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)

Takeaway

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

\[ a + b \quad \Rightarrow \quad ((((((a \land \neg b) + b) \ll 1) \land \neg((a \lor b) - (a \land b))) \ll 1) - (((a \land \neg b) + b) \ll 1) \oplus ((a \lor b) - (a \land b)))) \]
# 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**

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

**Problem:** Reverting an obfuscating transformation is hard.
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");
        }
    }
}
```

Diagram:

- Node A: $a < 10$
- Node B: $a > b$
- Node C: $\text{print(\"OK\")}$

Formula:

$$a < 10 \land a > b$$

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 State**
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

\[ \phi_b = b \]
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**

$$\phi_b = b$$

$$\phi_b = b + (a | -1) - 1$$
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**

- \( \phi_b = b \)
- \( \phi_b = b + (a | -1) - 1 \)
- \( \phi_b = b + (a | -1) - 1 - ((\sim a) \& -1) \)
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);
```

<table>
<thead>
<tr>
<th>Symbolic State</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \phi_b = b )</td>
</tr>
<tr>
<td>( \phi_b = b + (a</td>
</tr>
<tr>
<td>( \phi_b = b + (a</td>
</tr>
<tr>
<td>( \phi_b = 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
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>

⇒ 

Problem

Synthesizing programs (expressions) with complex behaviors is hard.
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

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>
</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>
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

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>

⇒ \( a + b \)

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

Idea: Using symbolic execution to reduce the synthesis 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.
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

**Execution tracing (DBI)**

**Enumerative Synthesis Oracle**
(generated once for all)

**Simplification Strategy**
(for each sub-expression)

- inputs
- equivalent expression
- outputs

**Obfuscated program**

**Obfuscated expressions**

**Execution trace**

**Dynamic Symbolic Execution**

**synthesized expressions**
QSynth: Overview

QBDI Tool:

Execution tracing
(DBI)

Obfuscated program

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
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.)*

https://qbdi.quarkslab.com/
QSynth: Overview

Tool: QBDI
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

Dynamic Binary Analysis
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.

```c
if (a > 0) {
    b += a - 1;
} else {
    b += 2;
}
```

DSE: Symbolic expression computation
Triton allows computing any **symbolic expression** along the trace by backtracking on data dependencies

```c
if (a > 0) {
    b += a - 1;
} else {
    b += 2;
}
```

```
if
    -
    +
    a
    1
    +
    b
    2
```

\[ \phi \equiv (b + (a - 1)) - 1 \]

O \(\phi\) the associated I/O oracle can be evaluated on different inputs
Triton allows computing any **symbolic expression** along the trace by backtracking on data dependencies.

\begin{align*}
\varphi & \triangleq (b + (a - 1)) - 1 \\
O_\varphi & \text{ the associated I/O oracle can be evaluated on different inputs}
\end{align*}
QSynth: Overview

Execution tracing (DBI) → Dynamic Symbolic Execution

Obfuscated program → Execution trace → Obfuscated expressions

Enumerative Synthesis Oracle (generated once for all) → Simplification Strategy (for each sub-expression)

Tool: TRILON

inputs → outputs → equivalent expression → synthesized expressions
QSynth: Overview

Obfuscated program

Execution tracing (DBI)

Execution trace

Dynamic Symbolic Execution

Obfuscated expressions

Enumerative Synthesis Oracle

generated once for all

inputs

outputs

equivalent expression

Simplification Strategy

(for each sub-expression)

synthesized expressions
Definition

We call Synthesis Primitive any program $SP$ taking as input parameters a black-box oracle $O_\varphi$ and a set of input parameters to the oracle $I$, and returning, in case of success, a program $p$, such that for any $i \in I$ then $p(i) = O_\varphi(i)$.

$$SP(O_\varphi, I) \Rightarrow p \mid \forall i \in I, p(i) \equiv O_\varphi(i)$$

$$SP(O_\varphi, I) \Rightarrow \emptyset$$
Generate a set of programs based on a given grammar: (operators & variables)

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

\[
\text{vector } I = \{(1, 1), (1, 0), (2, 1)\}
\]
Generate a set of programs based on a given grammar: \((\text{operators} \& \text{variables})\)

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

and with a set of inputs: \((\text{pseudo-random})\)

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

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

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

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

**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>
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>
<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. } I\)
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 $I = \{(1, 1), (1, 0), (2, 1)\}$

**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)
Obfuscated program
Dynamic Symbolic Execution
Execution trace
Enumerative Synthesis Oracle
(generated once for all)
Simplification Strategy
(for each sub-expression)
inputs
outputs
inputs
outputs
equivalent expression
synthesized expressions
Obfuscated expressions
QSynth: Overview

Obfuscated program

Execution tracing (DBI)

Execution trace

Dynamic Symbolic Execution

Obfuscated expressions

Enumerative Synthesis Oracle (generated once for all)

inputs
outputs

Simplification Strategy (for each sub-expression)

equivalent expression

synthesized expressions
AST simplification - Example

\( \varphi \overset{\Delta}{=} (((A \lor B) + (A \land B)) \land A) - (((A \lor B) + (A \land B)) \lor A) \)

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

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

\[ \mathcal{O}_\varphi[\text{outputs}] = \{3, 0, 1\} \]

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

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

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

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

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

\( O_\phi[\text{outputs}] = \{0, 1, 2\} \)

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

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

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

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

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

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

$S\mathcal{P}[\text{outputs}]: \text{found} \Rightarrow A + B$
AST simplification - Example

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

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

\[ \mathcal{O}_\varphi \text{outputs} = \{2, 1, 3\} \]

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

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

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

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

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

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

\[ \mathcal{O}_{\varphi,\text{outputs}} = \{0, 1, 3\} \]

\[ \mathcal{SP}[\text{outputs}]: \textbf{found} \Rightarrow V1 \oplus A \]
AST simplification - Example

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

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 \textit{EncodeArithmetic (EA)}, \textit{EncodeData (ED)} and \textit{Virtualization (VR)}.

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

<table>
<thead>
<tr>
<th></th>
<th>Mean size $\varphi$ (in node)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Original</td>
</tr>
<tr>
<td>#1: Syntia $^\dagger$</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>

$^\dagger$use EA-ED (with 5 derivations max, other are 21 max)

\# lookup table ($SP$): 3,358,709 expressions \hspace{1em} (14 sets of 3 vars & 5 operators each)

input vector size $I$ (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>3.97</td>
<td>203.19</td>
<td>3.71</td>
</tr>
<tr>
<td>QSynth</td>
<td>0</td>
<td>500</td>
<td>500</td>
</tr>
</tbody>
</table>

Orig, Obf<sub>S</sub>, Obf<sub>B</sub>, Synt are respectively original, obfuscated (source, binary level) and synthesized expressions.
**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><strong>Syntia</strong></td>
<td>/</td>
<td>/</td>
<td>/</td>
</tr>
<tr>
<td><strong>QSynth</strong></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 rsp. original, obfuscated (source, binary level) and synthesized exprs

**Accuracy & Speed**

<table>
<thead>
<tr>
<th></th>
<th>Semantic</th>
<th>Time</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Sym.Ex</td>
<td>Synthesis</td>
<td>Total</td>
</tr>
<tr>
<td><strong>Syntia</strong></td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>34 min</td>
</tr>
<tr>
<td><strong>QSynth</strong></td>
<td>500</td>
<td>1m20s</td>
<td>15s</td>
<td>1m35s</td>
</tr>
</tbody>
</table>
## Simplification

<table>
<thead>
<tr>
<th>Dataset</th>
<th>EA</th>
<th>VR-EA</th>
<th>EA-ED</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean expr. size</td>
<td>Obf₂</td>
<td>Synt</td>
<td>Obf₂</td>
</tr>
<tr>
<td>Orig</td>
<td>13.5</td>
<td>13.5</td>
<td>13.5</td>
</tr>
<tr>
<td>Obf₂</td>
<td>245.81</td>
<td>443.64</td>
<td>9223.46</td>
</tr>
<tr>
<td>Synt</td>
<td>21.92</td>
<td>25.42</td>
<td>3812.84</td>
</tr>
</tbody>
</table>

### Simplification

<table>
<thead>
<tr>
<th>Dataset</th>
<th>EA</th>
<th>VR-EA</th>
<th>EA-ED</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>500</td>
<td>500</td>
<td>500</td>
</tr>
<tr>
<td>0</td>
<td>500</td>
<td>500</td>
<td>234</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td>234</td>
</tr>
</tbody>
</table>

### Mean Scale factor

<table>
<thead>
<tr>
<th>Dataset</th>
<th>EA</th>
<th>VR-EA</th>
<th>EA-ED</th>
</tr>
</thead>
<tbody>
<tr>
<td>Obf₂/Orig</td>
<td>x18.34</td>
<td>375 (75.00%)</td>
<td>x405.25</td>
</tr>
<tr>
<td>Synt/Orig</td>
<td>x1.64</td>
<td>x1.90</td>
<td>x234.44</td>
</tr>
</tbody>
</table>

Orig, Obf₂, Obf₂, Synt are respectively original, obfuscated (source, binary level) and synthesized expressions.
Tigress benchmark

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>Obf₃</td>
<td>Synt</td>
</tr>
<tr>
<td>Dataset 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EA</td>
<td>13.5</td>
<td>245.81</td>
</tr>
<tr>
<td>Dataset 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VR-EA</td>
<td>13.5</td>
<td>443.64</td>
</tr>
<tr>
<td>Dataset 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EA-ED</td>
<td>13.5</td>
<td>9223.46</td>
</tr>
</tbody>
</table>

Orig, Obf₃, Obf₃, Synt are respectively original, obfuscated (source, binary level) and synthesized expressions.

Accuracy & Speed

<table>
<thead>
<tr>
<th>Semantic</th>
<th>Time</th>
<th>per fun.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sym.Ex</td>
<td>Synthesis</td>
</tr>
<tr>
<td>Dataset 2</td>
<td>OK: 413</td>
<td>KO: 4</td>
</tr>
<tr>
<td>EA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dataset 3</td>
<td>OK: 401</td>
<td>KO: 43</td>
</tr>
<tr>
<td>VR-EA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dataset 4</td>
<td>-</td>
<td></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 \((\text{composite})\) obfuscations

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

Limitation:
▶ synthesizing expressions using constants
▶ addressing encoded-data \((\text{which scale})\)
**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!


Presetting pre-computed synthesis lookup tables

**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)