BinComp: A Stratified Approach to Compiler Provenance Attribution

Ashkan Rahimian, Paria Shirani, Saed Alrabaee, Lingyu Wang, Mourad Debbabi
Agenda

- Introduction
- Use cases
- Motivating Example
- BinComp Approach
- Evaluation
- Comparison
- Discussion and Limitations
Introduction

- Compiler provenance attribution encompasses
  - Compiler family
  - Compiler version
  - Optimization level
  - Compiler-related functions

- Why compiler provenance attribution?
  - It is a necessary component at pre-processing stage for binary analysis
  - It is important in digital forensics
    - It provides information about the process by which a malware binary is produced
Use Cases

- Compiler identification can be useful for
  - Tool chain malware analysis in binaries
    - Clustering malware samples based on families (assuming each family is using a specific compiler/linker setting)
    - Reasoning about the evolution of malware binaries
  - Library identification, authorship attribution, clone detection, etc.
Motivating Example

```c
#include <stdio.h>
int main()
{
    return 0;
}
```

- Total: 34 functions
  - User function(s): 1
    - Compiler functions: 28
    - Library functions: 5

- Total: 73 functions
  - User function(s): 3
    - Compiler functions: 52
    - Library functions: 18

- Total: 134 functions
  - User function(s): 35
    - Compiler functions: 34
    - Library functions: 65
Extract Compiler-related Functions

GCC

['
'_p_environ', '__strlen', '__glob_match', '__rewinddir',
'_mingwthr_run_key_dtors.part.0', '__cexit', '__strcpy', '__signal',
'_memcpy', '_GetCommandLineA@0', '__pei386_runtime_relocator',
'_errno', '__mingw_globfree', '__register_frame_ctor', '__readdir',
'_setmode', '_puts', '_SetUnhandledExceptionFilter@4', ...
]

VC++

['
'_mainCRTStartup', '__ValidateImageBase', '?
__CxxUnhandledExceptionFilter@@YGPJPAU_EXCEPTION_POINTERS@Z', '__dllonexit', '_controlfp_s', '_invoke_watson', '_amsig_exit', '?
terminate@@YAXXZ', '__setdefaultprecision', '__SEH_epilog4', '_lock',
'_onexit', 'start', '_pre_cpp_init', '_XcptFilter', '_init', ...
]

ICC

['
_cxa_atexit', '.strncat', 'getenv@@GLIBC_2.2.5',
'_ZNSt8ios_base4InitD1Ev', '__intel_cpu_features_init', 'setenv',
'_ZNSt8ios_base4InitC1Ev', '.fprintf', '.catopen',
'catopen@@GLIBC_2.2.5', '.term_proc', 'vsprintf@@GLIBC_2.2.5',
'_start', '__strlen', '__strcpy', '__strlen', ...
]
BinComp Approach

- A layered approach
  - Compiler identification (syntax features)
  - Labeling compiler functions (semantic features)
  - Version & optimization level detection (structural & semantic features)
1st layer: Compiler Identification

- Collecting a set of known source code
- Observing the compiled outputs
- Extracting the features
  - CTP (Compiler Transformation Profile)
  - CT (Compiler Tags)
- Exact matching
1\textsuperscript{st} layer: Compiler Identification

- **CTP (Compiler Transformation Profile)**
  - How source-level data and control structures are reflected in the assembly output
    - For example, corresponding assembly code of \texttt{if/else} compiled with VS will be \texttt{cmp} or \texttt{test}, then \texttt{jcc}.

- **CT (Compiler Tag)**
  - Compilers may embed certain tags in the form of strings or constants
    - For example, GCC writes a \texttt{.comment} section that contains the GCC version string
2nd layer: Compiler Function Labeling

- Signature generation
  - Extracting CF (Compiler Functions)
    - Numerical vectors such as number of instructions, type of registers, etc.
    - Symbolic vectors: such as function names, function prototypes, etc.

- Signature detection
  - Computing the similarity

---

| Function Name, ne Demangled Name, i Imported Function, k Call List (From), ke Demangled Calls, ix Number of Imported Functions, kx Number of Calls |
|---|---|---|---|---|
| c Constant, s String, cx Number of Constants, sx Number of Strings |
| p Function Prototype, a Function Argument, r Return Type, g Number of a, b Size of Arguments, gx RES. Number of g |
| m Number of Instructions, l Size of Local Variables, f Function Flags, o Code References (From), z Function Size, mx RES. Number of m, ox Number of o, cc Cyclomatic Complexity, bx Number of Basic Blocks |
| d Dictionary (Malware Tag), t API Tag, dx Number of d, tx Number of t |
| av Anti-VM, reg General Register, mem Memory Reference, bix [Base +Index], bid [Base+Index+Displacement], imm Immediate, ifa Immediate Far Address, ina Immediate Near Address, fpp FPP Register, ctr Control Register, dbr Debug Register, trr Trace Register |
| DTR Data Transfer, DTO Data Transfer Address Object, FLG Flag Manipulation, DTC Data Transfer Conversion, ATH Binary Arithmetic, LGC Logical, CTL Control Transfer, INO Input Output, INT Interrupt and System, FLT Floating, MSC Misc. |
2nd layer: Compiler Function Labeling

- Example of CF features

<table>
<thead>
<tr>
<th>COMP</th>
<th>OPL</th>
<th>Symbolic Function ID</th>
<th>DTR, DTO, FLG, ATH, LGC, CTL, INO, INT, FLT, REG, MEM, IMM, IFA, INA</th>
</tr>
</thead>
<tbody>
<tr>
<td>VS</td>
<td>OP2</td>
<td>@__security_check_cookie@4</td>
<td>0, 0, 0, 0, 0, 4, 0, 0, 0, 001, 001, 000, 000, 002</td>
</tr>
<tr>
<td>VS</td>
<td>OP2</td>
<td>__mainCRTStartup</td>
<td>3, 0, 1, 2, 1, 6, 0, 0, 0, 077, 028, 017, 000, 025</td>
</tr>
<tr>
<td>GCC</td>
<td>OP0</td>
<td>__mingw_CRTStartup</td>
<td>3, 1, 1, 3, 6, 7, 0, 0, 0, 216, 015, 068, 000, 071</td>
</tr>
<tr>
<td>GCC</td>
<td>OP0</td>
<td>gcc_register_frame</td>
<td>3, 0, 1, 1, 0, 3, 0, 0, 0, 029, 001, 014, 000, 009</td>
</tr>
</tbody>
</table>

Instruction Categories.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DTR</td>
<td>Data Transfer</td>
<td>INA</td>
<td>Indirect Near Address</td>
</tr>
<tr>
<td>INO</td>
<td>Input/Output</td>
<td>DTO</td>
<td>Data Transfer Object</td>
</tr>
<tr>
<td>FLT</td>
<td>Float Point</td>
<td>FLG</td>
<td>Flag Manipulation</td>
</tr>
<tr>
<td>REG</td>
<td>Registers</td>
<td>LGC</td>
<td>Logical Instructions</td>
</tr>
<tr>
<td>MEM</td>
<td>Memory</td>
<td>CTL</td>
<td>Control Instructions</td>
</tr>
<tr>
<td>IMM</td>
<td>Immediate Value</td>
<td>IFA</td>
<td>Indirect Far Address</td>
</tr>
<tr>
<td>INT</td>
<td>Interrupt/System</td>
<td>ATH</td>
<td>Arithmetic Instructions</td>
</tr>
</tbody>
</table>

\[ d_J(t_i, t_j) = \frac{S(t_i \land t_j)}{S(t_i \lor t_j)} \]

t_i, t_j: fingerprint vectors generated from the candidate function pairs
3rd layer: Version & Optimization

- Extracting the features
  - ACFG (Annotated Control Flow Graph)
  - CCT (Compiler Constructor Terminator)
**3rd layer: Version & Optimization**

- According to our experiments ACFG and CCT are the best features to detect the version and optimization
  - The different versions can affect the ACFG
  - The different optimization levels affect the CCT
    - For instance, the CCT for full optimized is a subset of CCT for the no-optimization code

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Version</th>
<th>Optimization</th>
</tr>
</thead>
<tbody>
<tr>
<td>GCC</td>
<td>3.4</td>
<td>00</td>
</tr>
<tr>
<td></td>
<td>4.4</td>
<td>02</td>
</tr>
<tr>
<td>ICC</td>
<td>10</td>
<td>00</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>02</td>
</tr>
<tr>
<td>VS</td>
<td>2010</td>
<td>00</td>
</tr>
<tr>
<td></td>
<td>2012</td>
<td>02</td>
</tr>
<tr>
<td>XCODE</td>
<td>5.1</td>
<td>00</td>
</tr>
<tr>
<td></td>
<td>6.1</td>
<td>02s</td>
</tr>
</tbody>
</table>
Evaluation

- **Dataset**
  - Four free open-source projects (SQLite, zlib, libpng, and openSSL)
  - Google Code Jam (232 files)
  - Students Code Projects (993 files)
## Evaluation

### Results

#### Accuracy for variations of compiler versions.

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Version</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>GCC</td>
<td>3.4.x</td>
<td>86%</td>
</tr>
<tr>
<td></td>
<td>4.4.x</td>
<td>89%</td>
</tr>
<tr>
<td>ICC</td>
<td>10.x</td>
<td>83%</td>
</tr>
<tr>
<td></td>
<td>11.x</td>
<td>90%</td>
</tr>
<tr>
<td>VS</td>
<td>2010</td>
<td>70%</td>
</tr>
<tr>
<td></td>
<td>2012</td>
<td>71%</td>
</tr>
<tr>
<td>XCode</td>
<td>5.x</td>
<td>78%</td>
</tr>
<tr>
<td></td>
<td>6.1</td>
<td>74%</td>
</tr>
</tbody>
</table>

#### Accuracy for variations of compiler optimization levels.

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Optimization</th>
<th>Average Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>GCC</td>
<td>O0, O2</td>
<td>91%</td>
</tr>
<tr>
<td>ICC</td>
<td>O0, O2</td>
<td>89%</td>
</tr>
<tr>
<td>VS</td>
<td>O0, O2</td>
<td>95%</td>
</tr>
</tbody>
</table>
## Comparison

<table>
<thead>
<tr>
<th></th>
<th>IDA Pro Disassembler</th>
<th>Rosenblum et al</th>
<th>BinComp</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Features</strong></td>
<td>Entry point signatures</td>
<td>Syntax (n-gram)</td>
<td>Syntax, Semantic, Structural</td>
</tr>
<tr>
<td><strong>Detection Method</strong></td>
<td>Signature based</td>
<td>Classification</td>
<td>Exact/Inexact matching</td>
</tr>
<tr>
<td><strong>Compilers</strong></td>
<td>VS, Delphi, Fortran</td>
<td>GCC, VS, ICC</td>
<td>VS, GCC, ICC, XCODE</td>
</tr>
<tr>
<td><strong>Required data set</strong></td>
<td>Small</td>
<td>Large</td>
<td>Small</td>
</tr>
<tr>
<td><strong>Scalability</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Time efficient</strong></td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Identifying Ver.</strong></td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Identifying Opt.</strong></td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
</tr>
</tbody>
</table>
Discussion and Limitations

- BinComp requires few data set and it can be applied for any compiler
- BinComp is efficient in terms of time and scalability
- Limitations
  - The binary code is deobfuscated
  - Only Intel x86/x86-64 architecture is considered
Thank You!
Evaluation

- Accuracy
  - Precision (P)
  - Recall (R)

\[
P = \frac{TP}{TP + FP}
\]

\[
R = \frac{TP}{TP + FN}
\]

- Our application domain is much more sensitive to false positives than false negatives

\[
F_{0.5} = 1.25 \cdot \frac{PR}{0.25P + R}
\]
Neighbor hash graph kernel (NHGK)

- Condense the information contained in a neighborhood into a single hash value
- Label each node in the function call graph
  - each function is characterized by its numerical and symbolic feature vectors

Our method strives to model the composition of functions
- The neighborhood of a function must be taken into account.
- We compute a neighborhood hash over all of its direct neighbors in the function call graph

$$h(f_i) = \text{shr}_1(G(f_i)) \oplus \left( \bigoplus_{f_j \in N_{f_i}} G(f_j) \right)$$

- $\text{shr}_1$: a one-bit shift right operation
- $\oplus$: a bit-wise XOR on the binary labels