mean Scale factor</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Orig</td>
<td>Obf</td>
<td>Synt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>13.5</td>
<td>245.81</td>
<td>21.92</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Dataset 3</td>
<td>VR-EA</td>
<td>13.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>13.5</td>
<td>9223.46</td>
<td>3812.84</td>
</tr>
</tbody>
</table>

Orig, Obf_S, Obf_B, Synt are respectively original, obfuscated (source, binary level) and synthesized expressions
### Simplification

<table>
<thead>
<tr>
<th>Mean expr. size</th>
<th>Simplification</th>
<th>Mean Scale factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Orig</td>
<td>Obf&lt;sub&gt;B&lt;/sub&gt;</td>
<td>Synt</td>
</tr>
<tr>
<td>Dataset 2</td>
<td>EA</td>
<td>13.5</td>
</tr>
<tr>
<td>Dataset 3</td>
<td>VR-EA</td>
<td>13.5</td>
</tr>
<tr>
<td>Dataset 4</td>
<td>EA-ED</td>
<td>13.5</td>
</tr>
</tbody>
</table>

Orig, Obf<sub>S</sub>, Obf<sub>B</sub>, Synt are respectively original, obfuscated (source, binary level) and synthesized expressions.

### Accuracy & Speed

<table>
<thead>
<tr>
<th>Semantic</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sym.Ex</td>
</tr>
<tr>
<td>Dataset 2</td>
<td>OK: 413</td>
</tr>
<tr>
<td>Dataset 3</td>
<td>OK: 401</td>
</tr>
<tr>
<td>Dataset 4</td>
<td>EA-ED</td>
</tr>
</tbody>
</table>
Conclusion

Challenge
⇒ Deobfuscating some data-flow based (composite) obfuscations
Conclusion

Challenge
⇒ Deobfuscating some data-flow based (composite) obfuscations

Results
⇒ A scalable synthesis algorithm improving the state-of-the-art in both speed and accuracy
Conclusion

Challenge
⇒ Deobfuscating some data-flow based (composite) obfuscations

Results
⇒ A scalable synthesis algorithm improving the state-of-the-art in both speed and accuracy

Limitation:
► synthesizing expressions using constants
► addressing encoded-data (which scale)
Conclusion

**Challenge**
⇒ Deobfuscating some data-flow based (composite) obfuscations

**Results**
⇒ A scalable synthesis algorithm improving the state-of-the-art in both **speed** and **accuracy**

**Limitation:**
- synthesizing expressions using constants
- addressing encoded-data *(which scale)*

**Future work:**
- experimenting other synthesis primitives & simplification strategies *(D&C..)*
- combining with other approach *(not necessarily synthesis-based)*
- testing against other obfuscators
Thank you!


Goal: Finding the smallest discriminative input vector size
How: Checking equivalence by SMT with synthesized expr. (on EA)

**x axis:** input vector size, **y axis:** Function number
Presetting pre-computed synthesis lookup tables

**Goal:** Finding the *smallest discriminative input vector size*

**How:** Checking equivalence by SMT with synthesized expr. (on EA)

**Conclusion**

We chose **15** as a good trade-off between **semantic accuracy** and **evaluation speed**.
Synthesis time distribution (on EA)
Synthesis simplification (on EA)