463.15 Code Stylometry

Computer Security II
CS463/ECE424
University of Illinois
Outline

• Stylometry and authorship attribution background

• Code stylometry methods
  – Source code stylometry
  – Executable binary stylometry
Motivating Examples

• There has been debate over who wrote:
  – Shakespeare’s works
  – Bible passages
  – The Federalist Papers
  – The Unabomber’s Manifesto
We can infer the identity of an author of a document by examining it.

**Stylometry:** inferring properties of the author by examination

- This idea is over a century old
- Stylome/fingerprint: differences in how individuals write
Linguistic Stylometry

• Use different features of written text to fingerprint authors
  – Vocabulary
  – Average word length
  – Frequency of specific words
  – Many others

• Machine learning is generally used to classify works based on these features
Examples of Linguistic Stylometry

• [Narayanan12] used stylometry to identify anonymous bloggers in large datasets
  – This is a privacy issue

• Adversarial stylometry [Brennan12]
  – Authorship attribution based on linguistics can be evaded
  – Defenses:
    • Obfuscate writing style
    • Imitate someone else’s writing style
Code Stylometry

• We want to determine who wrote some code
• Goal: programmer de-anonymization
• Can you think of reasons why we would want to determine code authorship?
We want to determine who wrote some code
Goal: programmer de-anonymization
Can you think of reasons why we would want to determine code authorship?

- Company wants to determine which employee wrote harmful code
- Government wants to determine who is engaging in cyber warfare
- A professor wants to determine if students are plagiarizing assignments
- Identify Satoshi Nakomoto
- Identify cyber criminals
- Determine source of malware
- Reveal creators of anti-censorship tools
Types of Code Stylometry

- Source code stylometry
- Executable binary stylometry
- Malware attribution
We can study source code for authorship attribution

Examples of features used for source code stylometry:
- Simple byte-level and word-level n-grams
- Abstract syntax trees
- Lexical markers such as line length
- Layout

Techniques usually include classification by ML
We want to study executable binaries for authorship attribution.

Binaries are typically produced by compiling or assembling source code.

Goal: perform stylometry on executable binaries.
Harder than source code stylometry

During compilation,

– Variable names, function names, and other symbols and metadata about the source code can be removed
– The structure of the code can be changed through optimization

This removes information that may suggest authorship
What information can we use about binary code to reveal authorship information?

- Use tools to parse executable binaries
- Reconstruct instruction sequences and control flow graphs
- Use this information as features to determine a code author’s stylometric fingerprint
• **Goal:** executable binary stylometry using automatic tools

• **Main idea:**
  
  – Use *machine learning* to classify sample executable binaries from a set of known authors
  
  – Determine a good set of *features* for executable binary stylometry

[Caliskan18]
Consider an analyst interested in determining the author of an executable binary purely based on its style (not content)

Assume that the analyst only has access to executable binary samples each assigned to one of a set of candidate programmers

The analyst:

- Obtains labeled executable binary samples from each candidate programmer (training set)
- Converts each labeled sample into numerical feature vector
  - Obtains this by using low-level features from disassemblers and decompilers
- Derives a classifier from these vectors using machine learning
- Uses this classifier to attribute the anonymous executable binary (test sample) to the most likely programmer
Disassemblers

– Programs that translate executable binary code into assembly code
– The inverse of an assembler

Decompilers

– Programs that translate executable binary code into high level source code
– The inverse of a compiler

These tools do not perfectly reconstruct the original source or assembly code
A *control flow graph* is a graph of all paths that might be traversed through a program during execution.

Each *node* represents a basic block in the code:
- A basic block is a piece of code with no jumps.

*Directed edges* represent jumps in the control flow.
Background: Abstract Syntax Trees

• Tree representation of the **abstract syntactic structure** of source code written in a programming language
  – i.e. a structure containing only the meaning of a program, but no language details (ex. semicolons, spaces, formatting)

• Each **node** of the tree denotes a construct that occurs in the source code

• These trees **abstract away** certain parts of the high-level language such as: parentheses, if statements, etc
Background: AST and CFG examples

Abstract syntax tree (AST)

\[
\begin{align*}
&\text{func} \\
&\quad \text{decl} \\
&\quad \quad \text{int} \quad = \quad \text{call} \\
&\quad \quad \quad \quad \text{v0} \\
&\quad \quad \quad \quad \text{f0} \\
&\quad \quad \quad \quad \text{v0} \\
&\quad \quad \quad \quad \text{C0} \\
&\quad \text{if} \\
&\quad \quad \text{pred} \quad \text{stmt} \\
&\quad \quad \quad \text{<} \\
&\quad \quad \quad \quad \text{...} \\
\end{align*}
\]

Control-flow graph (CFG)

\[
\begin{align*}
&\text{entry} \\
&\quad \text{blk1} \\
&\quad \quad \text{blk2} \quad \text{blk3} \\
&\quad \quad \quad \text{blk4} \\
&\quad \quad \quad \text{exit}
\end{align*}
\]
• **Executable Binary Stylometry** [Caliskan18]
  
  – To extract the features of executable binary code for stylometry:
    
    • Use automated decompilation of binaries
    • Generate abstract syntax trees of decompiled source code
    • Use multiple tools for disassembly and decompilation in parallel

  – **ML framework**
    
    • Feature reduction
    • Make predictions about code authorship using a random forest classifier
[Caliskan18] Overview
Stylistic Features

• Representations of the program from binary code
  – Disassembler
    • Obtain low level features in assembly code
    • Based on machine code instructions, referenced strings, symbol information, and control flow graphs
  – Decompiler
    • Translate the program into C-like pseudo code
    • Pass this code to a fuzzy parser for C
    • Generate control flow graph to capture the flow of the program
    • Convert the low level instructions to high level decompiled source code in order to obtain abstract syntax trees
• Use these three data formats to numerically represent the stylistic properties embedded in binary code
Dimensionality Reduction

• Analyst determines the set of stylistic features through dimensionality reduction

• Two steps of feature selection:
  – Information gain based dimensionality reduction
  – Correlation based feature selection

• Select features particularly useful for classification

• 53 features are identified to represent programmer style
  – Out of 705,000 representations of code properties
Machine Learning Task

• Closed world problem
  – The set of potential code authors is known

• Supervised learning task
  – The training data is labeled

• Multi-class problem
  – Classifier calculates the most likely author for the anonymous executable binary sample among multiple code authors
Experimental Setup

- [Caliskan18] performs experiments with data from the Google Code Jam
  - GCJ is an annual programming competition
  - Contestants implement solutions for the same tasks
- Focused on C++ code
- Compiled with gcc or g++
  - Experimented with no optimizations, and optimization levels- 1, 2, and 3
### Results

<table>
<thead>
<tr>
<th>Related Work</th>
<th>Number of Programmers</th>
<th>Number of Training Samples</th>
<th>Accuracy</th>
<th>Classifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rosenblum [39]</td>
<td>20</td>
<td>8-16</td>
<td>77%</td>
<td>SVM</td>
</tr>
<tr>
<td>This work</td>
<td>20</td>
<td>8</td>
<td>90%</td>
<td>SVM</td>
</tr>
<tr>
<td>This work</td>
<td>20</td>
<td>8</td>
<td>99%</td>
<td>RF</td>
</tr>
<tr>
<td>Rosenblum [39]</td>
<td>100</td>
<td>8-16</td>
<td>61%</td>
<td>SVM</td>
</tr>
<tr>
<td>This work</td>
<td>100</td>
<td>8</td>
<td>84%</td>
<td>SVM</td>
</tr>
<tr>
<td>This work</td>
<td>100</td>
<td>8</td>
<td>96%</td>
<td>RF</td>
</tr>
<tr>
<td>Rosenblum [39]</td>
<td>191</td>
<td>8-16</td>
<td>51%</td>
<td>SVM</td>
</tr>
<tr>
<td>This work</td>
<td>191</td>
<td>8</td>
<td>81%</td>
<td>SVM</td>
</tr>
<tr>
<td>This work</td>
<td>191</td>
<td>8</td>
<td>92%</td>
<td>RF</td>
</tr>
<tr>
<td>This work</td>
<td>600</td>
<td>8</td>
<td>71%</td>
<td>SVM</td>
</tr>
<tr>
<td>This work</td>
<td>600</td>
<td>8</td>
<td>83%</td>
<td>RF</td>
</tr>
</tbody>
</table>

**TABLE II: Comparison to Previous Results**
Fig. 5: Large Scale Programmer De-anonymization
Findings

• Even a single training sample per programmer is sufficient for de-anonymization
• Accuracy can be improved by finding the top-n most likely authors
• The feature set selected works across different sets of programmers
• This work can de-anonymize 600 programmers from their executable binaries
• Removing symbol information does not anonymize binaries
• Programmers can be de-anonymized from obfuscated binaries*
Practical Implications of this Work

• Coding style survives compilation!

• Why?
  – Decompiled source code is not necessarily similar to the original source code in terms of the features used in this work
  – The feature vector obtained from disassembly and decompilation can be used to predict the features in the original source code

• More skilled programmers are more fingerprintable
  – Programmers gradually acquire their own unique style as they gain experience
References


Discussion

• Can you think of some countermeasures that might be possible to preserve privacy against code stylometry analysis?

• What are the pros and cons of authorship attribution?
  – Natural language
  – Code