DeBIN: Predicting Debug Information in Stripped Binaries
https://debin.ai

Jingxuan He
Pesho Ivanov
Petar Tsankov
Veselin Raychev
Martin Vechev
Binaries with debug symbols

Assembly

80534BA:
push %ebp
push %edi
push %esi ...

Debug symbols

80534BA rfc1035_init int
8053DB1 fopen64 int
8063320 num_entries int

Descriptive names for functions and variables

int rfc1035_init() {
  ...
  if ( num_entries <= 0 ) {
    v0 = ("/etc/resolv.conf", 'r');
    if ( v0 || (v1 = fopen64("resolv.conf"))){
      // code to read and
      // manipulate DNS settings
    }
  }
  ...
}

Decompiled code
Stripped binaries

Assembly

80534BA:
push %ebp
push %edi
push %esi ...

Debug symbols

Non-descriptive names

int sub_80534BA() {
  ...
  if ( dword_8063320 <= 0 ) {
    v0 = ("/etc/resolv.conf", 'r');
    if ( v0 || (v1 = sub_8053B1("resolv.conf"))){
      ...
  }

Can we recover the debug symbols?

Yes, with roughly 65% accuracy!
Challenges

1. No mapping from registers and memory offsets to semantic variables

```assembly
<sum> start:
mov 4(%esp), %ecx
mov $0, %eax
mov $1, %edx
add %edx, %eax
add $1, %edx
cmp %ecx, %edx
jne 8048400
repz ret
<sum> end
```
Challenges

1. No mapping from registers and memory offsets to semantic variables
2. No names and types

Store the values of the unsigned integer variable \( n \)

Stores the result in an integer variable \( \text{res} \)

```
<sum> start:
    mov 4(%esp), %ecx
    mov $0, %eax
    mov $1, %edx
    add %edx, %eax
    add $1, %edx
    cmp %ecx, %edx
    jne 8048400
    repz ret
<sum> end
```
DeBIN: Recovering debug information

DeBIN recovers location information, types, and names.
Predicting Debug Information in Stripped Binaries

DEBIN uses machine learning to recover debug information (e.g., names and types) of stripped binaries (x86, x64, ARM). This is helpful for various binary analysis tasks, such as disassembly, verification, malware inspection and similarity.

DEMO

Select Binary File

Upload

Linux ELF binaries on x86, x64 and ARM (without Thumb instructions), 2MB maximum.

or try samples:  lcrack.x86  chgpasswd.x64  ls.ARM
How does DeBIN work?
DeBIN: System overview

Binary with debug symbols

Learning phase

Variable recovery model

Names/ types model

Prediction phase

Assembly

start:
mov 4(%esp), %ecx
mov $0, %eax
mov $1, %edx
add %edx, %eax

Debug symbols

start: sum int
4(%esp) n uint
$eax res int
$edx i uint

Stripped binary

Binary with debug symbols
Step 1: Recovering variables
Learning how to recover variables

Binaries with debug symbols

>8K binaries

>10K distinct features

>2M vectors

Extracted features

Binary feature vectors

Ensemble of trees

100 decision trees

Feature templates

plus[Reg][Val]
inst[Op][Reg]
dep[Reg][Reg]

...
Variable recovery

```
Variable recovery

%edx.2 → plus[%edx][1] inst[add][%edx]
```
Step 2: Predicting names and types
Probabilistic graphical model

<table>
<thead>
<tr>
<th>f</th>
<th>i</th>
<th>n</th>
<th>0.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>p</td>
<td>s</td>
<td>0.3</td>
</tr>
<tr>
<td>f</td>
<td>a</td>
<td>b</td>
<td>0.1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>f</th>
<th>i</th>
<th>i</th>
<th>0.8</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>i</td>
<td>j</td>
<td>0.6</td>
</tr>
<tr>
<td>f</td>
<td>p</td>
<td>p</td>
<td>0.3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1</th>
<th>i</th>
<th>i</th>
<th>0.8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>j</td>
<td>i</td>
<td>0.6</td>
</tr>
<tr>
<td>1</td>
<td>p</td>
<td>p</td>
<td>0.3</td>
</tr>
</tbody>
</table>

Known elements
- ECX.1, ...

Unknown elements
- EDX.2, EDX.3

Binary features
- f₁, f₂, ...

Factors
- cond-NE-EDX-ECX
dep-EDX-EDX
Probabilistic graphical model

<table>
<thead>
<tr>
<th>Relationship</th>
<th>Template</th>
<th>Condition for adding an edge</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Element used in</td>
<td>$(f, v, \text{func-loc}(v))$</td>
<td>variable $v$ is accessed inside the scope of function $f$</td>
</tr>
<tr>
<td>Function Call</td>
<td>$(f_1, f_2, \text{call})$</td>
<td>function $f_2$ is called by function $f_1$</td>
</tr>
<tr>
<td>Instruction</td>
<td>$(v, \text{insn}, \text{insn-loc}(v))$</td>
<td>there is an instruction $\text{insn}$ (e.g., add) that operates on variable $v$</td>
</tr>
<tr>
<td>Location</td>
<td>$(v, l, \text{locates-at})$</td>
<td>variable $v$ locates at location $l$ (e.g., memory offset $\text{mem}[\text{RSP}+16]$)</td>
</tr>
<tr>
<td>Locality</td>
<td>$(v_1, v_2, \text{local-loc}(v_1))$</td>
<td>variable $v_1$ and $v_2$ are locally allocated (e.g., EDX.2 and EDX.3)</td>
</tr>
<tr>
<td>Dependency</td>
<td>$(v_1, v_2, \text{dep-loc}(v_1) - \text{loc}(v_2))$</td>
<td>variable $v_1$ is dependent on variable $v_2$</td>
</tr>
<tr>
<td>Operation</td>
<td>$(v, \text{op}, \text{unary-loc}(v))$</td>
<td>unary operation $\text{op}$ (e.g., unsigned and low cast) on variable $v$</td>
</tr>
<tr>
<td></td>
<td>$(n_1, n_2, \text{op-loc}(n_1) - \text{loc}(n_2))$</td>
<td>binary operation $\text{op}$ (e.g., +, left shift « and etc.) on node $n_1$ and $n_2$</td>
</tr>
<tr>
<td></td>
<td>$(v_1, v_2, \text{phi-loc}(v_1))$</td>
<td>there is a $\phi$ expression in BAP-IR: $v_1 = \phi(..., v_2, ...)$</td>
</tr>
<tr>
<td>Conditional</td>
<td>$(v, \text{op}, \text{cond-loc}(n_1) - \text{loc}(n_2))$</td>
<td>there is a conditional expression $\text{op}(v)$ (e.g., $\text{not}(\text{EAX.2})$)</td>
</tr>
<tr>
<td></td>
<td>$(n_1, n_2, \text{cond-op-loc}(n_1) - \text{loc}(n_2))$</td>
<td>there is a conditional expression $n_1 \text{ op } n_2$ (e.g., EDX.3 = ECX.1)</td>
</tr>
<tr>
<td>Argument</td>
<td>$(f, a, \text{call-arg-loc}(a))$</td>
<td>there is a call $f(..., a, ...)$ with argument $a$</td>
</tr>
<tr>
<td>Type</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Operation</td>
<td>$(t, \text{op}, \text{t-unary-loc}(t))$</td>
<td>unary operation $\text{op}$ on type $t$</td>
</tr>
<tr>
<td></td>
<td>$(t_1, t_2, \text{t-op-loc}(t_1) - \text{loc}(t_2))$</td>
<td>binary operation $\text{op}$ on type $t_1$ and $t_2$</td>
</tr>
<tr>
<td></td>
<td>$(t_1, t_2, \text{t-phi-loc}(t_1))$</td>
<td>there is a $\phi$ expression: $t_1 = \phi(..., t_2, ...)$</td>
</tr>
<tr>
<td>Conditional</td>
<td>$(t, \text{op}, \text{t-cond-unary})$</td>
<td>there is a unary conditional expression $\text{op}(t)$</td>
</tr>
<tr>
<td></td>
<td>$(t_1, t_2, \text{t-cond-op-loc}(t_1) - \text{loc}(t_2))$</td>
<td>there is a binary conditional expression $t_1 \text{ op } t_2$</td>
</tr>
<tr>
<td>Argument</td>
<td>$(f, t, \text{t-call-arg-loc}(t))$</td>
<td>call $f(..., t, ...)$ with an argument of type $t$</td>
</tr>
<tr>
<td>Name &amp; Type</td>
<td>$(v, t, \text{type-loc}(v))$</td>
<td>variable $v$ is of type $t$</td>
</tr>
<tr>
<td></td>
<td>$(f, t, \text{func-type})$</td>
<td>function $f$ is of type $t$</td>
</tr>
</tbody>
</table>

Known elements: $f_4, f_5, f_6$  
Unknown elements: $i, j$  
Binary features: $p$  
Factors: $i, n$
Probabilistic graphical model

Next
How are the features and their weights learned?

Unknown elements
Known elements
Binary features
Factors
Learning how to predict names and types

> 8,000 binaries

Binaries with debug symbols

(\texttt{funary,Op,Var})
(\texttt{var-dep,Var_1,Var_2})
...

Feature templates

23 templates

Dependency graphs

Binary features and factors

Actual graphs have >1K nodes

Train model

Find \textit{weights} that maximize

\[ P(\bar{U} = \bar{u} | \bar{K} = \bar{k}_i) \text{ for all training samples } (\bar{u}_i, \bar{k}_i) \]
End-to-end recovery of debug information
Recovering debug information

<sum> start :
  mov 4(%esp), %ecx
  mov $0, %eax
  mov $1, %edx
  add %edx, %eax
  add $1, %edx
  cmp %ecx, %edx
  jne 8048400
  repz ret
<sum> end

Stripped binary

Registers / mem offsets
- EDX.2
- EDX.3
- EDX.1
- ECX.1

Known elements
- 0
- 1
- mov

Semantic variables
- EDX.2
- EDX.3
- ECX.1

Temporary
- EDX.1

Known elements
- 0
- 1
- mov
Recovering debug information

MAP inference

<table>
<thead>
<tr>
<th>EDX.2</th>
<th>EDX.3</th>
<th>weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>f4</td>
<td>p p</td>
<td>0.4</td>
</tr>
<tr>
<td>f5</td>
<td>i i</td>
<td>0.3</td>
</tr>
<tr>
<td>f6</td>
<td>i j</td>
<td>0.2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>EDX.3</th>
<th>ECX.1</th>
<th>weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>f1</td>
<td>i n</td>
<td>0.5</td>
</tr>
<tr>
<td>f2</td>
<td>p s</td>
<td>0.3</td>
</tr>
<tr>
<td>f3</td>
<td>a b</td>
<td>0.1</td>
</tr>
</tbody>
</table>

cond-NE-EDX-ECX

dep-EDX-EDX

1 EDX.2 EDX.3 weight
1 i i i 0.8
1 j i i 0.6
1 p p p 0.3
Recovering debug information

Stripped binary

Registers / mem offsets
- EDX.2
- EDX.3
- EDX.1
- ECX.1

Semantic variables
- EDX.2
- EDX.3
- ECX.1

Known elements
- 0
- 1

Semantics variables
- Temporary
- EDX.1

Known elements
- 0
- 1

Debug information

<table>
<thead>
<tr>
<th>Loc</th>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>sum</td>
<td>int</td>
</tr>
<tr>
<td></td>
<td>n</td>
<td>uint</td>
</tr>
<tr>
<td></td>
<td>i</td>
<td>uint</td>
</tr>
<tr>
<td></td>
<td>res</td>
<td>int</td>
</tr>
</tbody>
</table>
DeBIN implementation
DeBIN implementation

Static analysis: BAP
https://github.com/BinaryAnalysisPlatform/bap/

Learning and inference
http://scikit-learn.org
http://nice2predict.org

830 Linux packages
x86, x64, ARM

https://debin.ai
DeBIN evaluation

1. How accurate is DeBIN’s variable recovery?
2. How accurate is DeBIN’s name and type prediction?
3. Is DeBIN useful for malware inspection?
Variable recovery accuracy

DeBIN recovers variables with nearly 90% accuracy

\[
\text{Accuracy} = \frac{|TP| + |TN|}{|sem| + |tmp|} = \frac{\text{+}}{\text{+}}
\]

Results

<table>
<thead>
<tr>
<th>Arch</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>x86</td>
<td>87.1%</td>
</tr>
<tr>
<td>x64</td>
<td>88.9%</td>
</tr>
<tr>
<td>ARM</td>
<td>90.6%</td>
</tr>
</tbody>
</table>

Predicted as semantic registers and memory offsets
Name and type prediction accuracy

Precision = \frac{|CP|}{|PN|} = \frac{\text{Correctly predicted names and types}}{\text{Predicted names and types}}

Recall = \frac{|CP|}{|P|} = \frac{\text{Correct Predictions (CP)}}{\text{Total names and types (P)}}

F1 = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}
Evaluation of name and type prediction

<table>
<thead>
<tr>
<th>Arch</th>
<th>Precision</th>
<th>Recall</th>
<th>F1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>62.6</td>
<td>62.5</td>
<td>62.5</td>
</tr>
<tr>
<td>Type</td>
<td>63.7</td>
<td>63.7</td>
<td>63.7</td>
</tr>
<tr>
<td>Overall</td>
<td>63.1</td>
<td>63.1</td>
<td>63.1</td>
</tr>
<tr>
<td>Name</td>
<td>63.5</td>
<td>63.1</td>
<td>63.3</td>
</tr>
<tr>
<td>Type</td>
<td>74.1</td>
<td>73.4</td>
<td>73.8</td>
</tr>
<tr>
<td>Overall</td>
<td>68.8</td>
<td>68.3</td>
<td>68.6</td>
</tr>
<tr>
<td>Name</td>
<td>61.6</td>
<td>61.3</td>
<td>61.5</td>
</tr>
<tr>
<td>Type</td>
<td>66.8</td>
<td>68.0</td>
<td>67.4</td>
</tr>
<tr>
<td>Overall</td>
<td>64.2</td>
<td>64.7</td>
<td>64.5</td>
</tr>
</tbody>
</table>

Consistent precision/recall of roughly 65%
Malware inspection

We inspected 35 x86 malware samples from VirusShare

Manipulating DNS settings

```c
int sub_80534BA() {
    ...
    if ( dword_8063320 <= 0 ) {
        v1 = ("/etc/resolv.conf", 'r');
        if (v1 || (v1 = sub_8053B1("resolv.conf"))){
            ...
            ...
        }
    }
}
```

```c
int rfc1035_init_resolv() {
    ...
    if ( num_entries <= 0 ) {
        v0 = ("/etc/resolv.conf", 'r');
        if (v0 || (v0 = fopen64("resolv.conf"))){
            // code to read and
            // manipulate DNS settings
        }
    }
}
```

Leakage of sensitive data

```c
If ( sub_806d9f0(args) >= 0 ) {
    ...
    sub_80522B0(args);
    ...
}
```

```c
If ( setsockopt(args) >= 0 ) {
    ...
    sendto(args);
    ...
}
```
Summary

Try online: https://debin.ai

Two-stage prediction process

High precision and accuracy

DEBIN
Predicting Debug Information in Stripped Binaries

DEBIN uses machine learning to recover debug information e.g., names and types of stripped binaries (ELF, DEX, ARM). This is helpful for various binary analysis tasks such as decompilation, malware inspection and similarity.

Registers / mem offsets

<table>
<thead>
<tr>
<th>EDX.2</th>
<th>EDX.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>EDX.1</td>
<td>ECX.1</td>
</tr>
</tbody>
</table>

Unknown variables

| EDX.2 | EDX.3 | ECX.1 |

Temporary

| EDX.1 |

Known elements

| 0     | 1     | mov  |

| i     | n     |

| i     | n     |

i ≈ 65%

Loc | Name | Type |
---|------|------|
| sum | int  |      |
| n   | uint |      |
| i   | uint |      |
| res | int  |      |

Unknown variables

| EDX.3 | ECX.1 |

Known elements

| EDX.2 | EDX.3 | ECX.1 |

High precision and accuracy

Try online: https://debin.ai