When Coding Style Survives Compilation: De-anonymizing Programmers from Executable Binaries

Aylin Caliskan[*], Fabian Yamaguchi[†], Edwin Dauber[‡], Richard Harang[§], Konrad Rieck[¶], Rachel Greenstadt[‡], Arvind Narayanan[*]

[*]Princeton University, [†]Shiftleft Inc, [‡]Drexel University, [§]Sophos, Data Science Team, [¶]TU Braunschweig

NDSS 2018

Summarize the paper

- Problem
  - Programmer de-anonymization has implications for privacy and anonymity
  - Previous studies identify programmers using source code with high accuracy
- Contribution
  - Present a novel approach for executable binary authorship attribution
- Result
  - The method overperforms the state-of-the-art in both accuracy and on larger datasets
  - The method is robust to basic obfuscations, compiler optimizations, and stripped binaries
- Meaning
  - The authors show that programmers who would like to remain anonymous need to take extreme countermeasures to protect their privacy

Why de-anonymize programmers?

- To identify malware authors
- To solve the copyright or authorship disputes
- To detect ghostwriting
  - A professor want to determine whether a student's programming assignment has been written by a student who has previously taken the class
- To identify programmers of certain types of programs, such as censorship-circumvention tools
  - In an oppressive regime that prohibits certain types of software, the regime tries to unmask the programmers and might want to punish them

Why de-anonymize programmers?

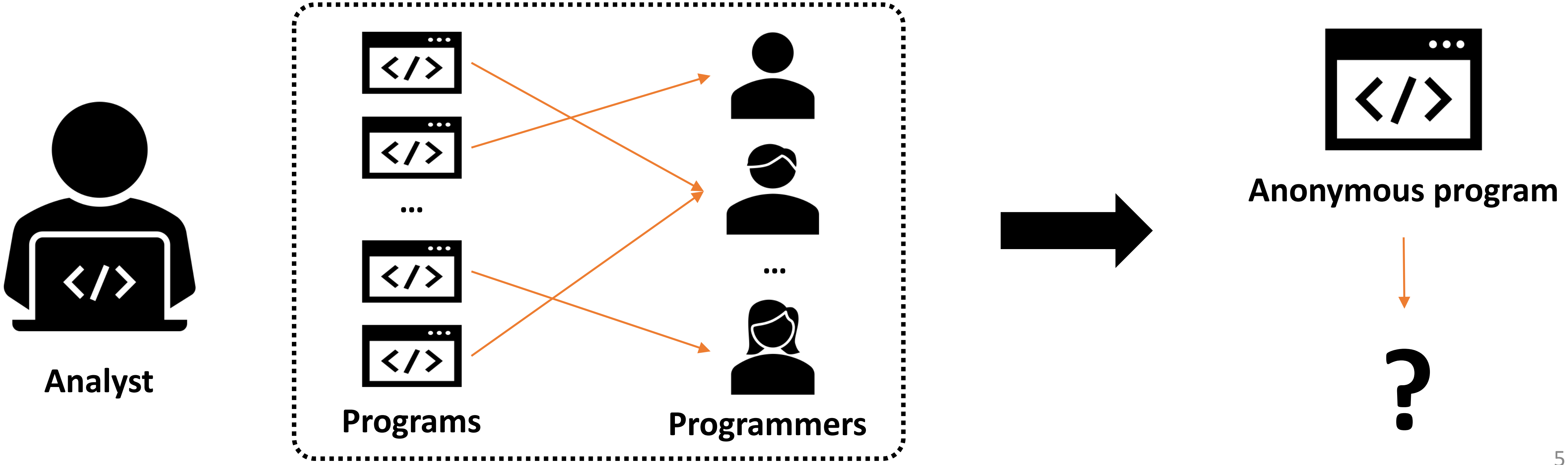
- To identify malware authors
- To solve the copyright or authorship disputes

How can we de-anonymize programmers?

- To identify programmers of certain types of programs, such as censorship-circumvention tools
    - In an oppressive regime that prohibits certain types of software, the regime tries to unmask the programmers and might want to punish them

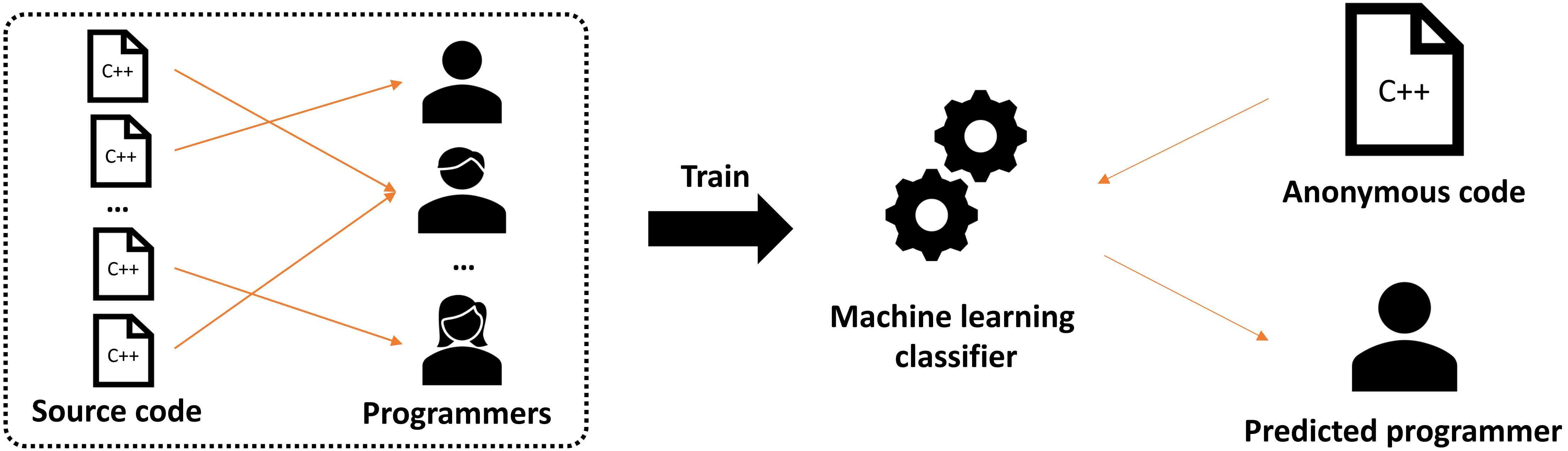
Problem statement

- Consider an analyst insterested in determining the programmer of an anonymous program
- The analyst has access to program samples each assigned to one of a set of candidate programmers



Motives

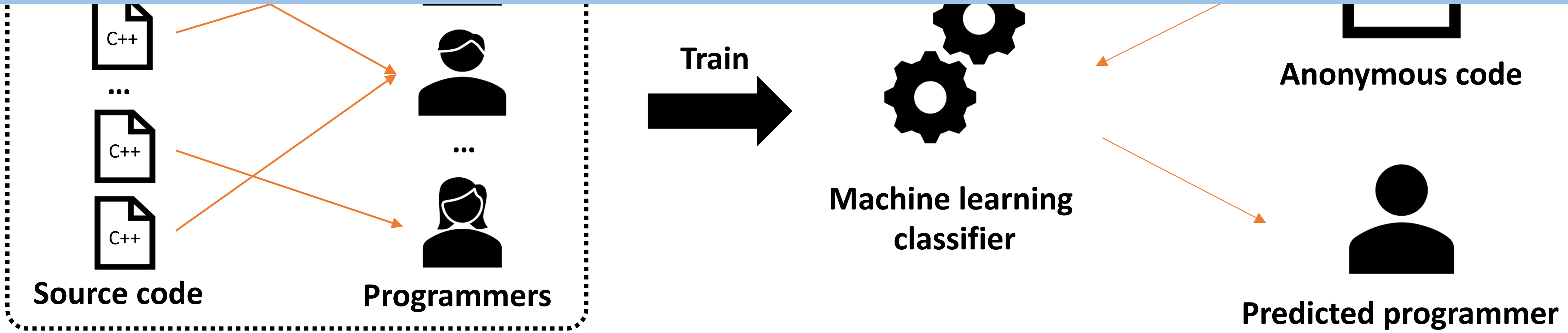
- *De-anonymizing Programmers via Code Stylometry* (USENIX 2015)
  - Present a method to identify programmers of C/C++ source code
  - Extract syntactic, lexical and layout features to investigate style in source code
  - Reach 94% accuracy in classifying 1,600 authors, 98% in 250 authors



Motives

- *De-anonymizing Programmers via Code Stylometry* (USENIX 2015)
  - Present a method to identify programmers of C/C++ source code
  - Extract syntactic, lexical and layout features to investigate style in source code

How about **executable binaries**?



Challenges

- Many distinguishing features present in source code, e.g. variable names, are removed in the compilation process
- Compiler optimization may alter the structure of a program

Source code

```
#include <cstdio>
#include <algorithm>
using namespace std;
#define For(i,a,b) for(int i = a; i < b; i++)
#define FOR(i,a,b) for(int i = b-1; i >= a; i–)
double nextDouble() {
        double x;
        scanf("%lf", &x);
        return x;}
int nextInt() {
        int x;
        scanf("%d", &x);
        return x; }
```

...

Executable binary

```
00100000 00000000 00001000 00000000
00000000 00000000 00110100 00000000
00000100 00001000 00000000 00000001
00000000 00000001 00000000 00000000
00000000 00000000 00000100 00000000
00000011 00000000 00000000 00000000
00000000 00000000 00110100 10000001
00000000 00000000 00010011 00000000
00000100 00000000 00000000 00000000
00000000 00000000 00000001 00000000
00000000 00000000 00000000 00000000
00000100 00001000 00000000 10000000
```

...

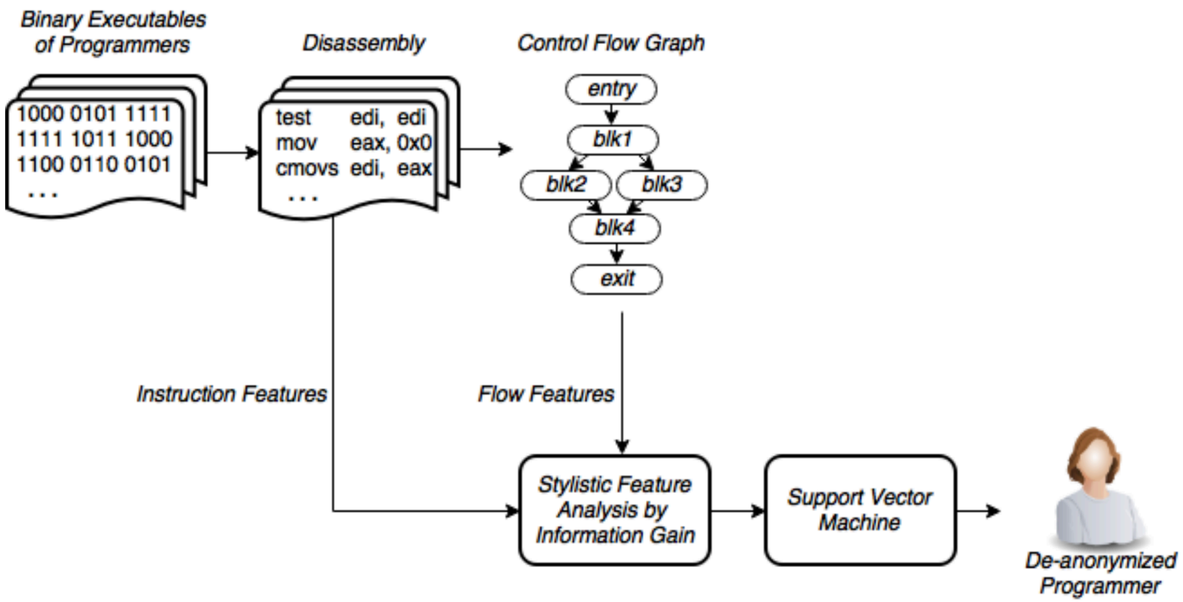
https://www.ndss-symposium.org/wp-content/uploads/2018/03/NDSS2018_06B-2_Caliskan_Slides.pdf

Related work

- Identifying programmers from compiled code has received little attention to date
  - *Oba2: an onion approach to binary code authorship attribution* (Alrabaee, Digital Investigation 2014)
  - *Who wrote this code? Identifying the authors of program binaries* (Rosenblum, ESORICS 2011)

Related work

- *Who wrote this code? Identifying the authors of program binaries* (ESORICS 2011)
  - Use the Paradyn project's Parse API for parse executable binaries to get the instruction sequences and control flow graphs
  - Train a support vector machine
  - Perform evaluation and experiments on controlled corpora that are not noisy



https://www.ndss-symposium.org/wp-content/uploads/2018/03/NDSS2018_06B-2_Caliskan_Slides.pdf

Related work vs. this work

- *Who wrote this code? Identifying the authors of program binaries* (ESORICS 2011)
  - Use the Paradyn project's Parse API for parse executable binaries to get the instruction sequences and control flow graphs

  ➔ Use 4 different resources to parse executable binaries to generate a richer representation (ndisasm, radare2, hex-rays, joern)

  - Train a support vector machine

  ➔ Train a Random forest classifier

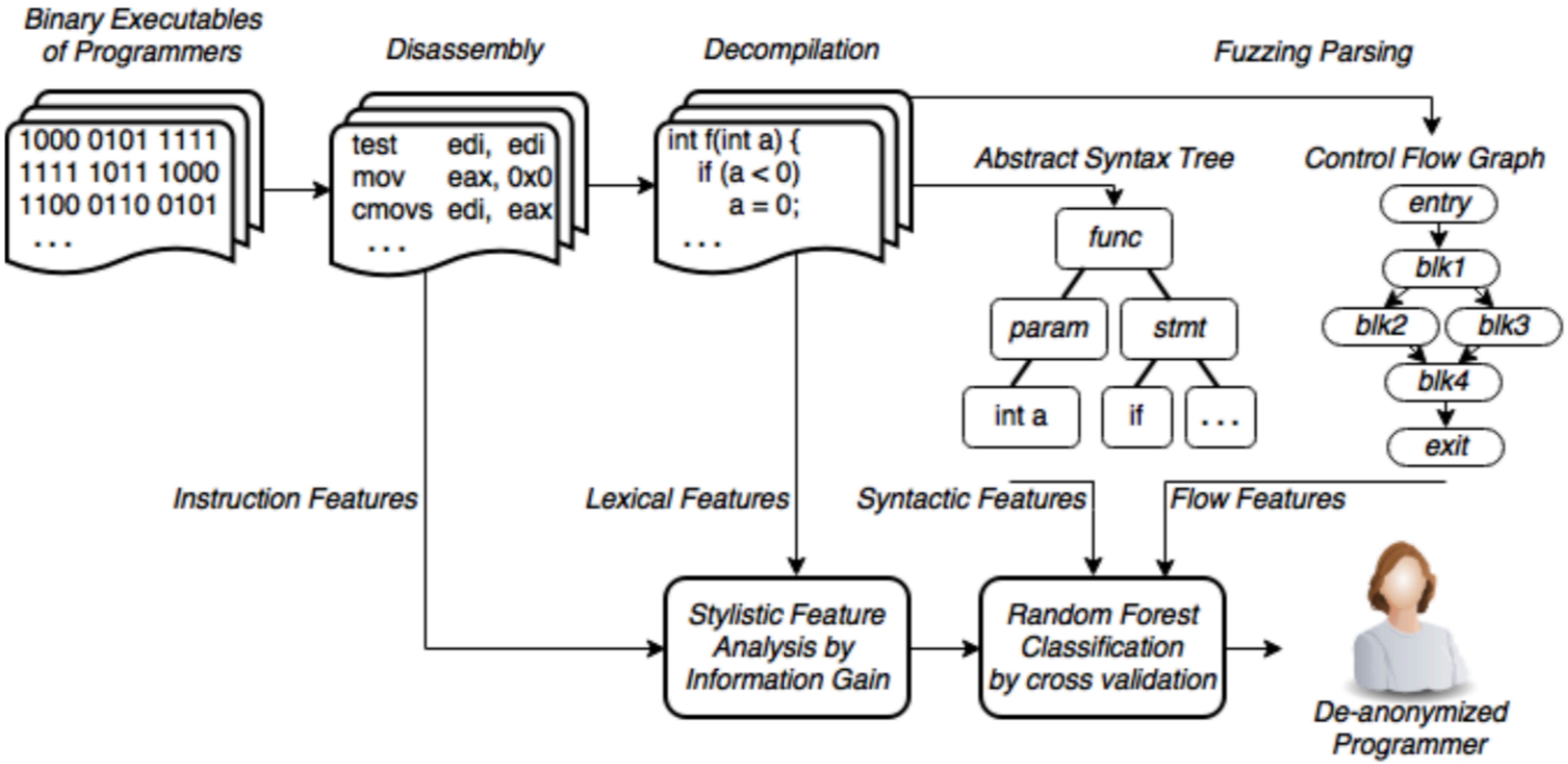
  - Perform evaluation and experiments on controlled corpora that are not noisy

  ➔ In addition, investigate noisy real-world dataset, an open-world setting, and effects of optimizations and obfuscations

Approach

- Automatically recognize programmers of compiled code
- Leverage supervised machine learning
- Consist of the following 4 steps
    1) Feature extraction via disassembly
    2) Feature extraction via decompilation
    3) Dimensionality reduction
    4) Classification

Overview



Binary Executables of Programmers
1000 0101 1111
1111 1011 1000
1100 0110 0101
...
Disassembly
test edi, edi
mov eax, 0x0
cmovs edi, eax
...
Decompilation
int f(int a) {
if (a < 0)
a = 0;
...
Fuzzing Parsing
Abstract Syntax Tree
func
param
stmt
int a
if
...
Control Flow Graph
entry
blk1
blk2
blk3
blk4
exit
Instruction Features
Lexical Features
Syntactic Features
Flow Features
Stylistic Feature Analysis by Information Gain
Random Forest Classification by cross validation
De-anonymized Programmer

Feature extraction via disassembly

- Disassemble the executable binary to extract low-level features
- Use two disassemblers
  - To generate two sets of instructions for each binary
  - Netwide disassembler (ndisasm)
    - Simply decoding the executable binary from start to end
    - No distinction is made between bytes that represent data and bytes that represent code
  - radare2 disassembler
    - Understand the executable binary format -> process relocation, symbol information, and strings referenced in the code
    - Identify functions in code and generate corresponding control flow graphs

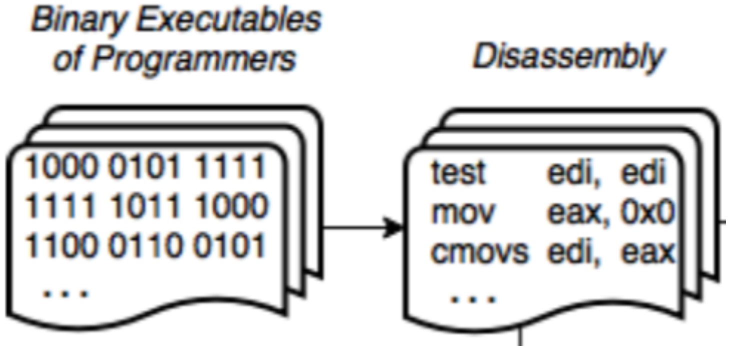
Feature extracti… y

Is it necessary to use two disassemblers?

- Disassemble the executa… tures
- Use two disassemblers
  - To generate two sets of instructions for each binary
  - Netwide disassembler (ndisasm)
    - Simply decoding the executable binary from start to end
    - No distinction is made between bytes that represent data and bytes that represent code
  - radare2 disassembler
    - Understand the executable binary format -> process relocation, symbol information, and strings referenced in the code
    - Identify functions in code and generate corresponding control flow graphs

Instruction features

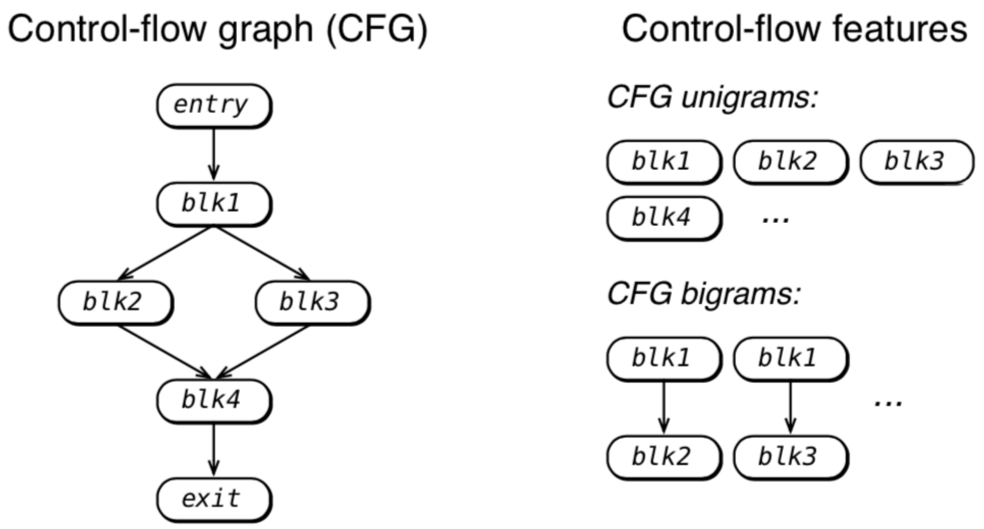
- Strip the hexadecimal numbers from assembly instructions and replace them with the uni-gram number
  - To avoid overfitting that might be caused by unique hexadecimal numbers
- Tokenize the instruction traces and extract token uni-grams, bi-grams, and tri-grams within a single line of assembly
- Extract 6-grams, which span two consecutive lines of assembly
  - A meaningful construct is longer than a line of assembly code



1-grams: test, edi, edi, mov, eax, …
2-grams: test edi, edi edi, mov eax, …
3-grams: test edi edi, mov eax 0x0, …
6-grams: test edi edi mov eax 0x0, …

Flow features

- Extract single basic blocks of radare2's control flow graphs
- Extract pairs of basic blocks connected by control flow

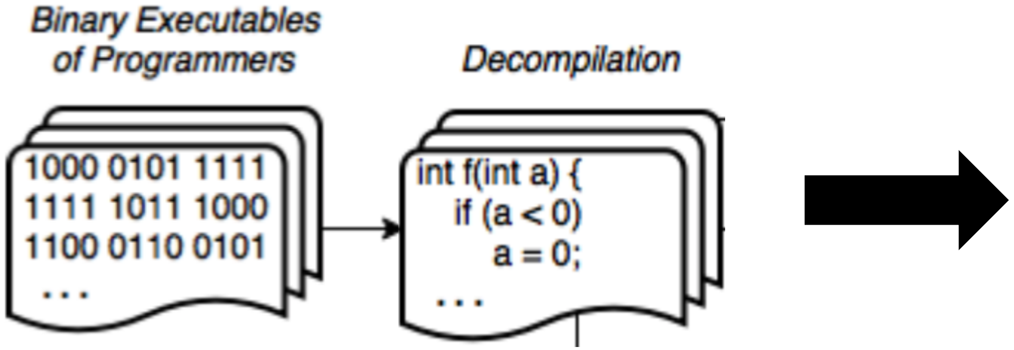


Feature extraction via decompilation

- Decompilers
  - Translate an executable binary to equivalent source code
  - Reconstruct higher level constructs
  - Extract syntactical features of code
- Employ the Hex-Rays decompiler
  - A commercial state-of-the-art decompiler
  - Convert executable programs into a human readable C-like pseudo code
  - Extract two types of features from this pseudo code
    - Lexical features
    - Syntactical features

Lexical features

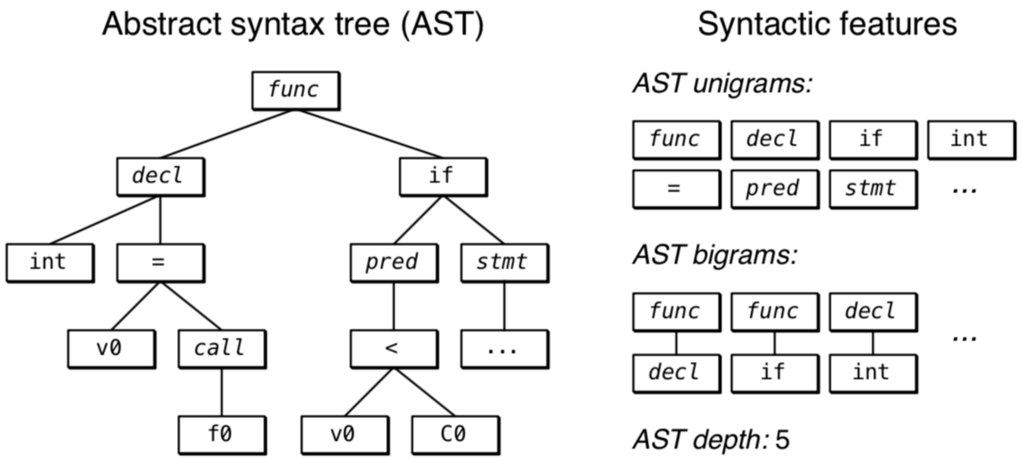
- The world unigrams
- Capture the integer types used in a program, names of library functions, and names of internal functions



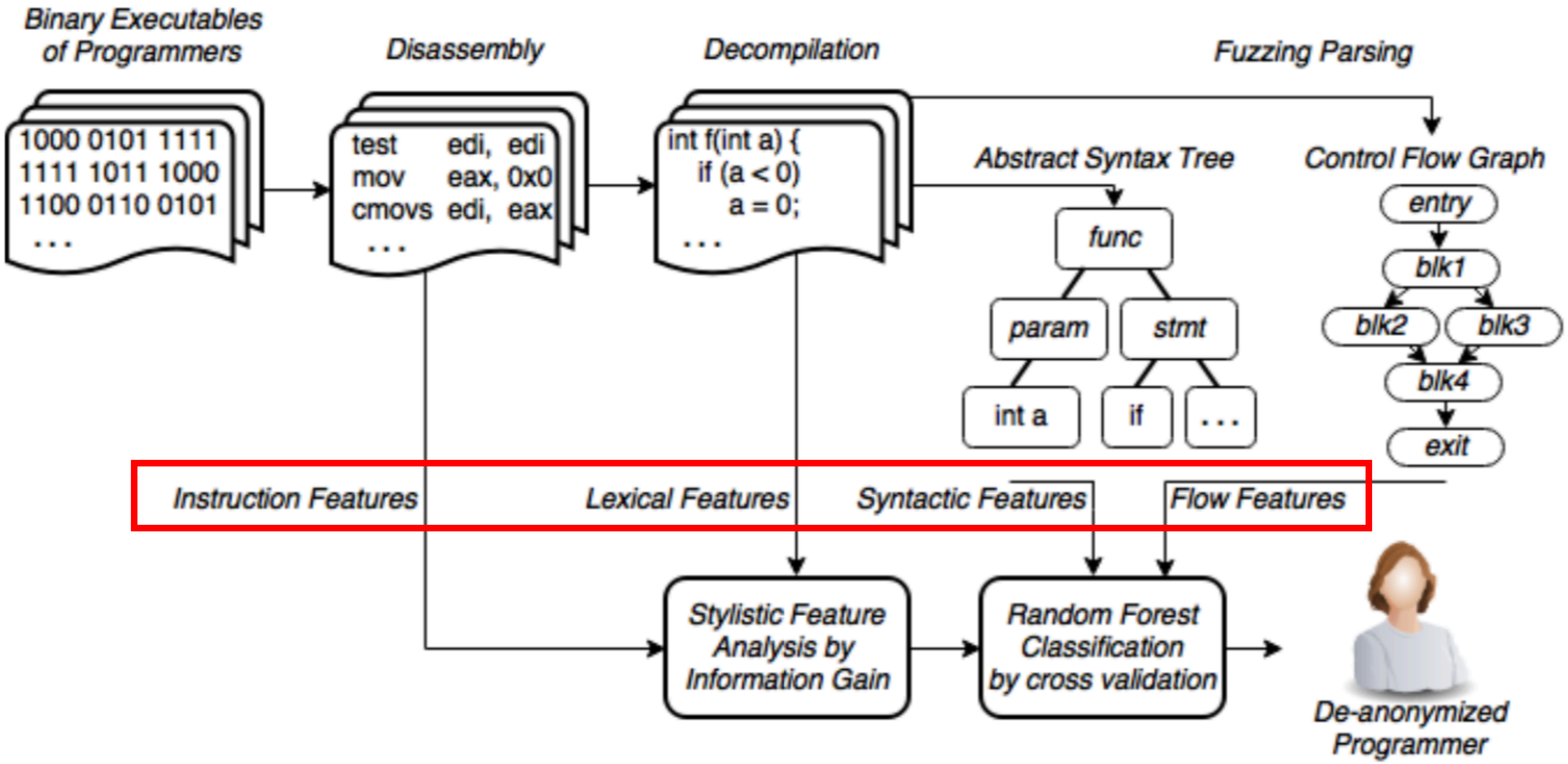
1-grams: 0, f, …

Syntactical features

- Obtain by passing the C-pseudo code to joern
  - A fuzzy parser for C that is capable of producing fuzzy abstract syntax trees (ASTs) from Hex-Rays pseudo code output
- Represent the grammatical structure of the program
- AST node unigrams, labeled AST edges, AST node term frequency inverse document frequency, and AST node average depth



Feature extraction

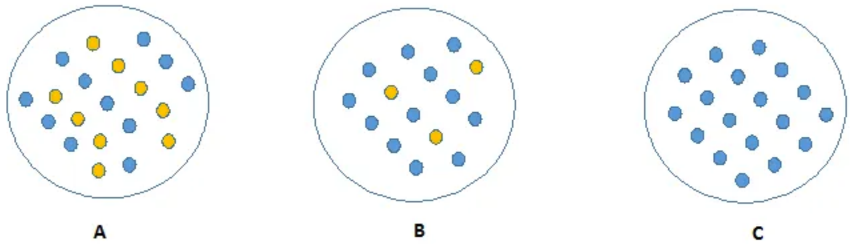


Dimensionality reduction

- Feature extraction produces a large amount of features, resulting in sparse feature vectors with thousands of elements (705,000 features)
- Not all features are equally informative to express a programmer’s style
- Reducing the dimensions of the feature set is important for avoiding overfitting and reducing the computational cost
- Use two dimensionality reduction steps
  - Information gain criterion (less than 2,000 features)
  - Correlation based feature selection (53 features)

Information gain criterion

- Which one can be described easily?
  - C: Require less information as all values are similar (pure)

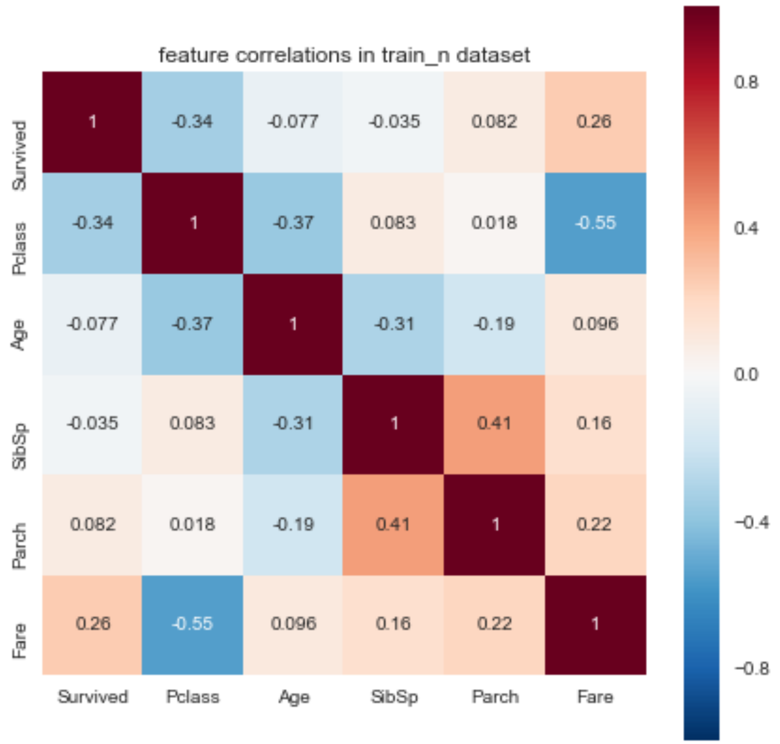


  - less impure node requires less information, more impure node requires more information
- Entropy: a degree of disorganization in a system
  - If the sample is completely homogeneous, then the entropy is zero and if the sample is an equally divided it has entropy of one
- Keep features with low entropy

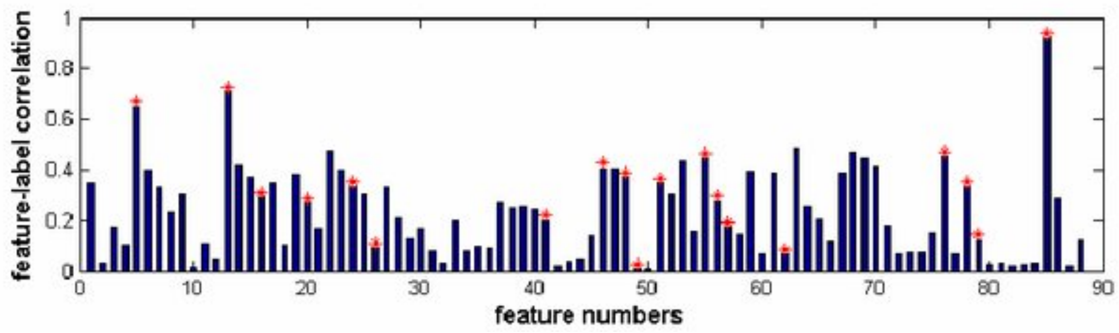
https://www.analyticsvidhya.com/blog/2015/01/decision-tree-algorithms-simplified/?utm_content=buffer47f23&utm_medium=social&utm_source=twitter.com&utm_campaign=buffer

Correlation based feature selection

- feature – feature
  - Keep features with low inter-class correlation



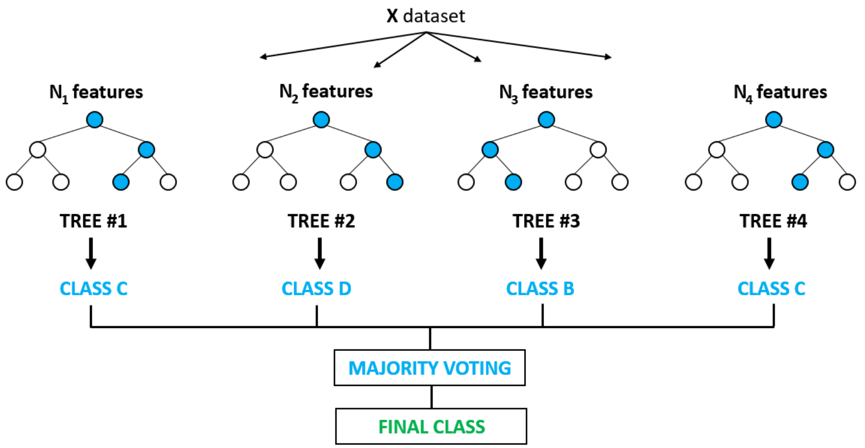
- feature – class
  - Keep features that are highly correlated with classes

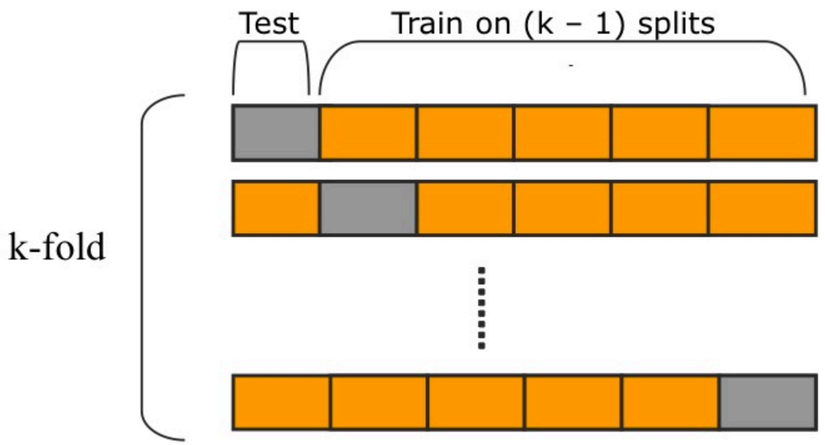


https://speedml.com/plot-correlation-of-features/
https://www.researchgate.net/publication/246175769_Feature_selection_for_partial_discharge_diagnosis/figures?lo=1

Classification

- Use random forests as classifier
  - Employ random forests with 500 trees
- Perform k-fold cross-validation 10 times
  - Split data into training and test sets stratified by author (ensure that # of samples per author in the training and test sets are identical)





https://medium.com/@ar.ingenious/applying-random-forest-classification-machine-learning-algorithm-from-scratch-with-real-24ff198a1c57
http://qingkaikong.blogspot.com/2017/02/machine-learning-9-more-on-artificial.html

Experiments

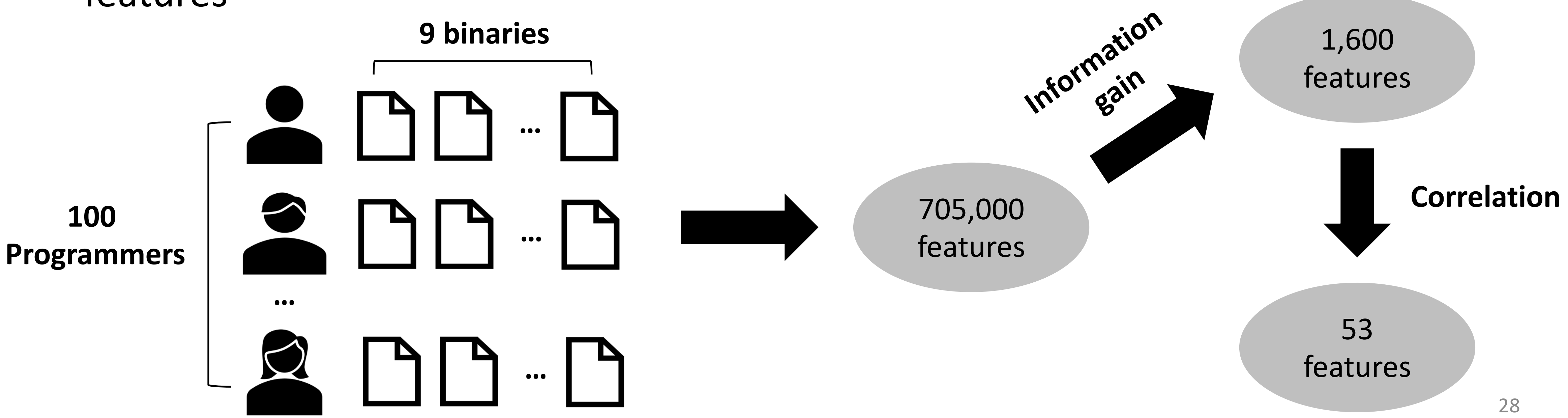
1. Google Code Jam Experiments
2. Real-world scenarios

Google Code Jam dataset

- Google Code Jam (GCJ) is an annual programming competition
  - Enable to directly compare results to previous work
  - Remove the potential confounding effect of identifying programming task rather than programmer by identifying functionality properties instead of stylistic properties
- Collect C++ solutions from the years 2008 to 2014 along with author names and problem identifiers
- Compile the source code with gcc or g++
  - without any optimization, with level-1/level-2/level3 optimizations
- 600 programmers who all have 9 executable binary samples, a total of 5,400 (600*9)

Feature extraction / selection

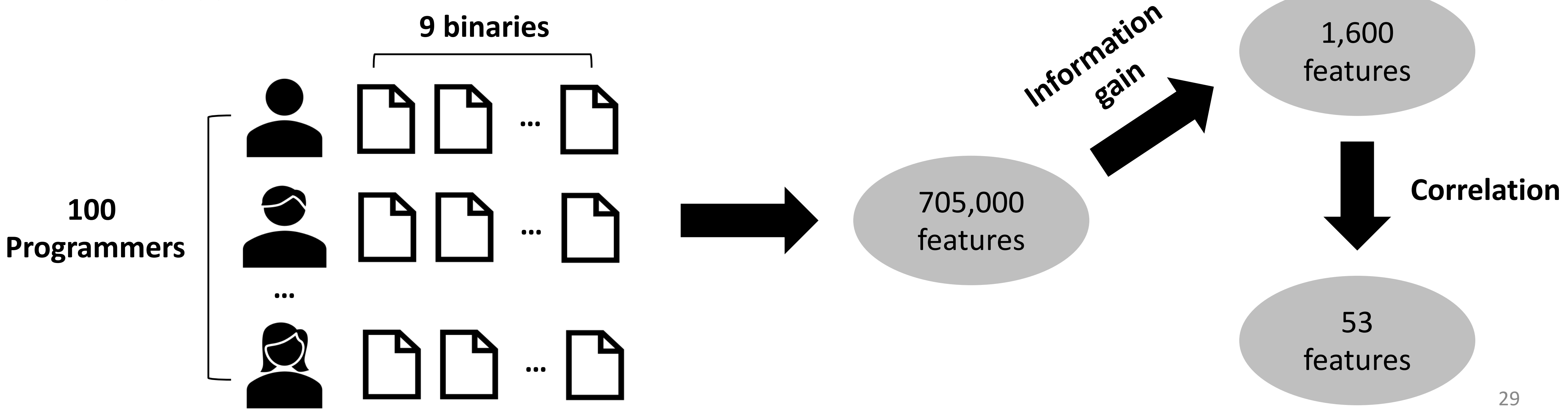
- Extract 705,000 representations of code properties of 100 programmers
- After applying information gain, they reduce the feature set to approximately 1,600 features from all feature types
- Correlation based feature selection preserves 53 of the highly distinguishing features



Feature extraction / selection

Why 100 programmers?

- Extract 705,000 representations of code properties of 100 programmers
- After applying information gain, they reduce the feature set to approximately 1,600 features from all feature types
- Correlation based feature selection preserves 53 of the highly distinguishing features



Predictive features

Feature	Source	# of selected features	Feature	Source	# of selected features
Instruction 1-grams	ndisasm/radare	2/3	Word 1-grams	hex-rays	6
Instruction 2-grams	ndisasm/radare	5/1	AST node TF	hex-rays	3
Instruction 3-grams	ndisasm/radare	4/0	Labeled AST edge TF	hex-rays	0
Instruction 6-grams	ndisasm/radare	24/0	AST node TFIDF	hex-rays	1
CFG node 1-grams	radare	3	AST node average depth	hex-rays	0
CFG edges	radare	1	C++ keywords	hex-rays	0

- Assembly instructions: arithmetic, logic, stack operations
- AST: file and stream operations, initializations of variables

Predictive features

Feature	Source	# of selected features	Feature	Source	# of selected features
Instruction 1-grams	ndisasm/radare	2/3	Word 1-grams	hex-rays	6
Instruction 2-grams	ndisasm/radare	5/1	AST node TF	hex-rays	3
Instruction 3-grams	ndisasm/radare	4/0	Labeled AST edge TF	hex-rays	0
Instruction 6-grams	ndisasm/radare	24/0	AST node TFIDF	hex-rays	1
CFG node 1-grams	radare	3	AST node average depth	hex-rays	0
CFG edges	radare	1	C++ keywords	hex-rays	0

- Assembly instructions: arithmetic, logic, stack operat
- AST: file and stream operations, initializations of vari

How to convert numeric vector?

The result of the main GCJ experiment

- Demonstrate how de-anonymizing programmers from their executable binaries is possible
- Take a set of 100 programmers who all have 9 binaries (optimization X)
- Process the executable binaries and extract/select features
- Perform 9-fold cross-validation (8 training samples, 1 test sample)
- 95% accuracy, significantly higher than previous work (61%)

➔ *The method can de-anonymize programmers from their executable binaries*

Validation of the selected feature set

- Take a different set of 100 programmers with 9 binaries (non-overlapping)
- Omit the feature selection step and extract 53 features
- 96% accuracy, very similar with the main accuracy (95%)
  - These selected features generalize to other programmers and problems, and these are not overfitting to the 100 programmers they were generated from

➔ *The feature set selected via dimensionality reduction works and is validated across different sets of programmers*

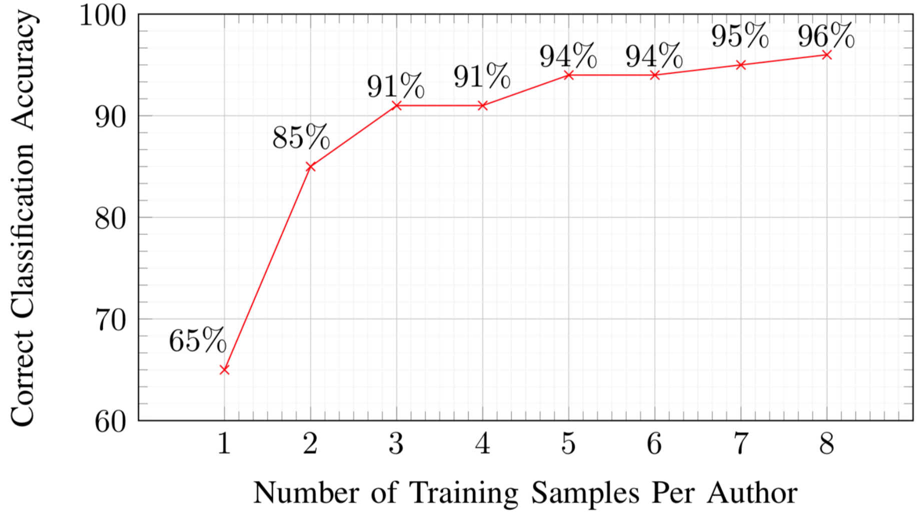
Change the # of training samples per author

- Supervised machine learning methods require the amount of labeled training data
  - Labeling training samples may be costly or laborious
- To determine how much training data is required to reach a stable accuracy and to explore the accuracy with severely limited training data

Change the # of training samples per author

- The classifier can identify the programmers with 65% accuracy on the basis of a single training sample
- The classifier obtains a stable accuracy by training on 6 samples

➔ Even a single training sample per programmer is sufficient for de-anonymization

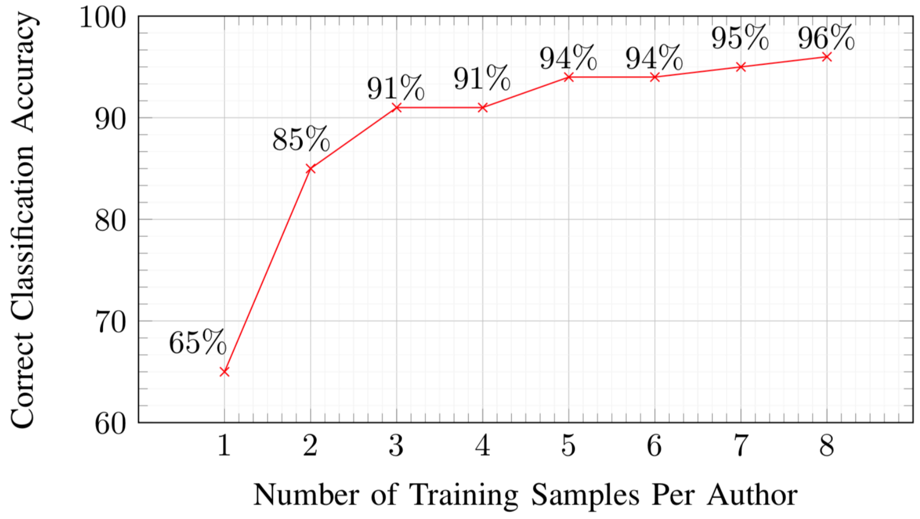


Change the # of training samples per author

- The classifier can identify the programmers with 65% accuracy on the basis of a single training sample
- The classifier obtains a stable accuracy by training on 6 samples

Perform cross-validation?
training:test=n:?

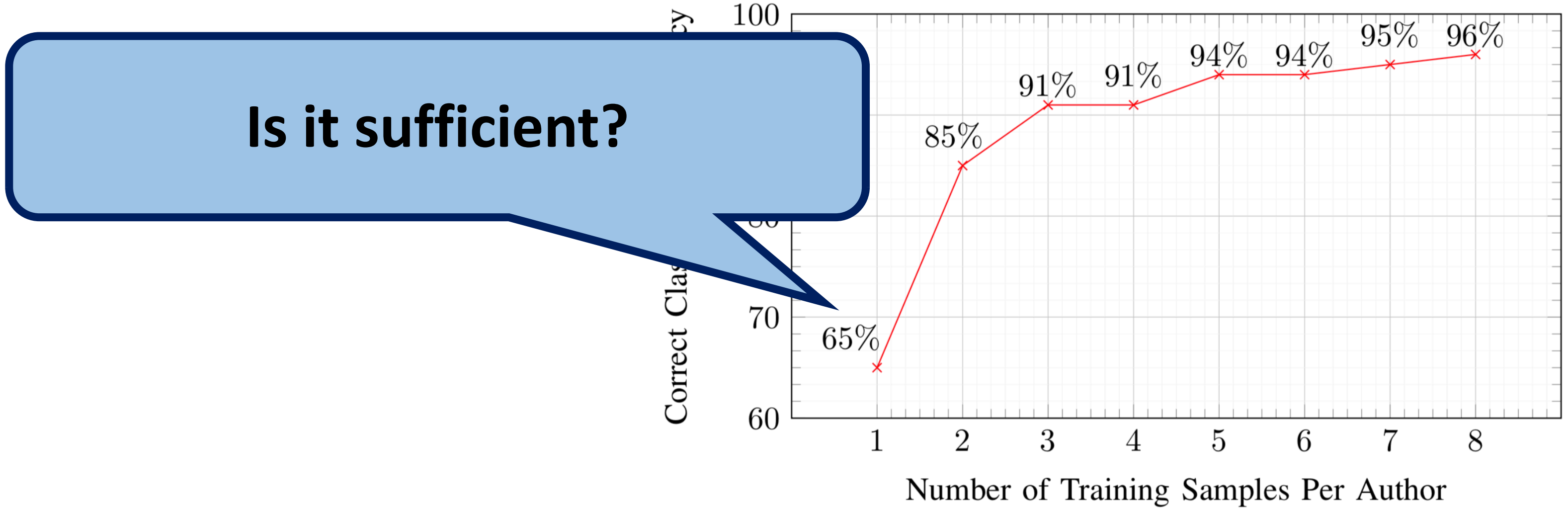
➔ Even a single training sample per programmer is sufficient for de-anonymization



Change the # of training samples per author

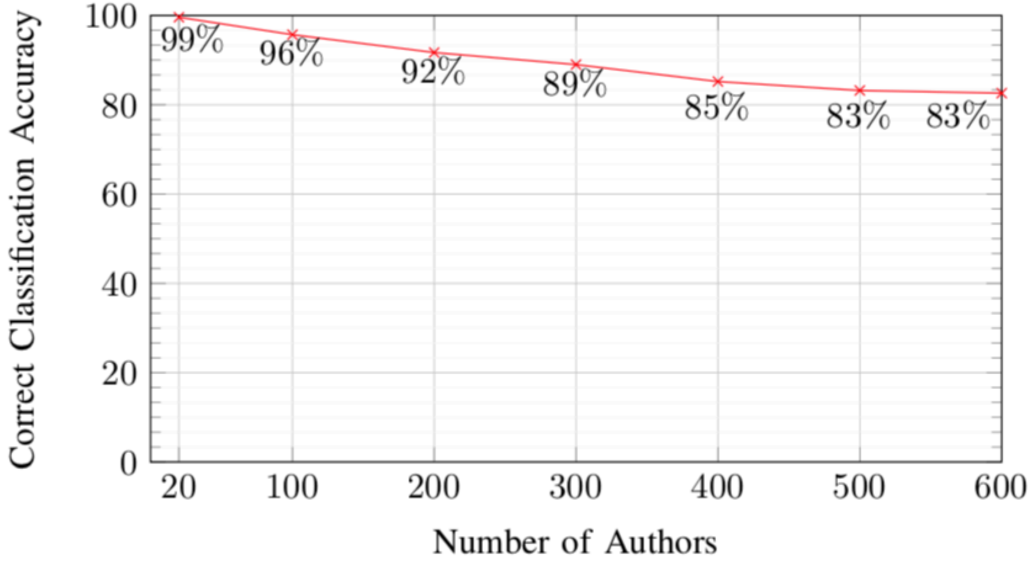
- The classifier can identify the programmers with 65% accuracy on the basis of a single training sample
- The classifier obtains a stable accuracy by training on 6 samples

➔ Even a single training sample per programmer is sufficient for de-anonymization



Large scale de-anonymization

- Demonstrate how well the method scales up to 600 programmers
- Omit the feature selection step and extract 53 feature
- Show that the method can scale to larger datasets with the reduced set of features with a surprisingly small drop on accuracy



Comparison with previous work

- *Who wrote this code? Identifying the authors of program binaries* (Rosenblum)
  - Evaluate on 191 programmers each with at least 8 training samples
  - Use 1,900 coding style features (vs. 53 features)
  - Support Vector Machine

- Accuracy is significantly higher in every case

Related Work	Number of Programmers	Number of Training Samples	Accuracy	Classifier
Rosenblum [39]	20	**8-16**	77%	SVM
This work	20	**8**	90%	SVM
This work	20	**8**	99%	RF
Rosenblum [39]	100	8-16	**61%**	SVM
This work	100	8	**84%**	SVM
This work	100	8	**96%**	RF
Rosenblum [39]	191	8-16	**51%**	SVM
This work	191	8	**81%**	SVM
This work	191	8	**92%**	RF
This work	**600**	8	**71%**	SVM
This work	**600**	8	**83%**	RF

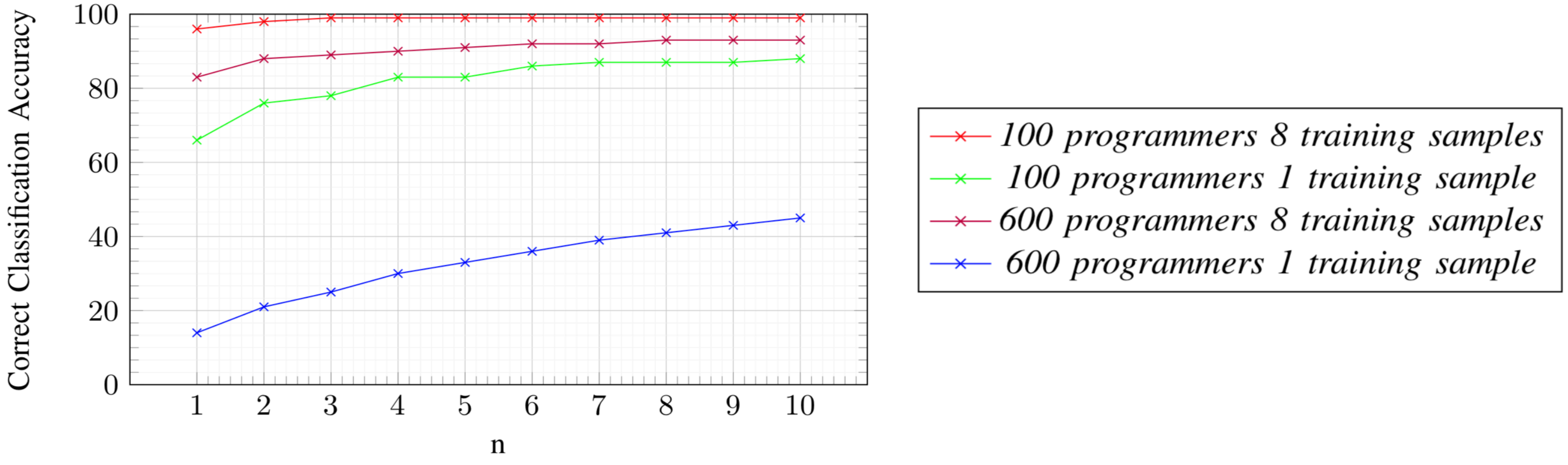
Relaxed classification

- In cases where we have too many classes or the classification accuracy is lower than expected, we can relax the classification to top–n classification
- In top–n relaxed classification, if the test instance belongs to one of the most likely n classes, the classification is considered correct
- This can be useful in cases when an analyst or adversary is interested in finding a suspect set of n authors, instead of a direct top–1 classification
- Once the suspect set size is reduced, the manual effort required by an analyst or adversary could reduces

Relaxed classification

- How correct classification accuracies approach 100% as the classification is relaxed to top-10

➔ In difficult scenarios, the classification task can be narrowed down to a small suspect set



Experiments

1. Google Code Jam Experiments
2. Real-world scenarios
   1) Optimized executable binaries
   2) Fully stripped executable binaries
   3) Obfuscated executable binaries
   4) Github executable binaries
   5) Nulled.IO executable binaries

Optimized & stripped executable binaries

# of programmers	# of training samples	Compiler optimization level	Accuracy
100	8	None	96%
100	8	1	93%
100	8	2	89%
100	8	3	89%
100	8	Stripped	72%

- Investigate how much programming style is preserved in executable binaries that are compiled with optimization

➔ *Programmers of optimized executable binaries can be de-anonymized*

➔ *Removing symbol information does not anonymize executable binaries*

Obfuscated executable binaries

- Apply 3 binary obfuscation techniques using Obfuscator-LLVM
  - Bogus control flow insertion
  - Instruction substitution
  - Control flow flattening
- 88% accuracy

➔ *The method can de-anonymize programmers from obfuscated binaries*

De-anonymization in the Wild

- Collect a dataset from GitHub
  - Clone 439 repositories from 161 authors
  - Select 3,438 C/C++ source files
  - Compile 1,075 object files from 90 authors
    - # of object files per author: 2 – 24
  - Select 50 authors that have 6 to 15 files, a total of 542 files
- Extract 53 features
- 65% accuracy

Case study: Nulled.IO Hacker Forum

- Nulled.IO: leacked well known hacker forum
- Created a dataset from 4 forum members with a total of 13 Windows executables
  - One of the members had only one sample, used to test set
- Extract a total of 605 features consisting of decompiled source code features and ndisasm disassembly features
- De-anonymize these programmers with 100% accuracy
- One sample is classified in all cases with the lowest confidence
  - It is below the verification threshold and is recognized by the classifier as a sample that does not belong to the rest of the programmers

Discussion

- Pros
  - Discover a set of features that effectively represent coding style in executable binaries
  - Perform experiments under various conditions
  - Outperform previous work with high accuracy
- Cons
  - Use a commercial decompiler, hex-rays
  - Can’t de-anonymize multiple authors of collaboratively generated binaries
  - Many assumptions: C/C++, ELF, compiler, optimization level, …

Q & A