No More Gotos: Decompilation Using Pattern-Independent Control-Flow Structuring and Semantics-Preserving Transformations

Khaled Yakdan\textsuperscript{1}  \hspace{1cm} Sebastian Eschweiler\textsuperscript{2}  \hspace{1cm} Elmar Gerhards-Padilla\textsuperscript{2}  \hspace{1cm} Matthew Smith\textsuperscript{1}

\textsuperscript{1}University of Bonn, Germany
\textsuperscript{2}Fraunhofer FKIE, Germany

NDSS 2015
Agenda

01 Motivation

02 Control Flow Structuring

03 The DREAM Decompiler

04 Results

05 Conclusion
Motivation
Decomposition in Security

Binary code

Khaled Yakdan (University of Bonn)  DREAM Decompiler  NDSS 2015
Motivation
Decomposition in Security

Source code

```c
int f(int a) {
    int i = 0;
    for (; i < a; i++)
        ...
}
```

Compilation

High-level abstractions are lost

Binary code

```
01010101010101010100
01010101010101010100
01010101010101010100
01010101010101010100
01010101010101010100
01010101010101010100
```
**Motivation**

Decompilation in Security

---

### Source code

```
int f(int a) {
    int i = 0;
    for (; i < a; i++)
        ...
}
```

### Decompiled code

```
int f(int arg) {
    int var = 0;
    while (var < arg)
        ...
    var = var + 1;
}
```

### High-level abstractions are lost

---

### Binary code

```
01010101010101010100 01010101010101010100 01010101010101010100
01010101010101010100 01010101010101010100 01010101010101010100
01010101010101010100 01010101010101010100 01010101010101010100
```

---

**Recovered abstractions**
Motivation
Decompilation in Security

- Manual reverse engineering
Motivation
Decomposition in Security

- Manual reverse engineering
- Apply source-based techniques to binary code
**Motivation**

Decomposition in Security

- Manual reverse engineering
- Apply source-based techniques to binary code
  - Find vulnerabilities, bugs
Motivation
Decompilation in Security

- Manual reverse engineering
- Apply source-based techniques to binary code
  - Find vulnerabilities, bugs
  - Taint tracking
Motivation
Decomposition in Security

- Manual reverse engineering
- Apply source-based techniques to binary code
  - Find vulnerabilities, bugs
  - Taint tracking

Goal: Enhanced Structuredness
Effective control flow structure recovery to improve readability and enhance program analysis
Control Flow Structuring

\[
\text{if} (c_1) \quad \text{while} (c_2)\]

\[
\neg c_1 \quad \neg c_2 \quad c_1 \quad c_2 \quad n_1 \quad n_2
\]

Khaled Yakdan (University of Bonn)
Control Flow Structuring

\[
\text{if} (c_1) \quad \text{while} (c_2)
\]

Khaled Yakdan (University of Bonn)
Structured code

```c
int f(int a, int b)
{
    int sum = 0;
    if(a < b){
        for(int i = a; i < b; i++)
            sum += i;
    }
    return sum;
}
```

Unstructured code

```c
int f(int a, int b)
{
    int sum = 0;
    if(a >= b)
        goto Label_2;
    int i = a;
    Label_1:
    if(i >= b)
        goto Label_2;
    sum += i;
    i++;
    goto Label_1;
    Label_2:
    return sum;
}
```
State of the art: *Structural Analysis* [Sharir80]

- Pattern-matching using a predefined set of region schemas (patterns)
State of the art: **Structural Analysis** [Sharir80]

- Pattern-matching using a predefined set of region schemas (patterns)
- Use goto statements if no match is found
State of the art: **Structural Analysis** [Sharir80]

- Pattern-matching using a predefined set of region schemas (patterns)
- Use goto statements if no match is found
- Example: Decompiling a P2P Zeus sample with Hex-Rays
  - 1,571 goto for 49,514 LoC
  - 1 goto for each 32 LoC
Prior Work on Control-Flow Structuring

Improving vanilla structural analysis to recover more structure

- SESS Analysis [Engel et al., SCOPES 2011]
- Phoenix Decompiler [Schwartz et al., USENIX Security 2013]
Prior Work on Control-Flow Structuring

Improving vanilla structural analysis to recover more structure

- SESS Analysis [Engel et al., SCOPES 2011]
- Phoenix Decompiler [Schwartz et al., USENIX Security 2013]

New control-flow structuring algorithm

Pattern-Independent Structuring

Semantics-Preserving Transformations
Decompiler for Reverse Engineering and Analysis of Malware

IR Transformation
Decompiler for Reverse Engineering and Analysis of Malware

IR Transformation → Data Flow Analysis
Decompiler for Reverse Engineering and Analysis of Malware

IR Transformation → Data Flow Analysis → Type Recovery
Decompiler for Reverse Engineering and Analysis of Malware

IR Transformation → Data Flow Analysis → Type Recovery
Decompiler for Reverse Engineering and Analysis of Malware

IR Transformation → Data Flow Analysis → Type Recovery

Control Flow Structuring

Khaled Yakdan (University of Bonn)
Decompiler for Reverse Engineering and Analysis of Malware

IR Transformation → Data Flow Analysis → Type Recovery

Control Flow Structuring → Post-structuring Optimizations
Decompiler for Reverse Engineering and Analysis of Malware

- IR Transformation
- Data Flow Analysis
- Type Recovery

- Control Flow Structuring
- Post-structuring Optimizations

Khaled Yakdan (University of Bonn)
Running Example
Running Example
Running Example

Khaled Yakdan (University of Bonn)
Acyclic Regions

Lexical order
Reaching conditions
Initial AST as a sequence of if constructs
Refine initial AST to find switch, if-else constructs

if (¬b) n
if (b ∧ c) n
if (¬b ∨ ¬c) n
n
if (¬b) n
if (b ∧ c) n
else n
n
n
n
n

b
¬b
b
¬b
b
c
¬c
c
¬c

Khaled Yakdan (University of Bonn)
DREAM Decompiler
NDSS 2015
Acyclic Regions

- Lexical order

![Diagram of graph with nodes and edges representing acyclic regions]

- Initial AST as a sequence of if constructs
- Refine initial AST to find switch, if-else constructs

\[
\begin{align*}
\text{if} (\neg b) & \quad n_1 \\
\text{if} (b \land c) & \quad n_2 \\
\text{else} & \quad n_3 \\
\text{if} (\neg b \lor \neg c) & \quad n_4
\end{align*}
\]
Acyclic Regions

- Lexical order
- Reaching conditions
Acyclic Regions

- Lexical order
- Reaching conditions
- Initial AST as a sequence of if constructs

```plaintext
if (¬b)
  n1
if (b ∧ c)
  n2
if (¬b ∨ ¬c)
  n3
n4
```
Acyclic Regions

- Lexical order
- Reaching conditions
- Initial AST as a sequence of if constructs
- Refine initial AST to find switch, if-else constructs

\[
\text{if} \ (¬b) \\
\quad n_1 \\
\text{if} \ (b \land c) \\
\quad n_2 \\
\text{if} \ (¬b \lor ¬c) \\
\quad \text{else} \\
\quad n_3 \\
\quad n_4
\]

\[
\text{if} \ (¬b) \\
\quad n_1 \\
\text{if} \ (b \land c) \\
\quad n_2 \\
\quad \text{else} \\
\quad n_3 \\
\quad n_4
\]
Running Example

Khaled Yakdan (University of Bonn)

DREAM Decompiler

NDSS 2015 14 / 28
Running Example

Diagram with nodes and edges labeled with propositions and negations.
Cyclic Regions

Identify loop nodes and successor node
Replace edges to the successor node by break statements
Structure loop body

Initial AST:
while (1)
{

}

Infer loop type and condition
while (1)
...
if (¬e) break
do ...
while (e)

Khaled Yakdan (University of Bonn)
Cyclic Regions

- Identify loop nodes and successor node

Initial AST:

```c
while (1) {
    // Body
}
```

Infer loop type and condition:

```c
while (1) . . .
if (!e) break
do . . .
while (e)
```

```
\[ n_5 \rightarrow n_6 \rightarrow n_7 \rightarrow \neg e \]
```

```
\[ \neg d \rightarrow d \rightarrow e \rightarrow \neg e \rightarrow n_8 \]
```

successor node
Cyclic Regions

- Identify loop nodes and successor node
- Replace edges to the successor node by **break** statements
Cyclic Regions

- Identify loop nodes and successor node
- Replace edges to the successor node by `break` statements
- Structure loop body $B_{AST}$
Cyclic Regions

- Identify loop nodes and successor node
- Replace edges to the successor node by \textbf{break} statements
- Structure loop body $B_{\text{AST}}$
- Initial AST: \textbf{while} (1) $\{B_{\text{AST}}\}$

\begin{verbatim}
while (1)
  ...
  if (¬e)
    break
\end{verbatim}
Cyclic Regions

- Identify loop nodes and successor node
- Replace edges to the successor node by **break** statements
- Structure loop body $B_{AST}$
- Initial AST: **while** (1) \{ $B_{AST}$ \}
- Infer loop type and condition

\[
\begin{align*}
\text{while} & \ (1) \\
\ldots & \\
\text{if} & \ (\neg e) \\
\text{break} & 
\end{align*}
\]

\[
\begin{align*}
\text{do} & \\
\ldots & \\
\text{while} & \ (e)
\end{align*}
\]
Running Example

Khaled Yakdan (University of Bonn)

DREAM Decompiler

NDSS 2015
Agenda

01 Motivation

02 Control Flow Structuring

03 The DREAM Decompiler

04 Results

05 Conclusion
Readability Enhancements

```c
int f(int a1){
    int v2 = 0;
    while((a1 <= 1 && v2 <= 100)
          || (a1 > 1 && v2 <= 10)){
        printf("inside_loop");
        ++v2;
        --a1;
    }
    printf("loop_terminated");
    return v2;
}
```

```
signed int f(signed int a1){
    signed int v2;
    v2 = 0;
    while ( a1 > 1 ){
        if ( v2 > 10 )
            goto LABEL_7;
    LABEL_6:
        printf("inside_loop");
        ++v2;
        --a1;
    }
    if ( v2 <= 100 )
        goto LABEL_6;
    LABEL_7:
    printf("loop_terminated");
    return v2;
}
```

DREAM

Hex-Rays
Evaluation

Metrics

- Correctness
- Structuredness
- Compactness
Correctness

Experiment Setup

Source Code
(coreutils)

\[\text{[Yamaguchi et al. IEEE S&P 2014]}\]
Correctness

Experiment Setup

Source Code (coreutils)

Parsing (*joern*)¹

CFGs

¹[Yamaguchi et al. IEEE S&P 2014]
Correctness

Experiment Setup

Source Code (coreutils)

Parsing (joern$^1$)

CFGs

Control Flow Structuring

Structured Code

$^1$[Yamaguchi et al. IEEE S&P 2014]
Correctness

Experiment Setup

Source Code (coreutils)

Parsing (joern\(^1\))

CFGs

Control Flow Structuring

Structured Code

Passes tests?

---

\(^1\)[Yamaguchi et al. IEEE S&P 2014]
### Correctness

#### Results

| Considered Functions $F$                              | $|F|$ | Number of gotos |
|-------------------------------------------------------|------|-----------------|
| Functions after preprocessor                         | 1,738| 219             |
| Functions correctly parsed by *joern* \(^2\)          | 1,530| 129             |
| Functions passed tests after structuring              | 1,530| 0               |

\(^2\)Errors have been reported to *joern’s* authors and are fixed in the current release.
Structuredness and Compactness

- Tested decompilers
  - DREAM
  - Phoenix (academic state of the art)
  - Hex-Rays (industry state of the art)

- Structuredness
  - Number of gotos

- Compactness
  - Total lines of code
  - Compact functions
Structuredness

- DREAM: 0
- Phoenix: 4,231
- Hex-Rays: 2,949

Khaled Yakdan (University of Bonn)
Compactness

- DREAM: 107 kSLOC
- Phoenix: 164 kSLOC
- Hex-Rays: 135 kSLOC
Conclusion

- Novel control flow structuring algorithm
  - pattern-independent structuring
  - semantics-preserving transformations
- DREAM decompiler
  - goto-free decompiled code
  - compact code
  - good readability
Thank You!

Questions?