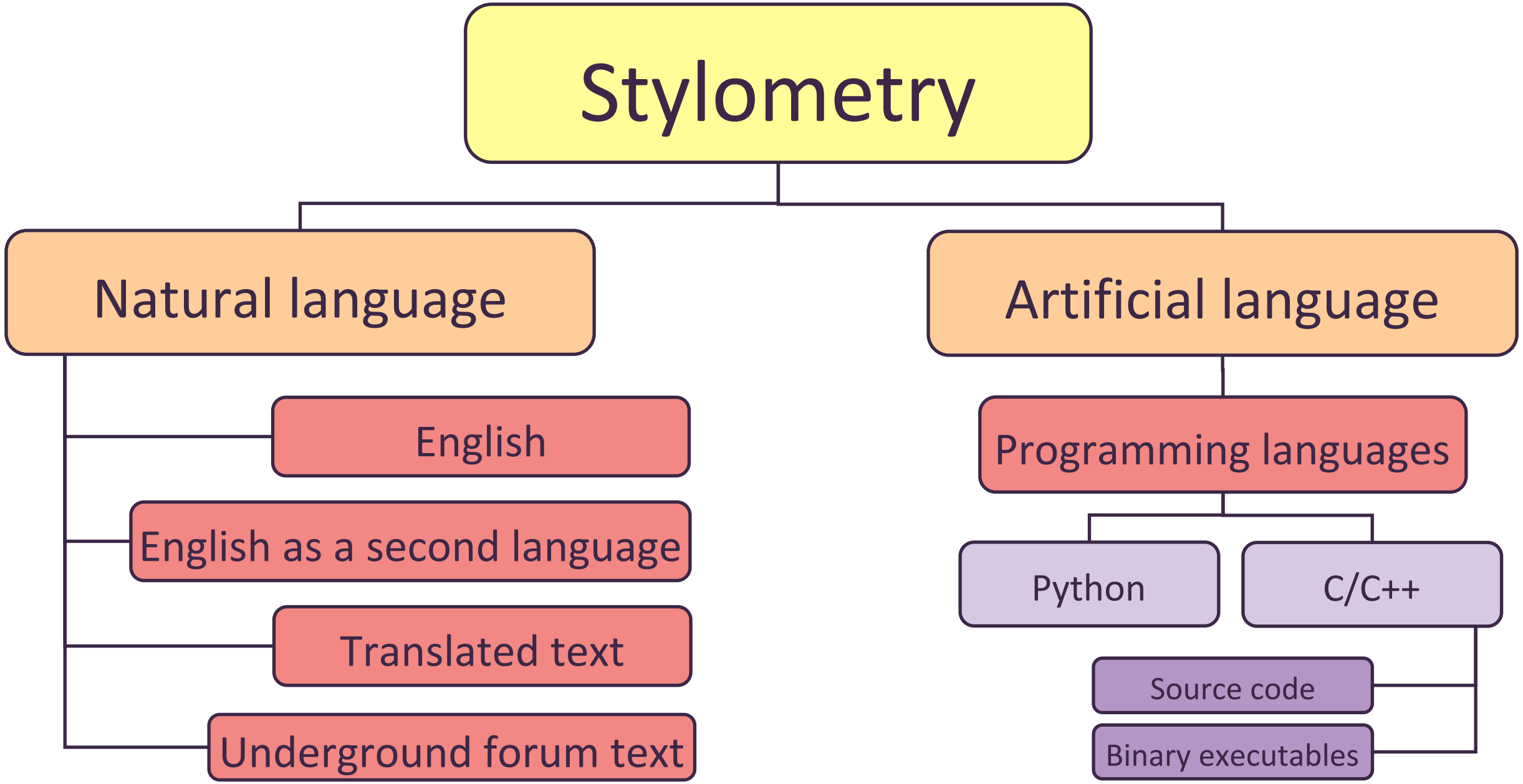
De-anonymizing Programmers from Source Code and Binaries

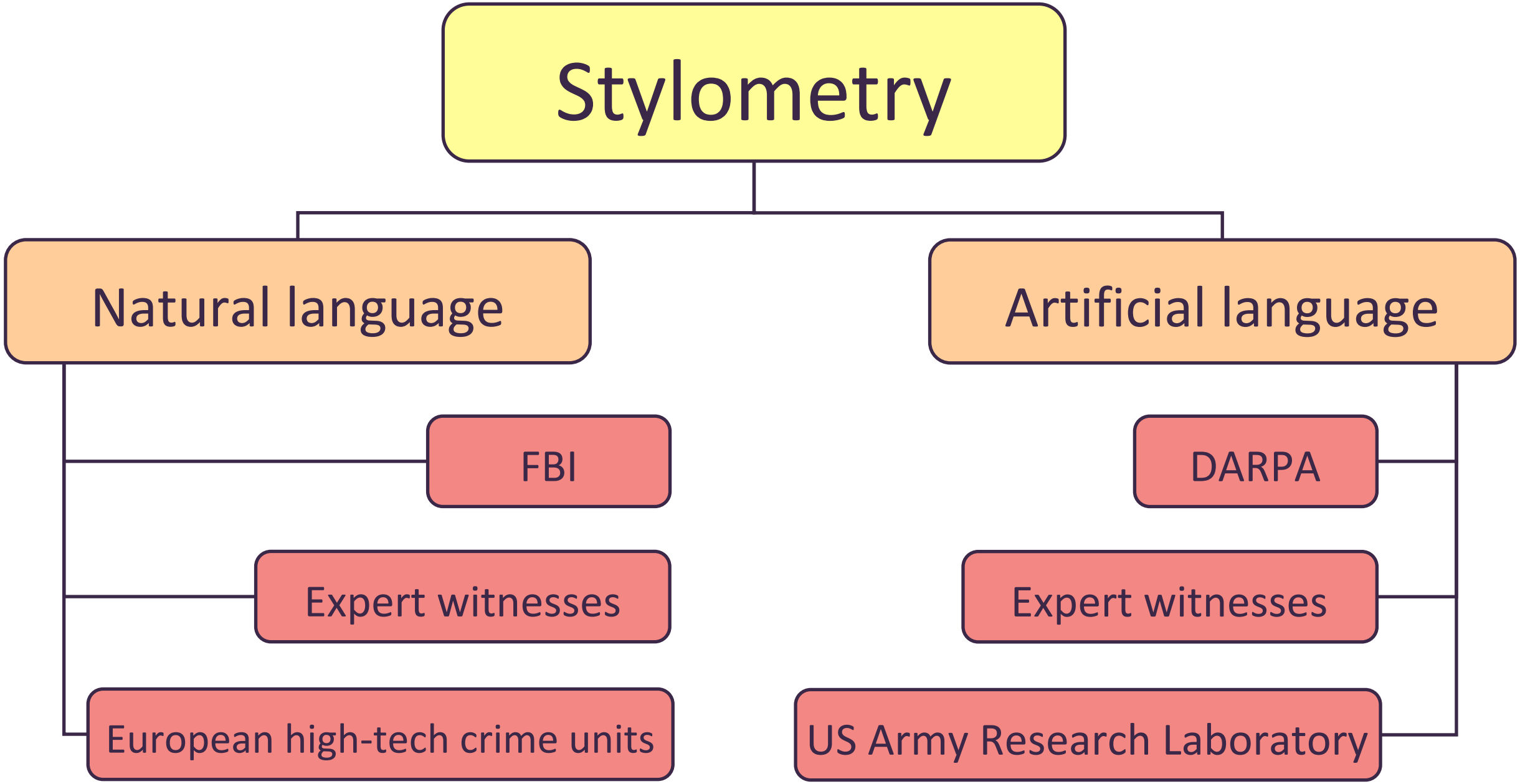
Rachel Greenstadt
@ragreens

Associate Professor
Drexel University

Aylin Caliskan
@aylin_cim

Assistant Professor
George Washington University


Stylometry
Natural language
Artificial language
English
English as a second language
Translated text
Underground forum text
Programming languages
Python
C/C++
Source code
Binary executables


Stylometry
Natural language
Artificial language
FBI
Expert witnesses
European high-tech crime units
DARPA
Expert witnesses
US Army Research Laboratory

Why de-anonymize programmers?











Source code stylometry

Iran confirms death sentence for 'porn site' web programmer.



No technical difference between
security-enhancing and privacy-infringing

Source code stylometry

A machine learning classification task

Application	Learner	Setting
Software forensics	Multiclass	Open world
Stylometric plagiarism detection	Multiclass	Closed world
Copyright investigation	Two-class	Closed world
Authorship verification	Two-class/One-class	One-class open world

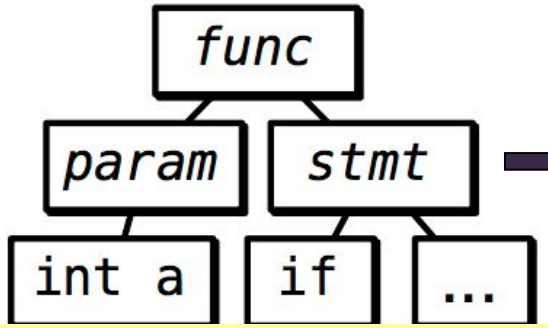
Language Processing

Supervised Machine Learning

Privacy and Security Implications

Source Code

```
int f(int a) {
  if (a < 0)
    a = 0;
  ...
```



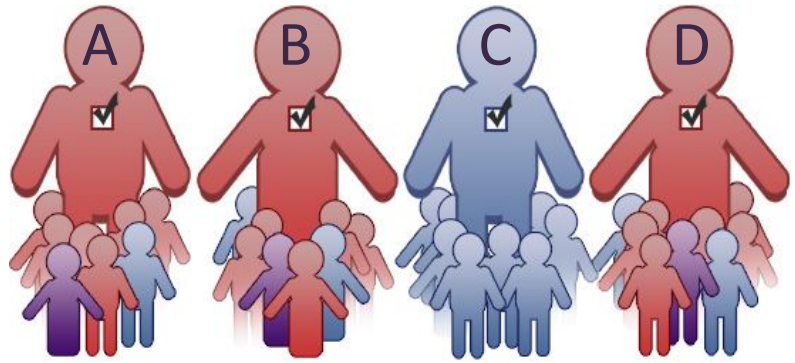
Fuzzy Parsing

Identifying Programmer Fingerprints



Random Forest Classifier

De-anonymizing Programmers

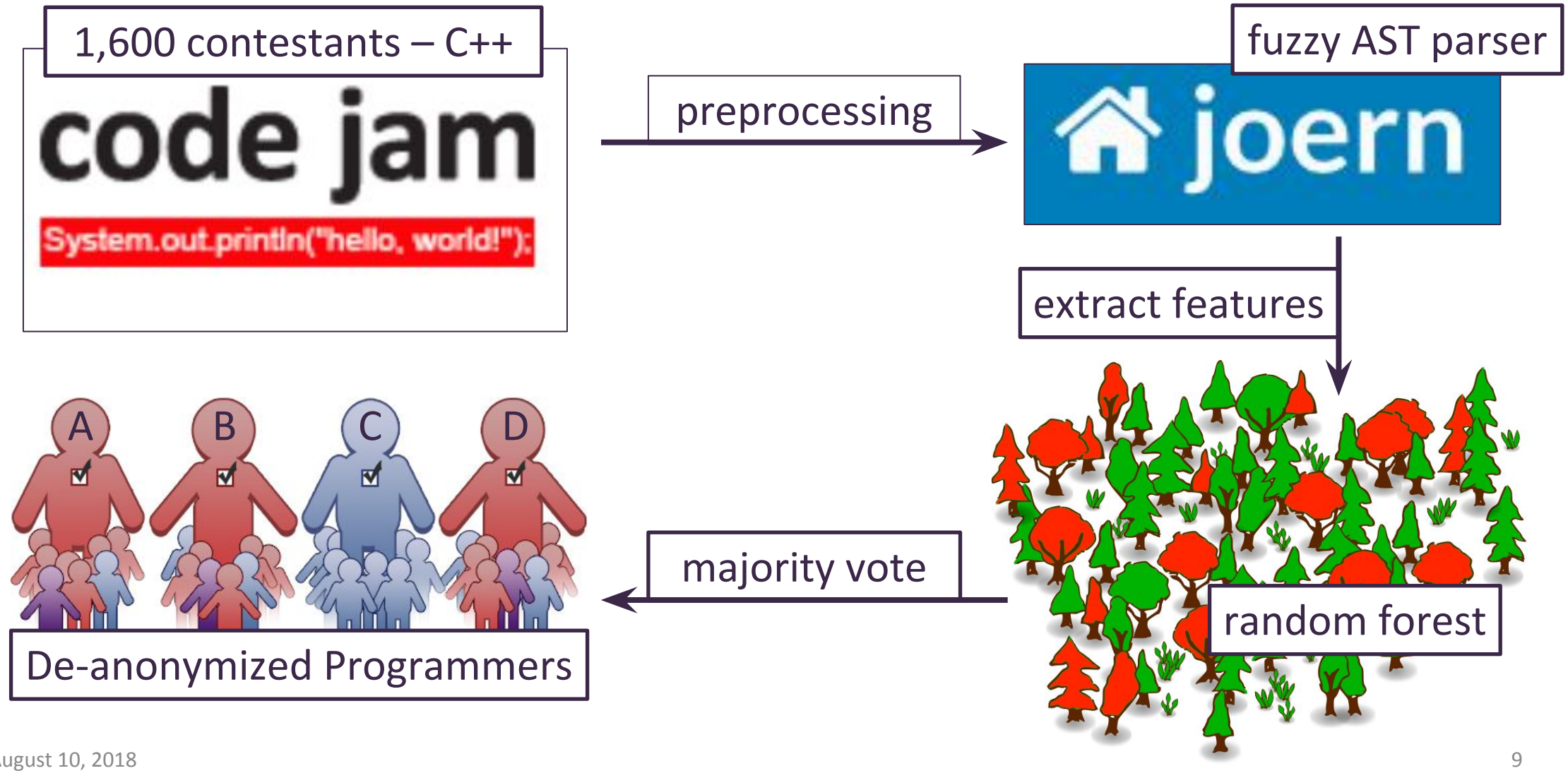


De-anonymizing programmers

Principled method & robust syntactic feature set

Application	Classes	Instances	Accuracy
Stylometric plagiarism detection	250 class	2,250	98%
Large scale de-anonymization	1,600 class	14,400	94%
Copyright investigation	Two-class	540	100%
Authorship verification	Two-class/One-class	2,240	91%
Open world problem	Multi-class	420	96%

Source code stylometry



Features

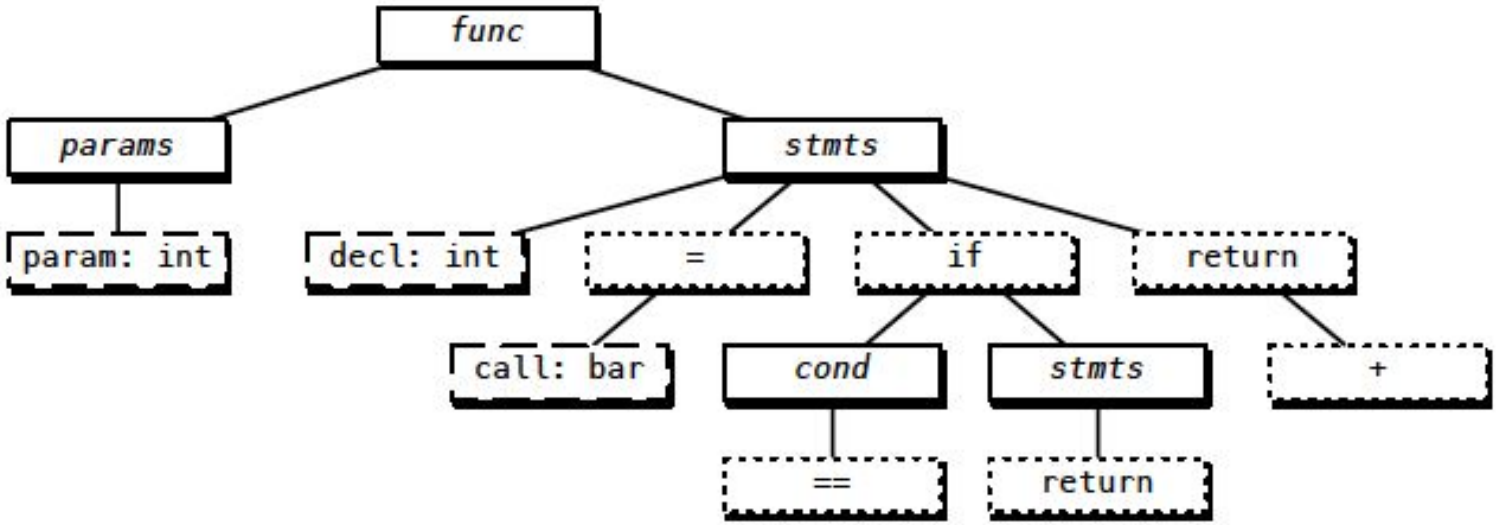
Source code

```
int foo(int y)
{
    int n = bar(y);

    if (n == 0)
        return 1;

    return (n + y);
}
```

Abstract syntax tree



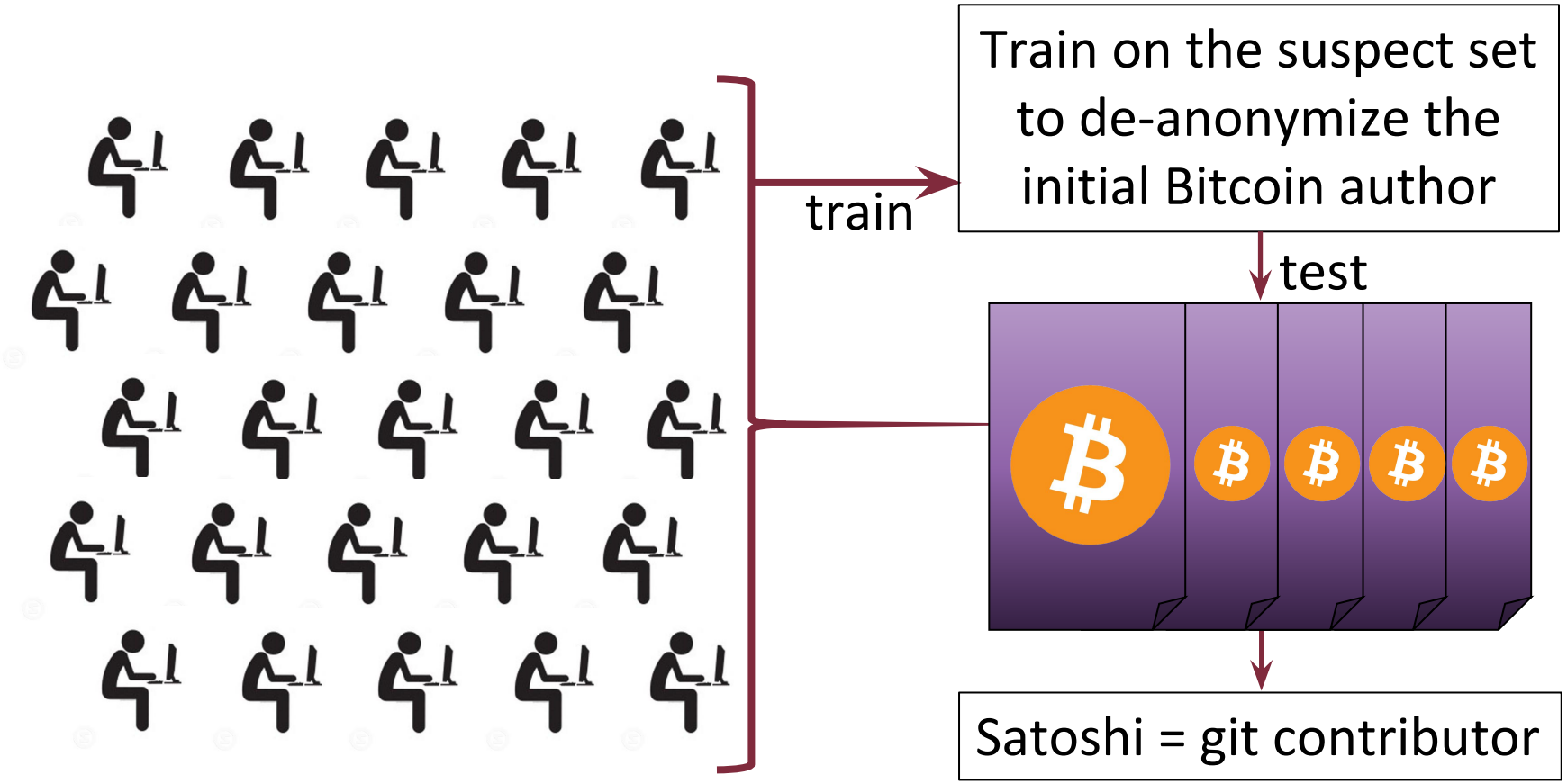
Case 1: Authorship attribution

- Who is this anonymous programmer?
- Who is Satoshi?



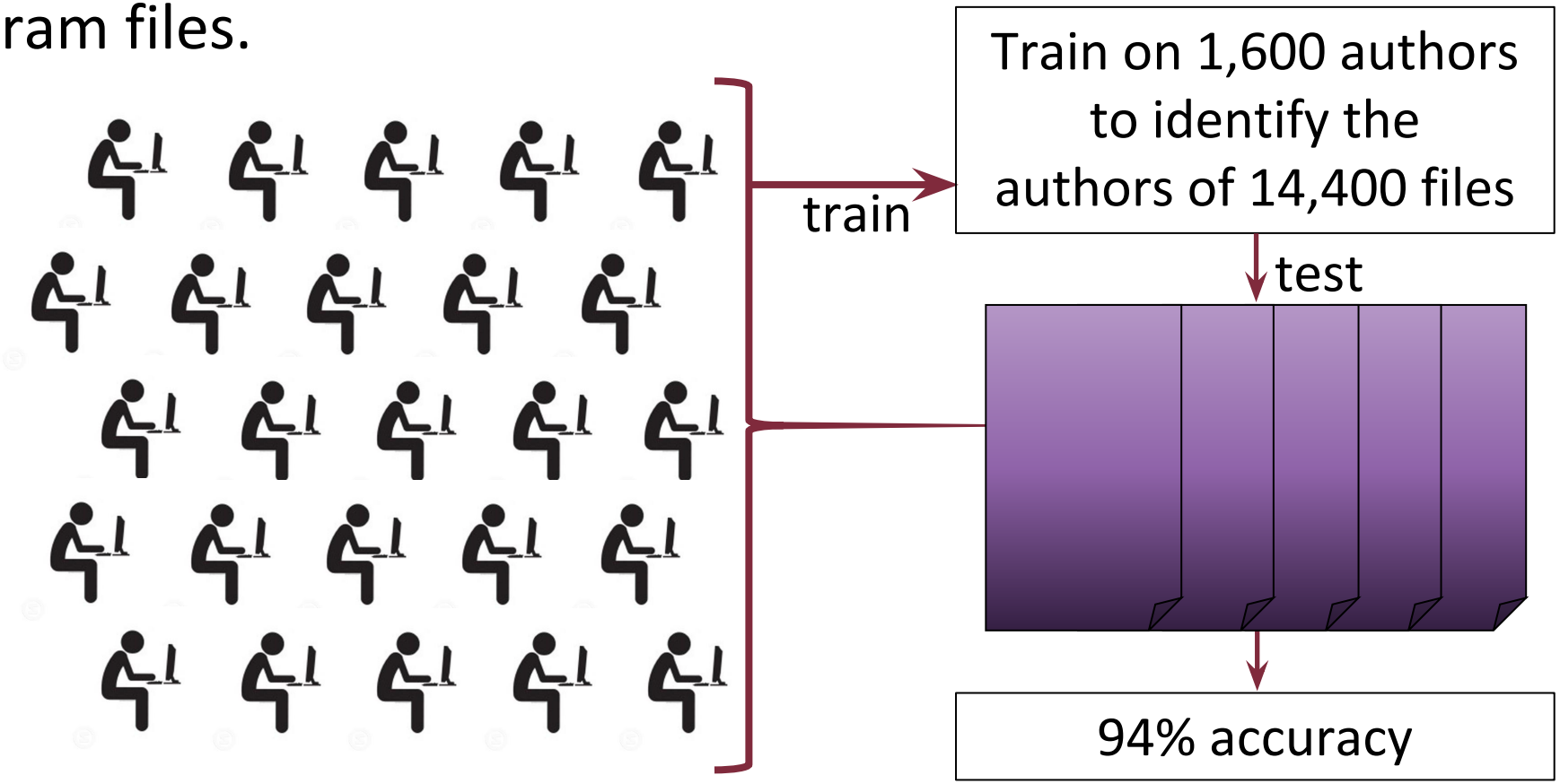
Case 1: Authorship attribution

- If only we had a suspect set for Satoshi…



Case 1: Authorship attribution

- 94% accuracy in identifying 1,600 authors of 14,400 anonymous program files.



Case 2: C++ Obfuscation - STUNNIX

Sample file with C++ code

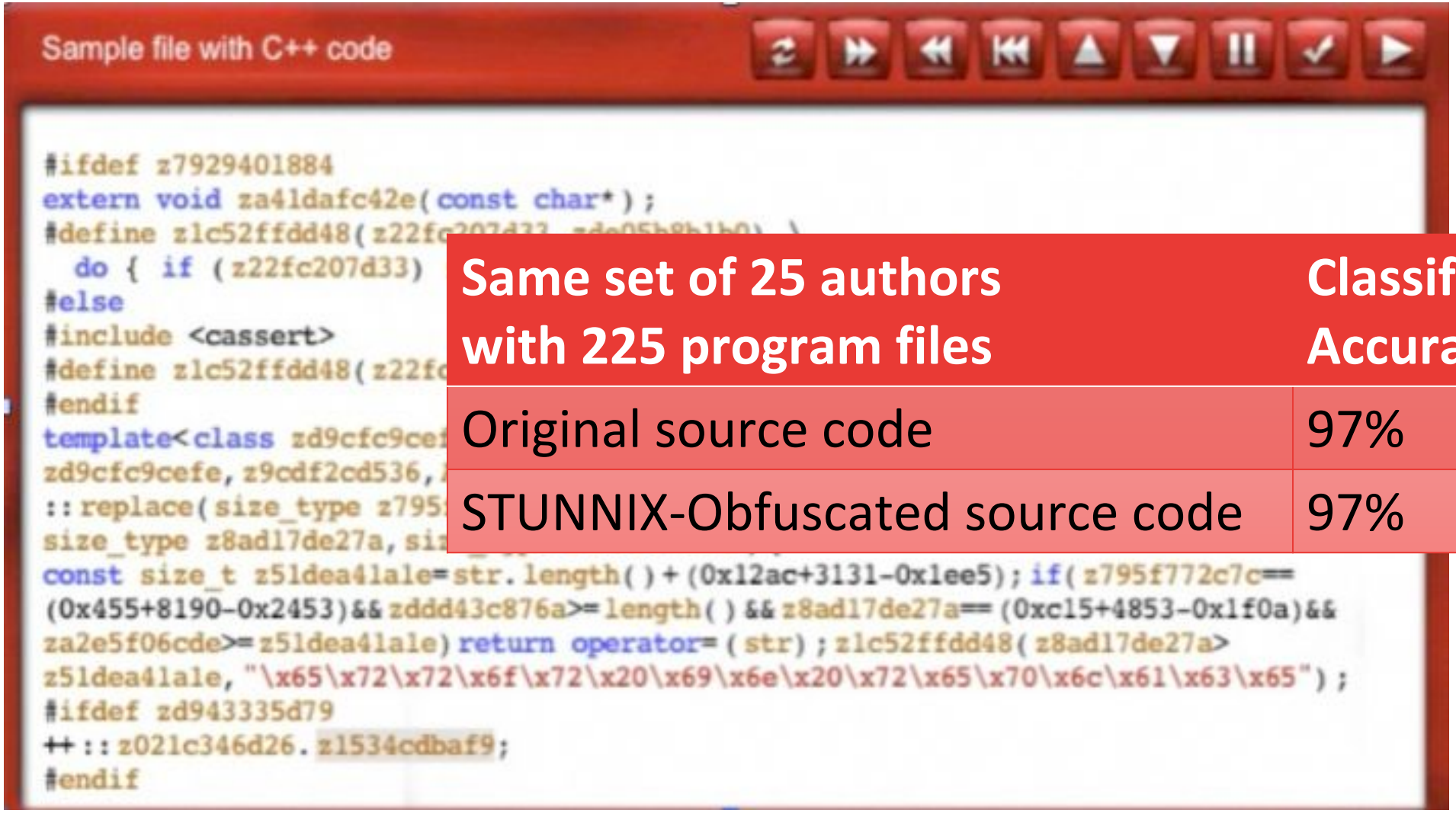


```
#ifdef __STL_USE_EXCEPTIONS /* this is conditional preprocessing */
extern void __out_of_range (const char *);
#define OUTOFRANGE(cond,msg) \
  do { if (cond) __out_of_range (#cond); } while (0)
#else
#include <cassert>
#define OUTOFRANGE(cond,msg) assert (!(cond))
#endif

template <class charT, class traits, class Allocator>
basic_string <charT, traits, Allocator>&
basic_string <charT, traits, Allocator>::
replace (size_type pos1, size_type n1,
  const basic_string& str, size_type pos2, size_type n2)
{
  //rather complex body follows
  const size_t len2 = str.length () + 2;

  if (pos1 == 0 && n1 >= length () && pos2 == 0 && n2 >= len2)
```

Case 2: C++ Obfuscation - STUNNIX



Same set of 25 authors with 225 program files	Classification Accuracy
Original source code	97%
STUNNIX-Obfuscated source code	97%

Case 2: C Obfuscation - TIGRESS

```c
#include<stdio.h>
int main()
{
  int T,test=1;
  double C,F,X,rate,time;
  scanf("%d",&T);
  while(T--)
    {
      scanf("%lf %lf %lf",&C,&F,&X);
      rate=2.0;
      time=0;
      while(X/rate>C/rate+X/(rate+F))
    {
      time+=C/rate;
      rate+=F;
    }
      time+=X/rate;
      printf("Case #%d: %lf\n",test++,time);
    }
  return 0;
}
```

Case 2: C Obfuscation - TIGRESS

```c
#include<stdio.h>
int main()
{
  int T,test=1;
  double C,F,X,rate,time;
  scanf("%d",&T);
  while(T--)
    {
      scanf("%lf %lf %lf",&C,&F,&X);
      rate=2.0;
      time=0;
      while(X/rate>C/rate+X/(rate+F))
    {
      time+=C/rate;
      rate+=F;
    }
      time+=X/rate;
      printf("Case #%d: %lf\n",test++,time);
    }
  return 0;
}
```



```
struct _IO_FILE;
struct timeval {
   long tv_sec ;
   long tv_usec ;
};
enum _1_main_$op {
    _1_main__string
$value_LIT_0$result_REG_1__convert_void_star2void_star
$result_STA_0$left_REG_0__local
$result_STA_0$value_LIT_0__store_void_star
$left_STA_0$right_STA_1__local
$result_STA_0$value_LIT_0__convert_void_star2void_star
$left_STA_0$result_REG_0__local
$result_REG_0$value_LIT_1__convert_void_star2void_star
$result_STA_0$left_REG_0__store_void_star$right_STA_0$left_REG_0 =
46,
    _1_main__local$result_REG_0$value_LIT_1__constant_int
$result_STA_0$value_LIT_0__store_int$right_STA_0$left_REG_0__local
$result_STA_0$value_LIT_0__convert_void_star2void_star
$left_STA_0$result_REG_0__string
$value_LIT_0$result_REG_1__convert_void_star2void_star
$result_STA_0$left_REG_0__store_void_star
$right_STA_0$left_REG_0__local
$result_REG_0$value_LIT_1__convert_void_star2void_star
$result_STA_0$left_REG_0 = 44,
    _1_main__convert_void_star2void_star
$left_STA_0$result_REG_0__load_int
$left_REG_0$result_REG_1__MinusA_int_int2int
$result_REG_0$left_REG_1$right_REG_2__store_int
$left_STA_0$right_REG_0__goto$label_LAB_0 = 161,
    _1_main__local$result_STA_0$value_LIT_0__local
$result_REG_0$value_LIT_1__convert_void_star2void_star
$result_STA_0$left_REG_0__load_double
$left_STA_0$result_REG_0__local
$result_REG_0$value_LIT_1__convert_void_star2void_star
$result_STA_0$left_REG_0__load_double
$left_STA_0$result_STA_0__convert_double2double
$left_STA_0$result_REG_0__local
```

Case 2: C Obfuscation - TIGRESS

```c
#include<stdio.h>
int main()
{
  int T,test=1;
  double C,F,X,rate,time;
  scanf("%d",&T);
  while(T--)
    {
      scanf("%lf %lf %lf",&C,&F,&X);
      rate=2.0;
      time=0;
      while(X/rate>C/rate+X/(rate+F))
    {
      time+=C/rate;
      rate+=F;
    }
      time+=X/rate;
      printf("Case #%d: %lf\n",test++,time);
    }
  return 0;
}
```



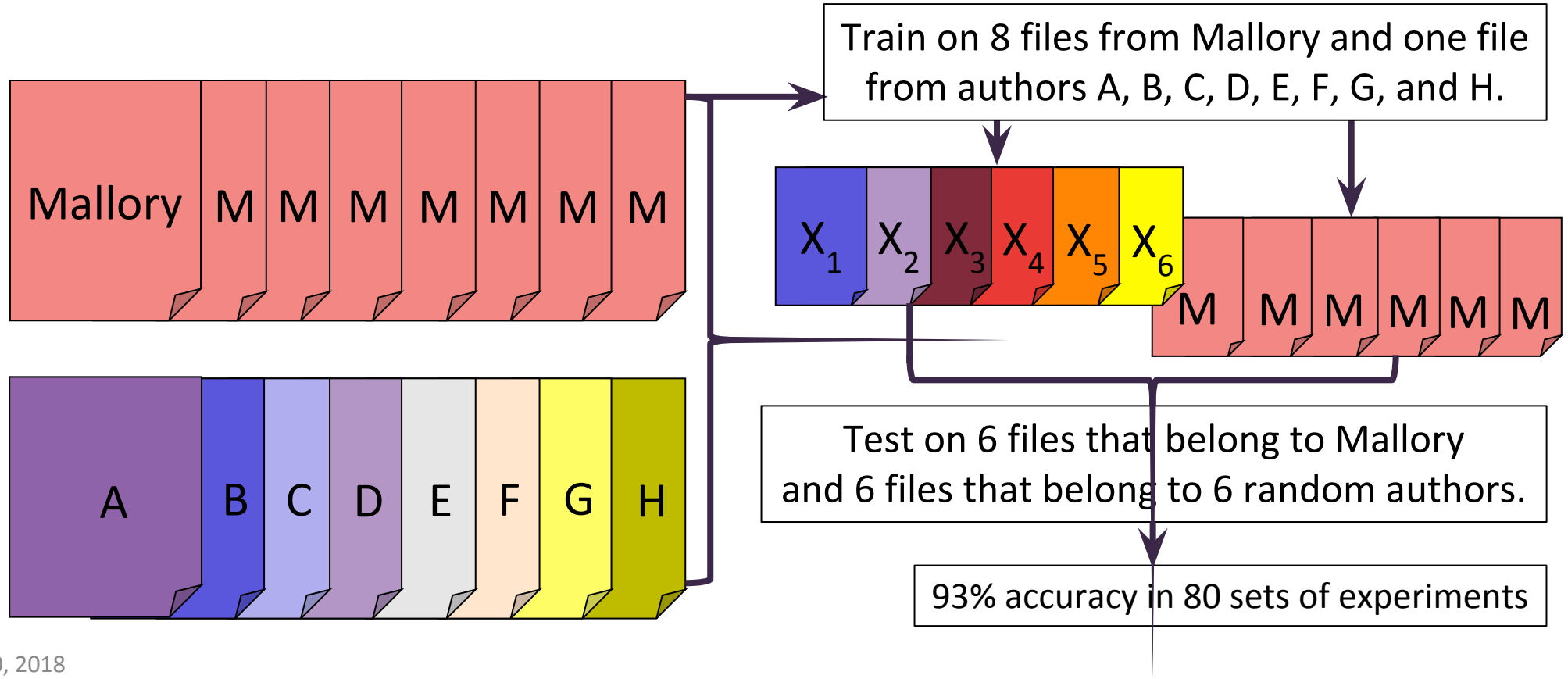
```
struct _IO_FILE;
struct timeval {
   long tv_sec ;
   long tv_usec ;
};
enum _1_main_$op {
    _1_main__string
$value_LIT_0$result_REG_1__convert_void_star2void_star
$result_STA_0$left_REG_0__local
$result_STA_0$value_LIT_0__store_void_star
$left_STA_0$right_STA_1__local
$result_STA_0$value_LIT_0__convert_void_star2void_star
$left_STA_0$result_REG_0__local
$result_REG_0$value_LIT_1__convert_void_star2void_star
$result_STA_0$left_REG_0__store_void_star$right_STA_0$left_REG_0 =
46,
    _1_main__local$result_REG_0$value_LIT_1__constant_int
$result_STA_0$value_LIT_0__store_int$right_STA_0$left_REG_0__local
$result_STA_0$value_LIT_0__convert_void_star2void_star
$left_STA_0$result_REG_0__string
$value_LIT_0$result_REG_1__convert_void_star2void_star
$result_STA_0$left_REG_0__store_void_star
```

Same set of 20 authors with 180 program files	Classification Accuracy
Original C source code	96%
TIGRESS-Obfuscated source code	67%

```
$left_STA_0$result_REG_0__local
$result_REG_0$value_LIT_1__convert_void_star2void_star
$result_STA_0$left_REG_0__load_double
$left_STA_0$result_STA_0__convert_double2double
$left_STA_0$result_REG_0__local
```

Case 3: Authorship verification

- Is this source code really written by this programmer?



What about executable binaries?

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; }
int n;
double a1[1001], a2[1001];
int main() {
        freopen("D-small-attempt0.in", "r", stdin);
        freopen("D-small.out", "w", stdout);
        int tt = nextInt();
        For(t,1,tt+1) {
                int n = nextInt();
```

. . .

Compiled code looks cryptic

```
00100000 00000000 00001000 00000000 00101000 00000000
00000000 00000000 00110100 00000000 00000000 00000000
00000100 00001000 00000000 00000001 00000000 00000000
00000000 00000001 00000000 00000000 00000101 00000000
00000000 00000000 00000100 00000000 00000000 00000000
00000011 00000000 00000000 00000000 00110100 00000001
00000000 00000000 00110100 10000001 00000100 00001000
00000000 00000000 00010011 00000000 00000000 00000000
00000100 00000000 00000000 00000000 00000001 00000000
00000000 00000000 00000001 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 10000000
00000100 00001000 00000000 10000000 00000100 00001000
11001000 00010111 00000000 00000000 11001000 00010111
00000000 00000000 00000101 00000000 00000000 00000000
```

. . .



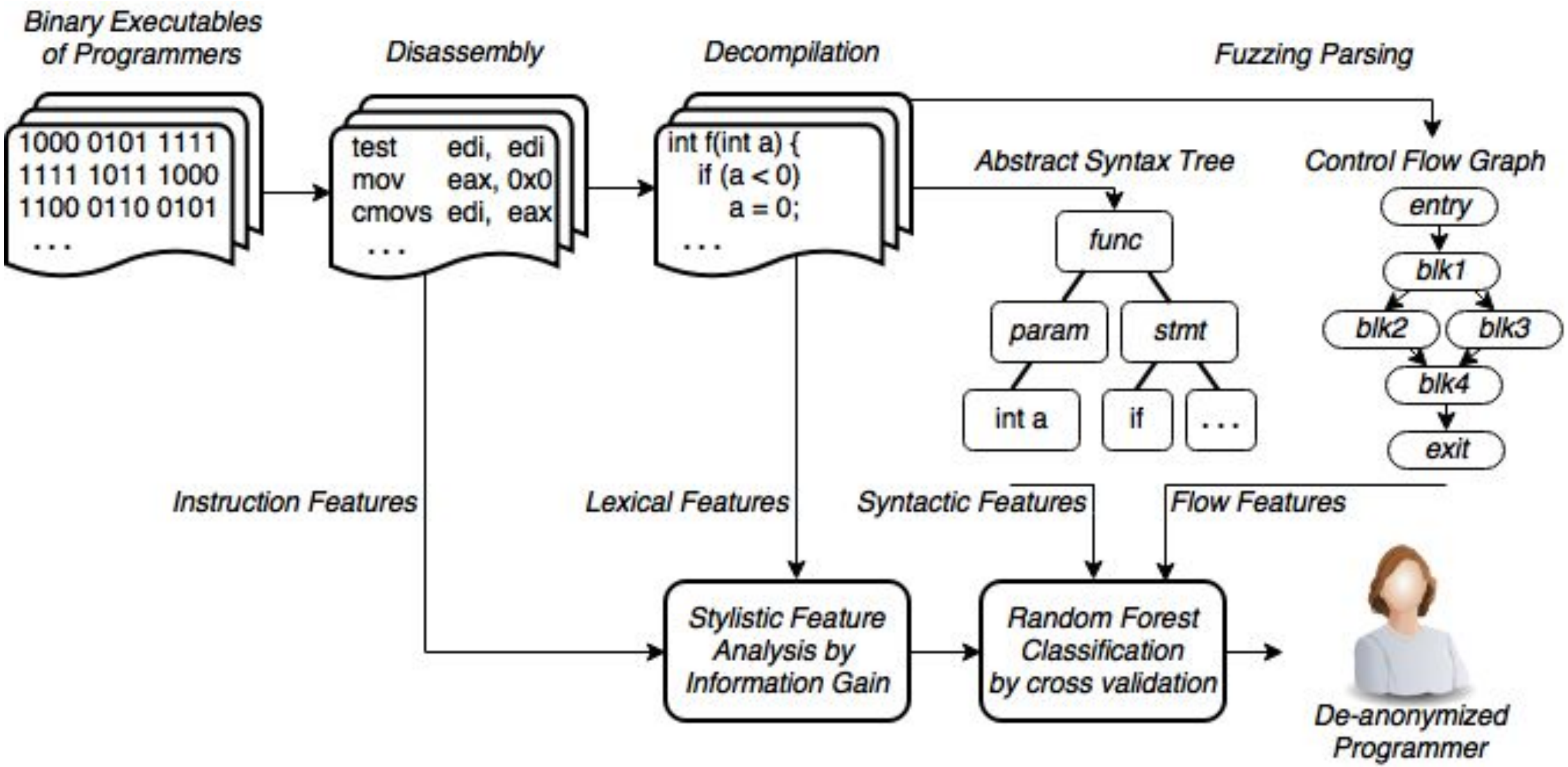
x0rz Follow
Security Researcher
Sep 13, 2016 · 4 min read

Interview with the LuaBot malware author

Creating a botnet of thousands of routers for DDoS activities

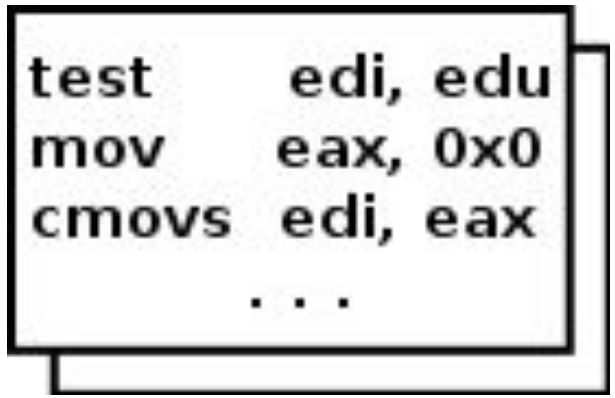
Who are you?

Just some guy who likes programming. I'm not known security researcher/programmer or member of any hack group, so probably best answer for this would be—nobody


Binary Executables of Programmers
Disassembly
Decompilation
Fuzzing Parsing
1000 0101 1111
1111 1011 1000
1100 0110 0101
...
test edi, edi
mov eax, 0x0
cmovs edi, eax
...
int f(int a) {
if (a < 0)
a = 0;
...
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

Features: Assembly

Disassembly



Assembly Features

Assembly unigrams

```
test
```

Assembly bigrams

```
eax, 0x0
```

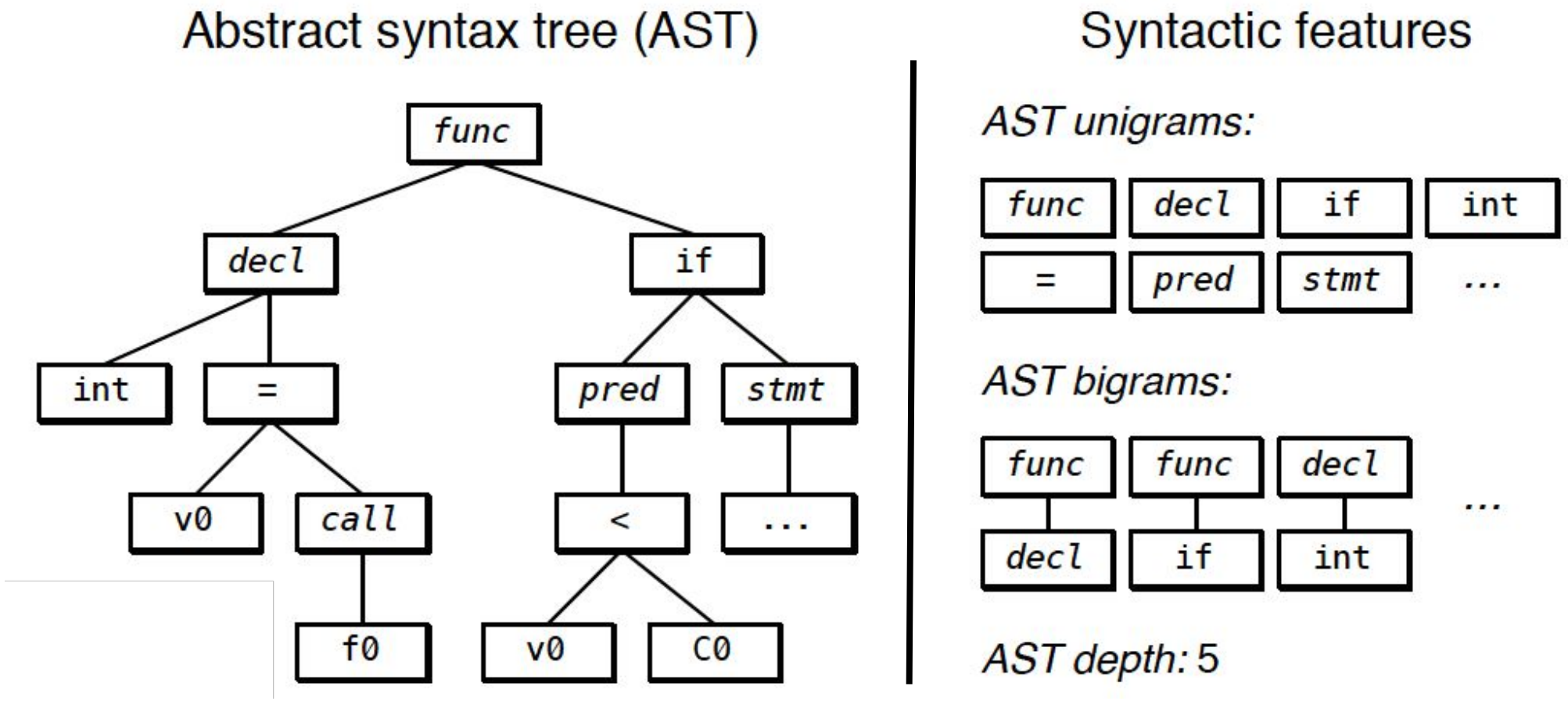
Assembly trigrams

```
cmovs edi, eax
```

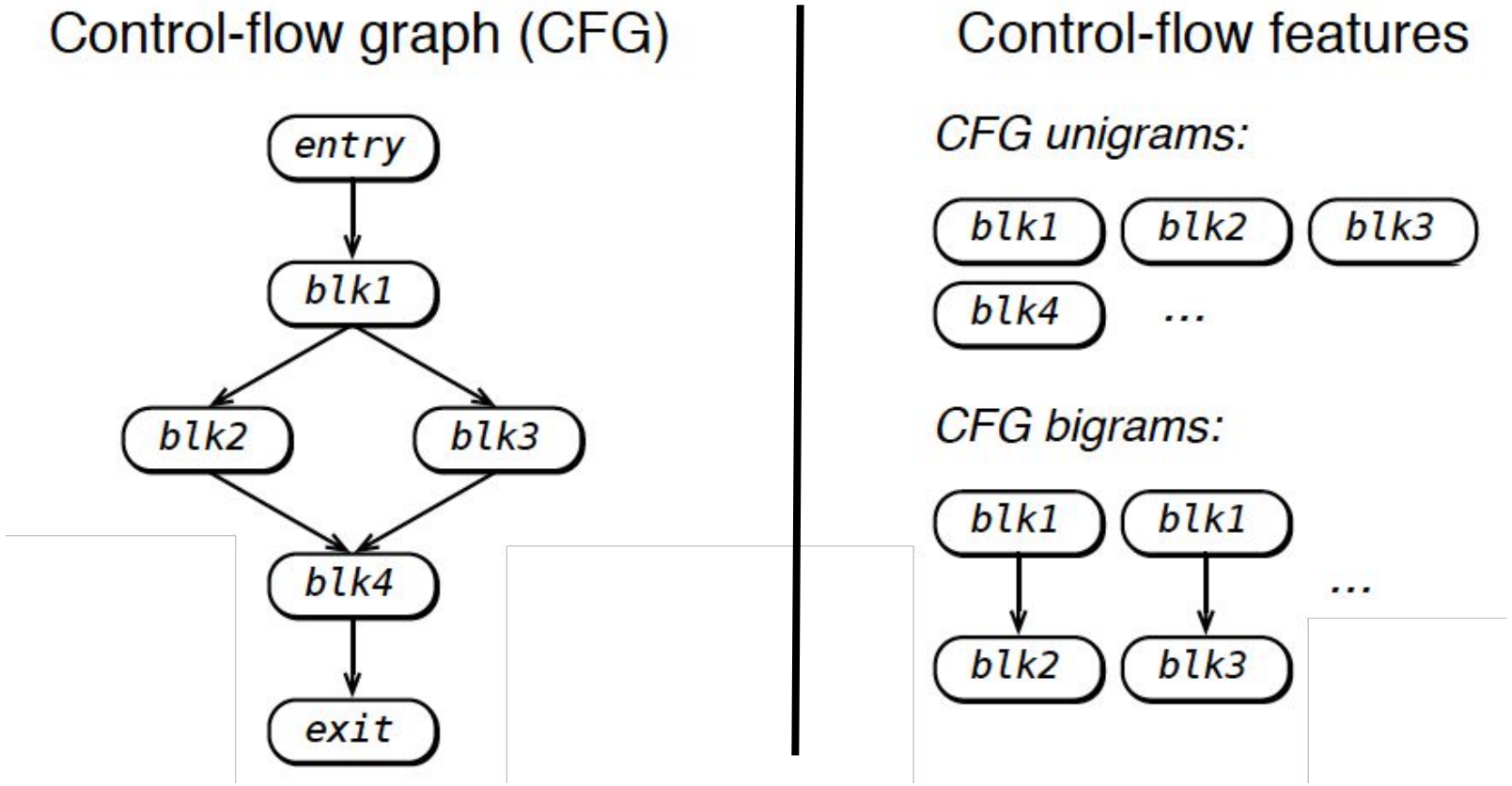
Two consecutive assembly lines

```
mov   eax, 0x0
cmovs edi, eax
```

Features: Syntactic

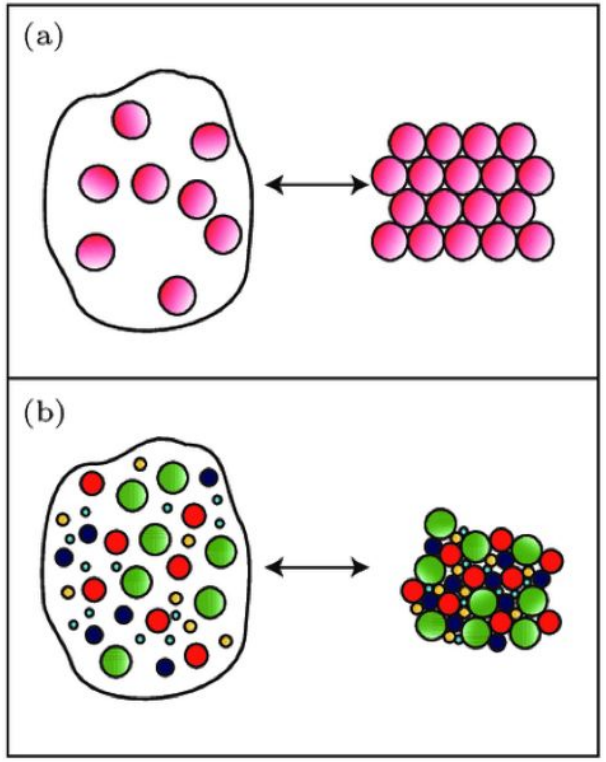


Features: Control flow



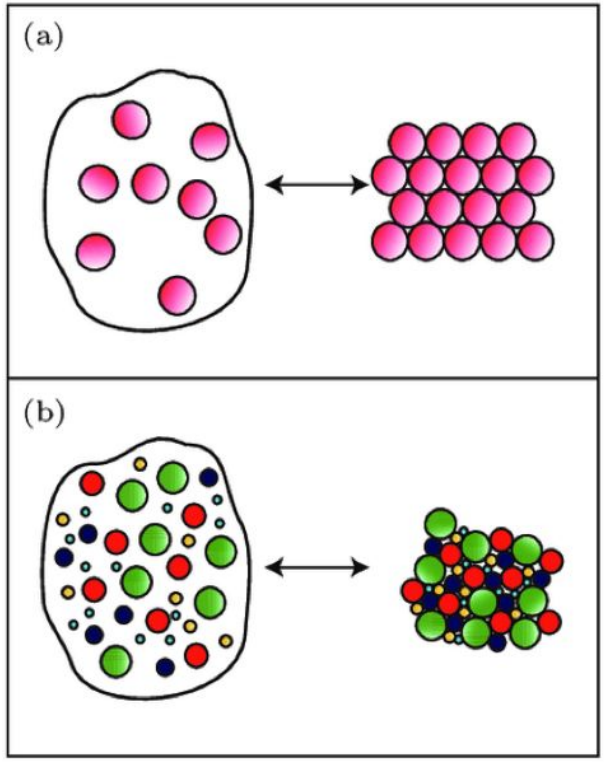
Dimensionality Reduction

- Information gain criterion
  - Keep features that reduce entropy – see (a)
  - Reduce dimension from ~700,000 to ~2,000

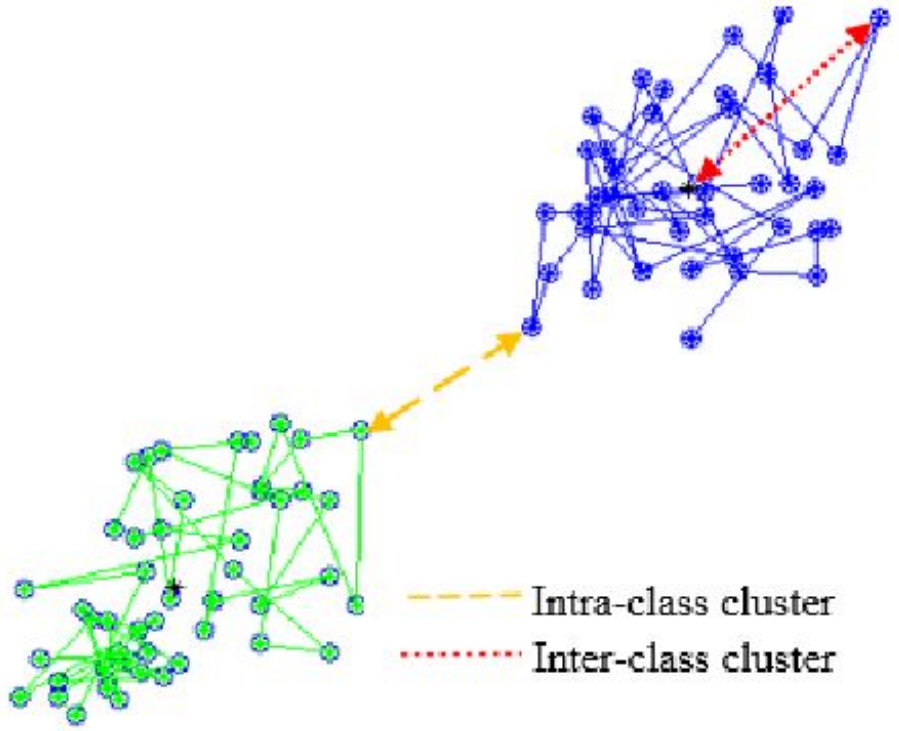


Dimensionality Reduction

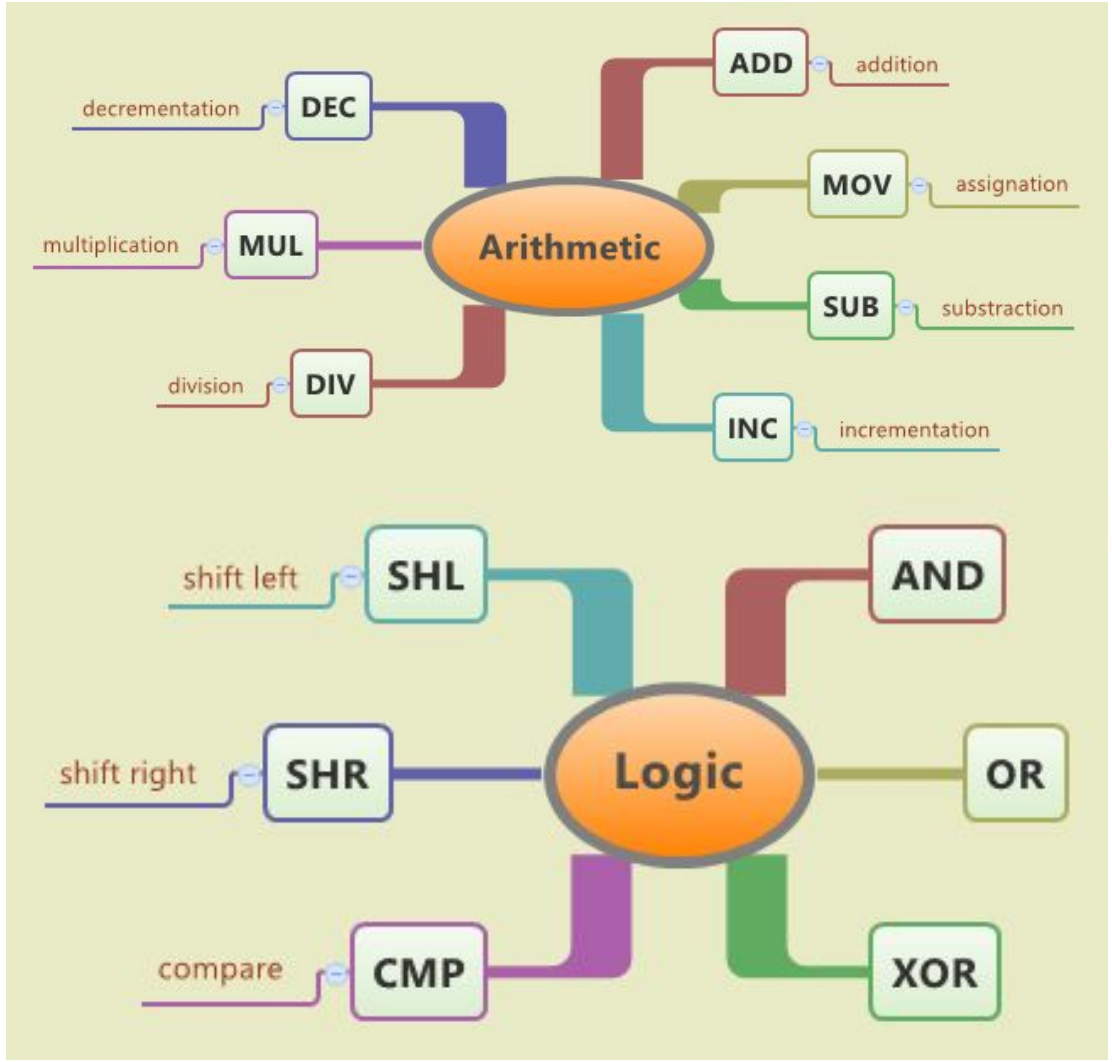
- Information gain criterion
  - Keep features that reduce entropy – see (a)
  - Reduce dimension from ~700,000 to ~2,000

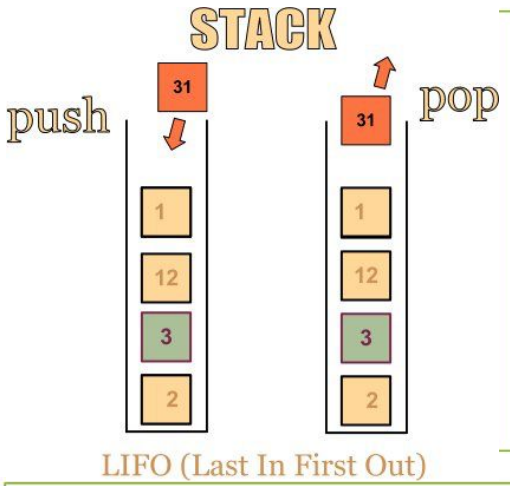


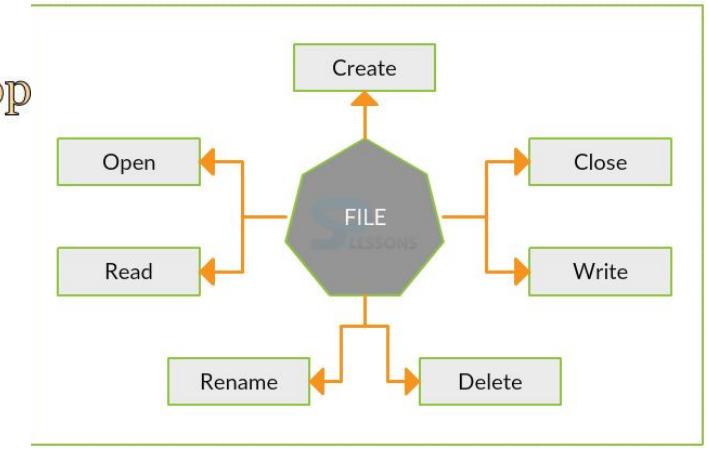
- Correlation based feature selection
  - Keep features with low inter-class correlation
  - Reduce dimension from ~2,000 to 53

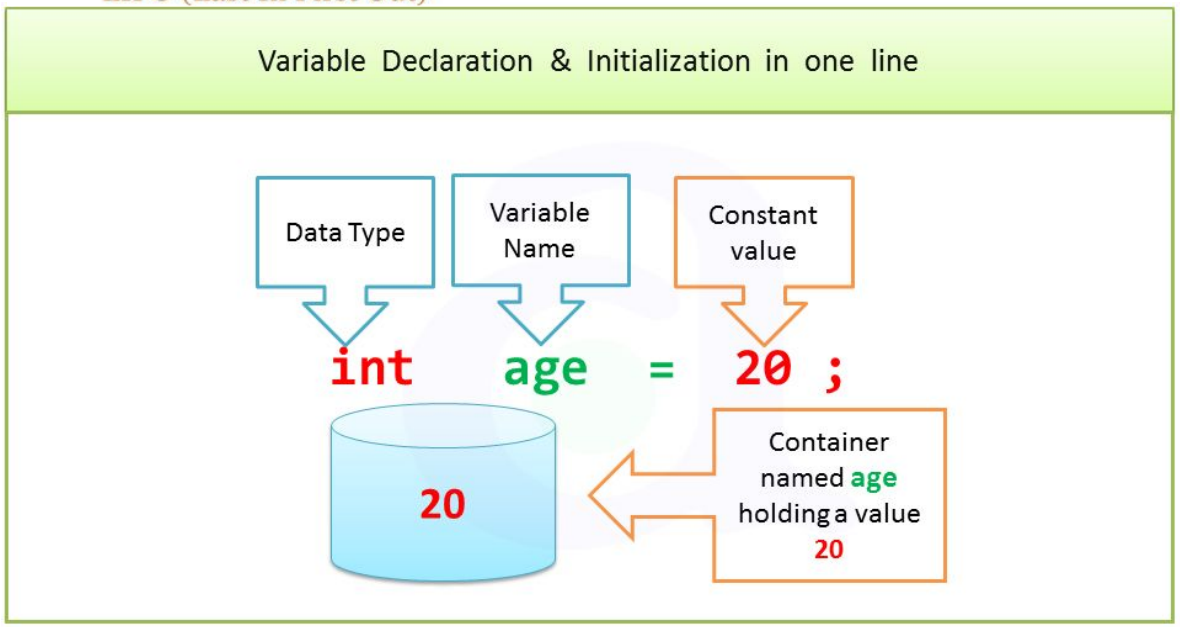


Predictive features









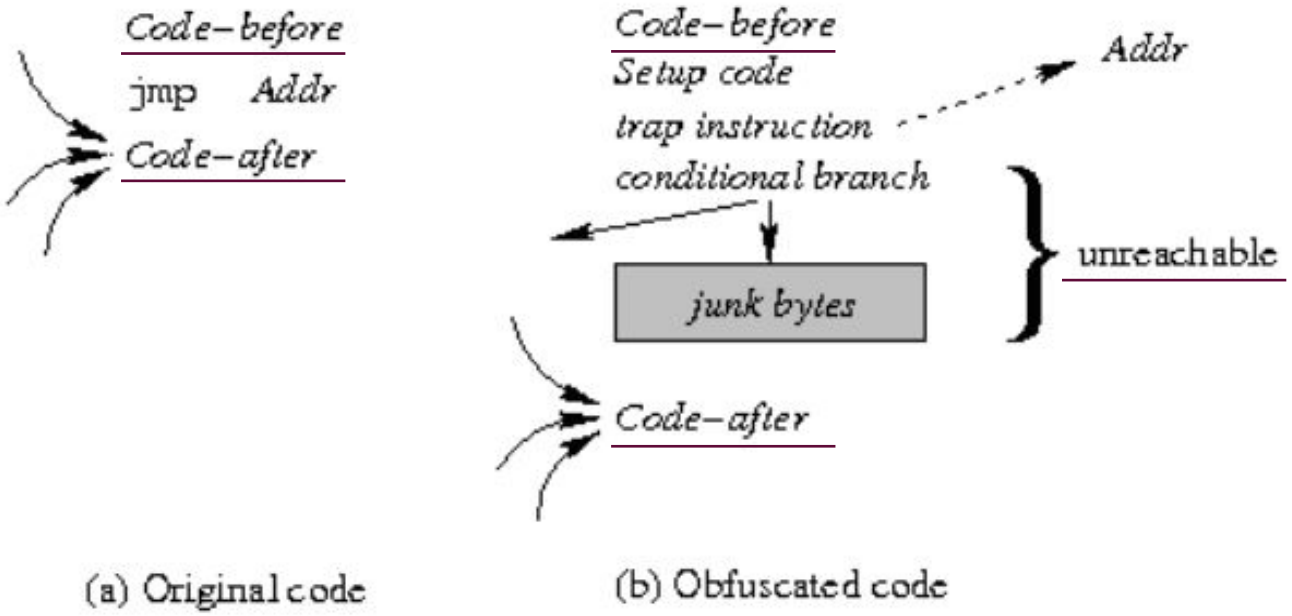
int variable Declaration and Initialization

Optimizations and stripping symbols

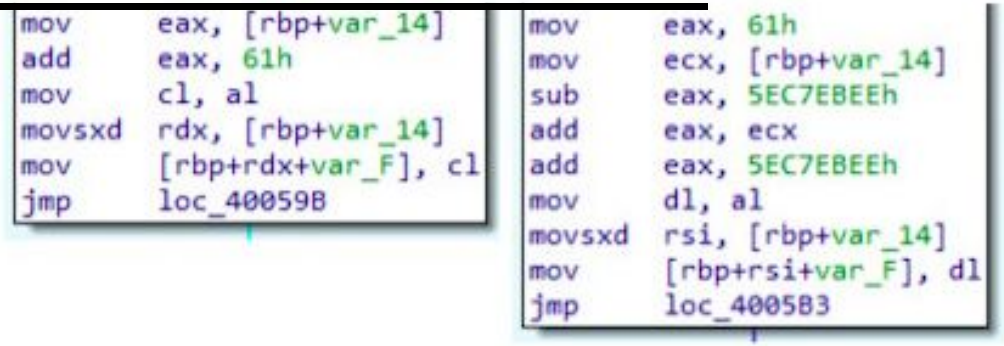
Number of programmers	Number of training samples	Compiler optimization level	Accuracy
100	8	None	96%
100	8	1	93%
100	8	2	89%
100	8	3	89%
100	8	Stripped symbols	72%

Obfuscation

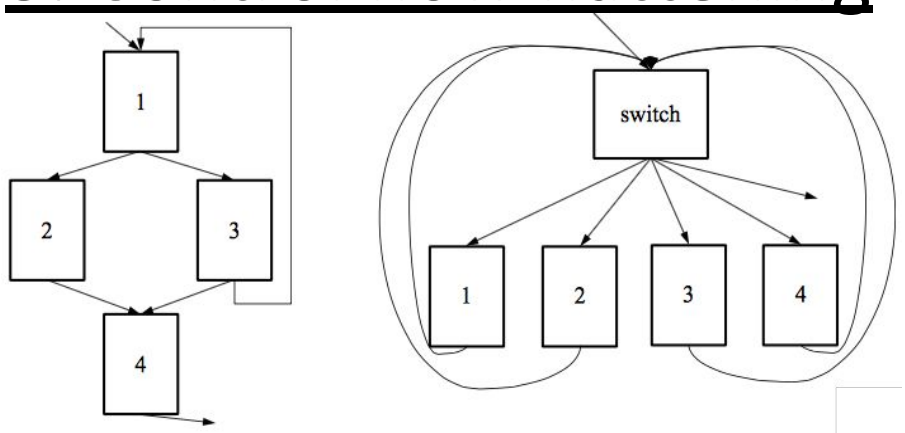
1. Bogus control flow insertion



2. Instruction substitution



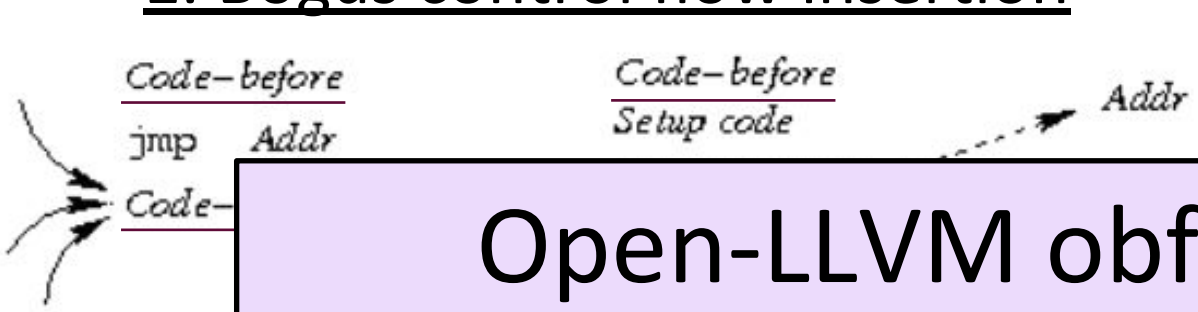
3. Control flow flattening



Obfuscation

1. Bogus control flow insertion

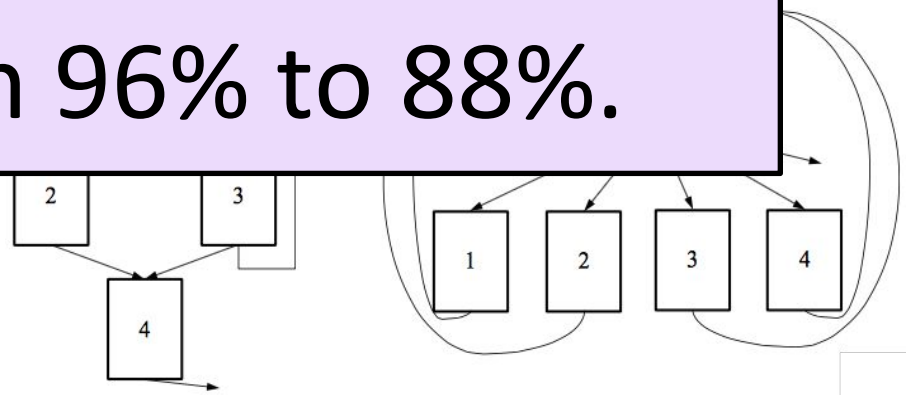
2. Instruction substitution



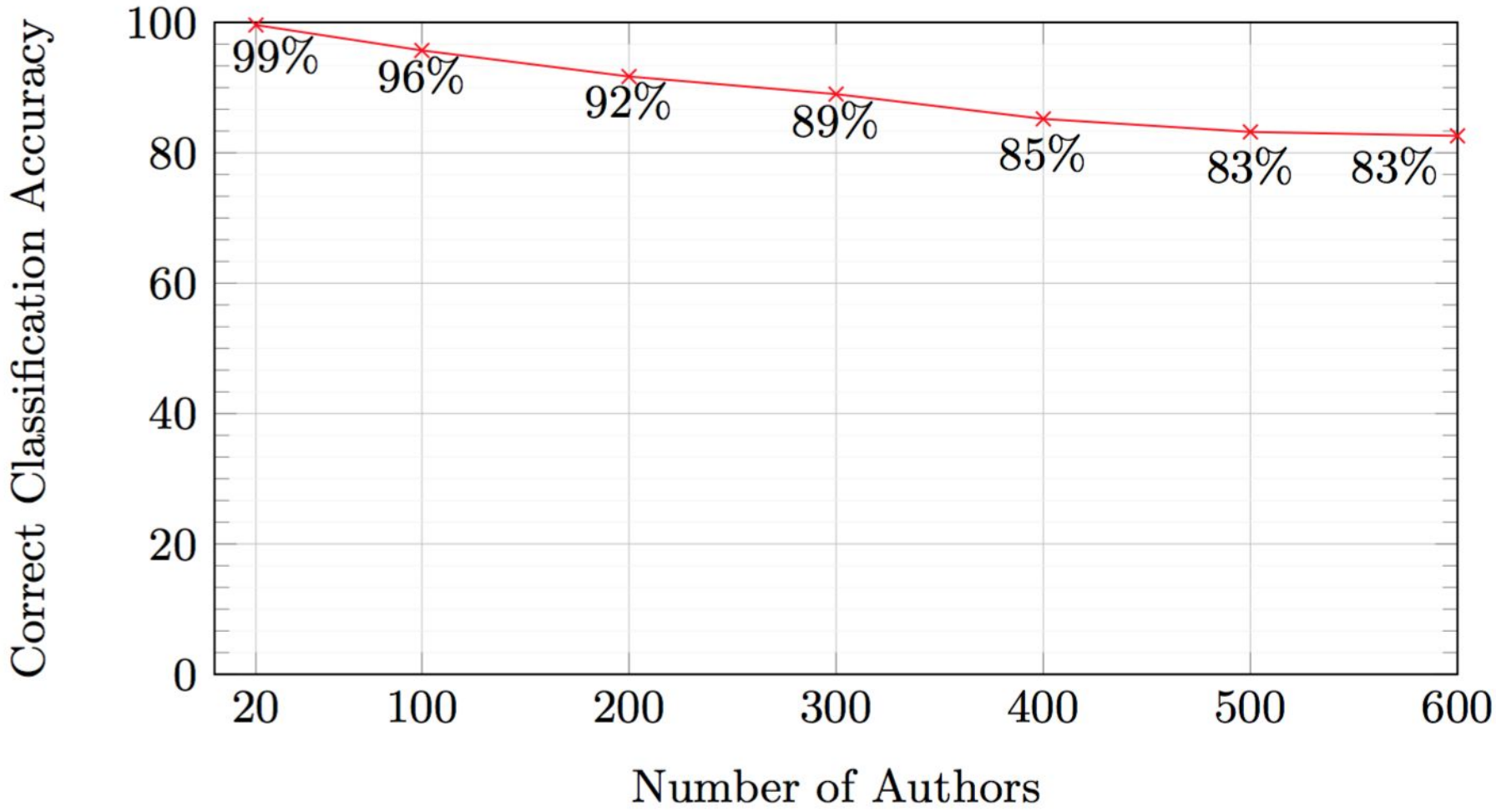

```
mov     eax, [rbp+var_14]
add     eax, 61h
mov     cl, al
movsxd  rdx, [rbp+var_14]
```

```
mov     eax, 61h
mov     ecx, [rbp+var_14]
sub     eax, 5EC7E8EEh
add     eax, ecx
```

Open-LLVM obfuscations reduce de-anonymization accuracy of 100 programmers from 96% to 88%.



Large scale programmer de-anonymization



GitHub and Nulled.IO

- De-anonymizing 50 GitHub programmers
  - with 65% accuracy.

- De-anonymizing 6 malicious programmers
  - Nulled.IO hackers and malware authors
  - with 100% accuracy.

Programmer De-anonymization on GitHub

- ✓ Single authored GitHub repositories
- ✓ The repository has at least 500 lines of code

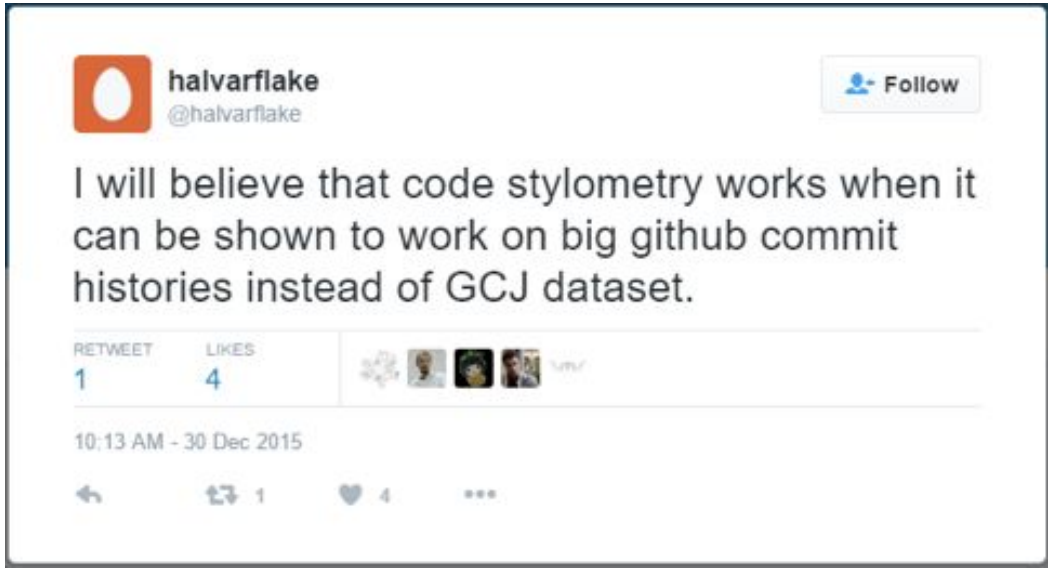
Type	Amount
Authors	161
Repositories	439
Files	3,438
Repositories / Author	2 - 8
Files / Author	2 - 344

Compile repositories

Dataset	Authors	Total Files	Accuracy
GitHub	50	542	65%
GCJ	50	450	97%



Collaborative Code





Segment and Account Attribution

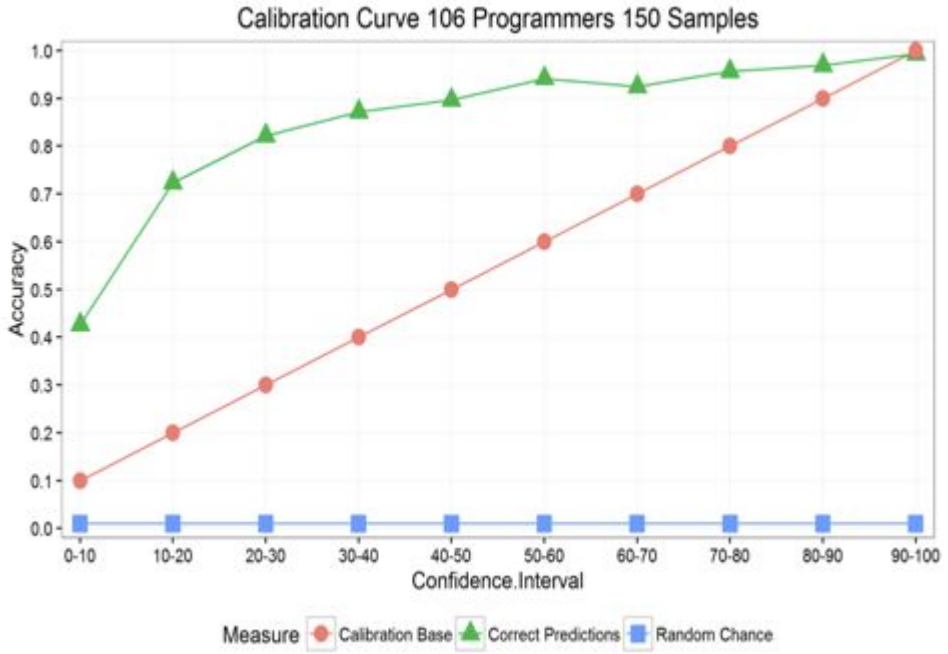
- Sometimes we only care who wrote a small piece of code
- Sometimes we want to deanonymize a pseudonymous account
  - Without whole files belonging to it, only small pieces
- In these cases, we can only attribute small segments, or “snippets”

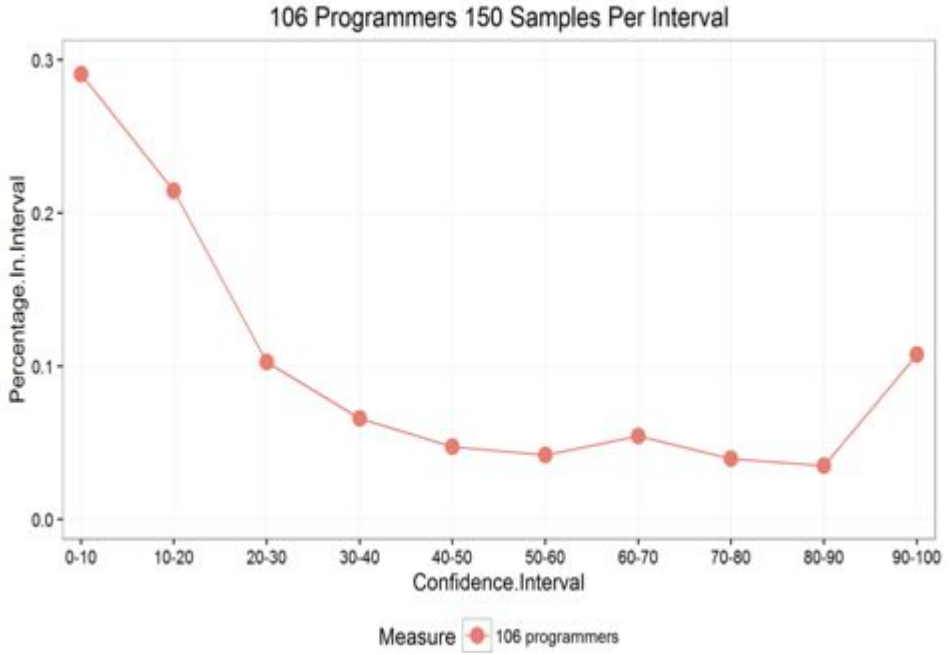
- Using the manual feature set
  - Large, sparse features (3,407 nonzero out of 369,097 total)

Segment attribution results

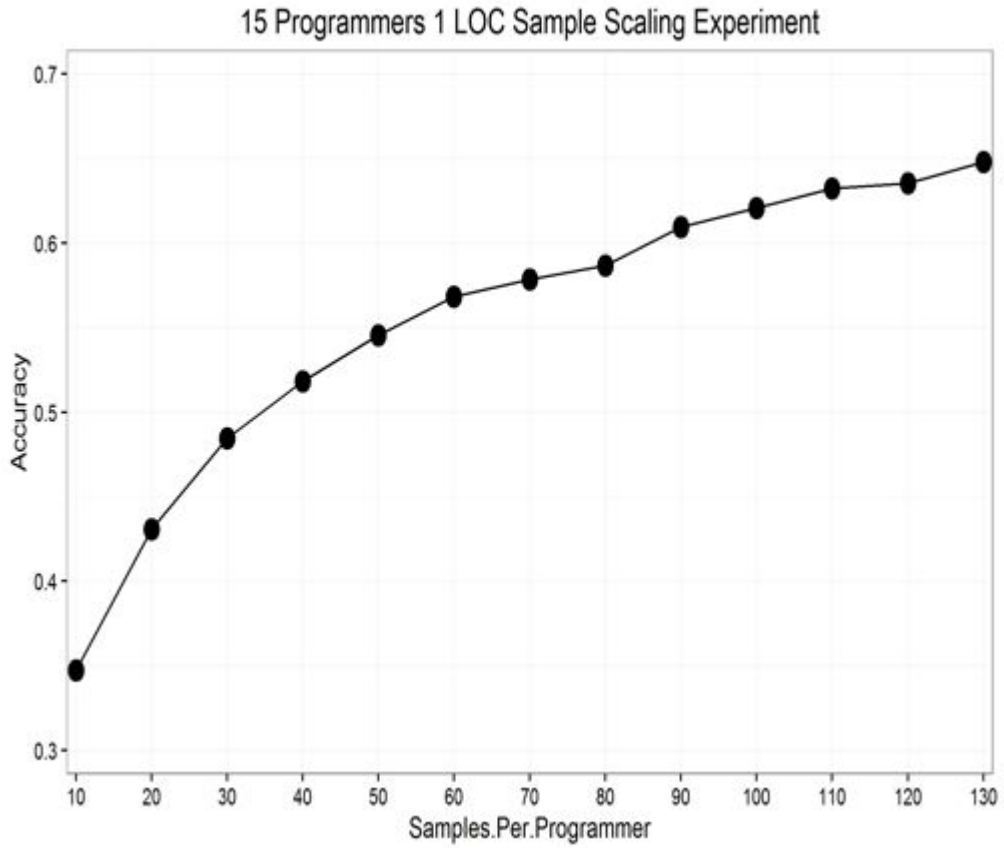
73% accuracy

(average sample 4.9 lines of code)





Accuracy vs LOC

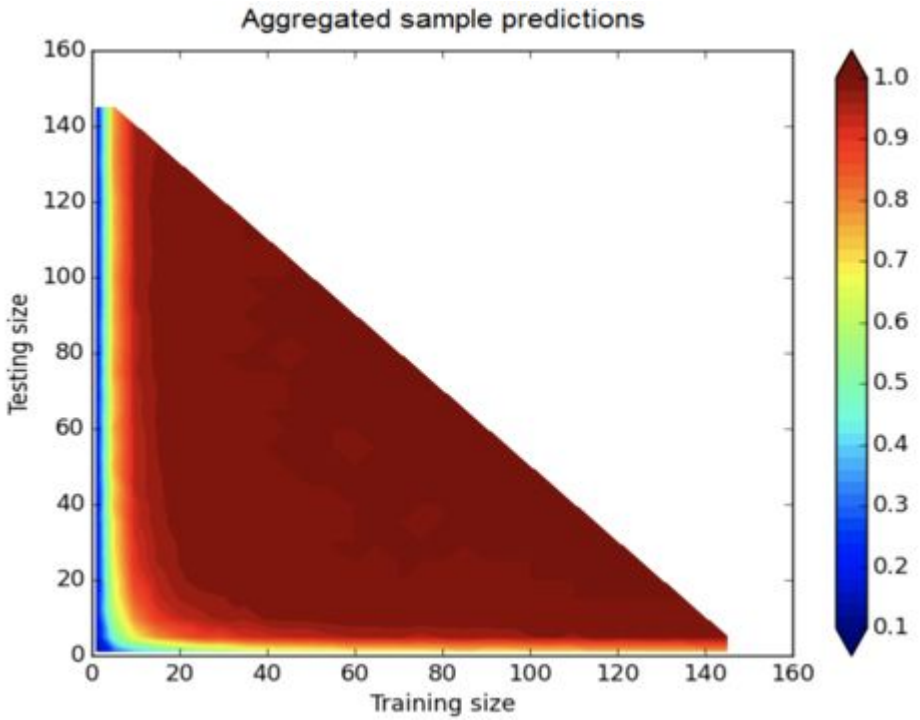


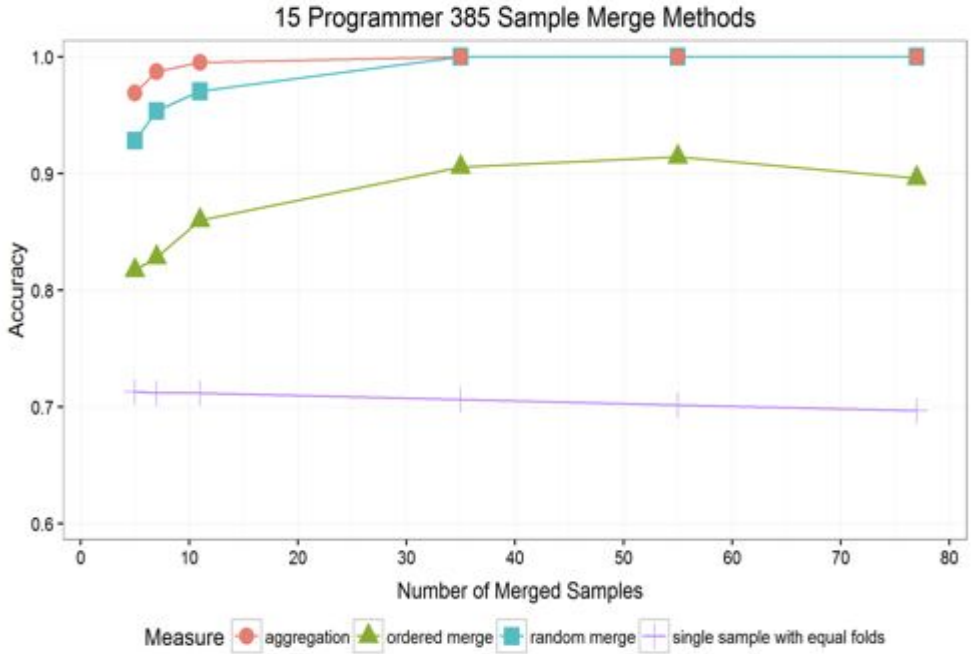
Samples	min LOC	Programmers	Accuracy
4	38	90	54%
6	28	90	63%
10	18	90	76%
23	8	90	75%
90	3	90	77%
150	1	90	75%

Attribute accounts not individual commits?

Works much much better!

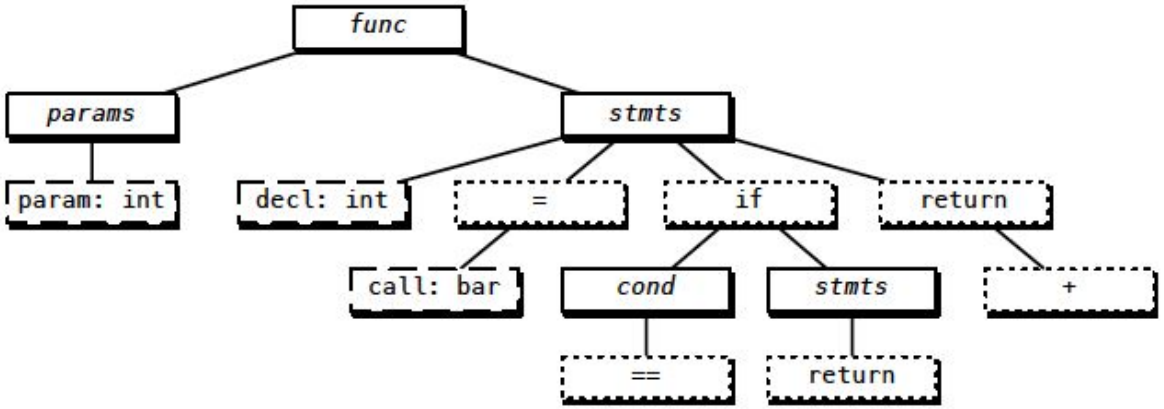
- close to 100% after 4 snippets





Deep Learning AST Representations

Using AST features allowed us to get good results.

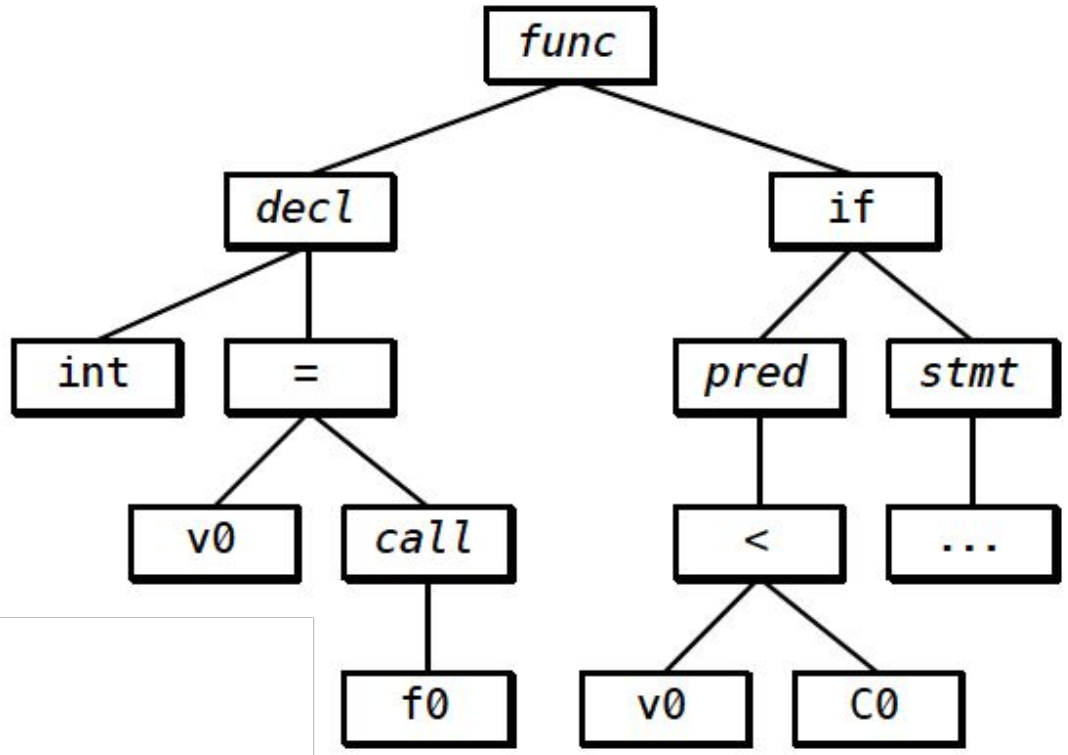


But….

A Tree is not a feature!

We manually chose features of the ASTs

Abstract syntax tree (AST)



Syntactic features

AST unigrams:

func	decl	if	int
=	pred	stmt	...

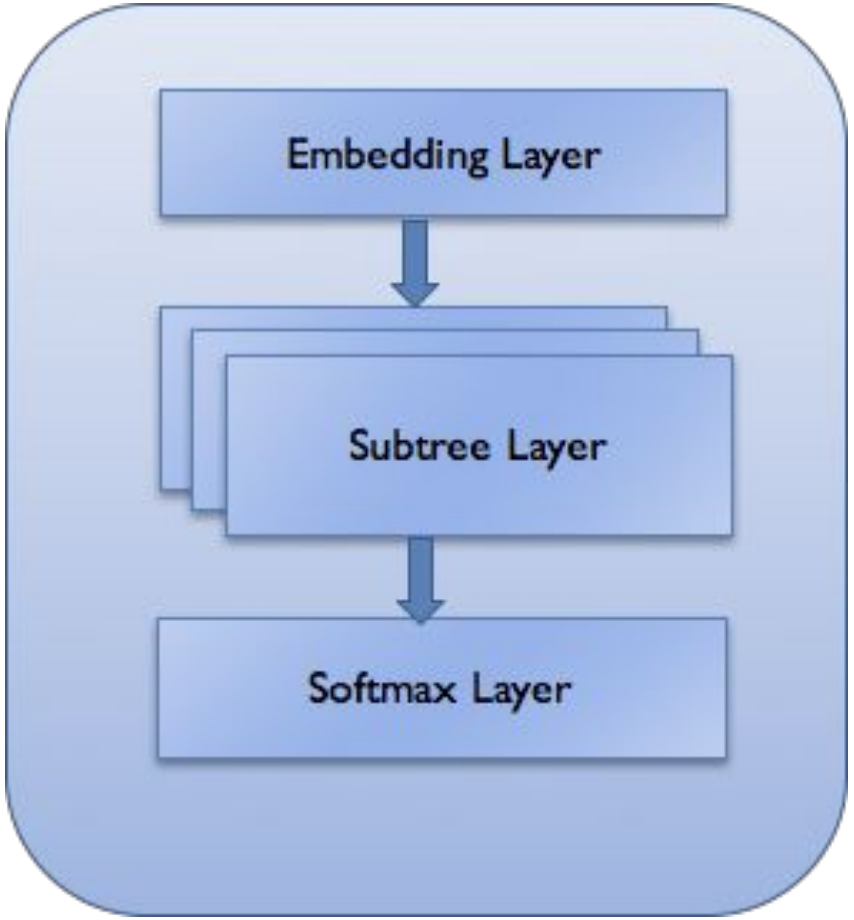
AST bigrams:

func	func	decl	...
decl	if	int	

AST depth: 5

Can a deep neural net do better?

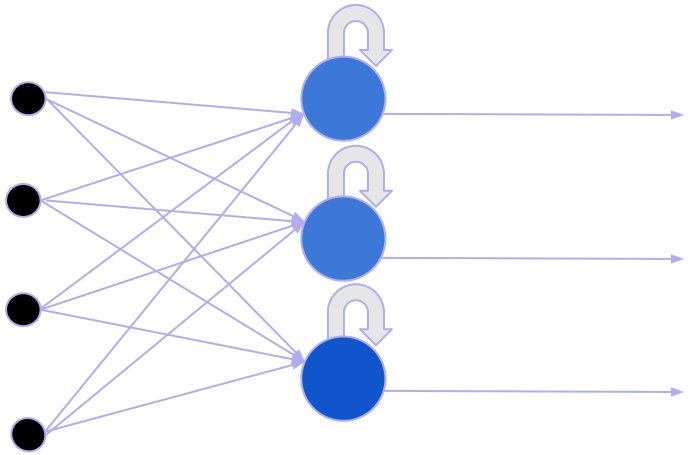
- Embedding Layer
  - Map AST nodes to feature vectors
- Subtree Layers
  - Learn the structure of the AST
    - Subtree LSTM
    - Subtree BiLSTM (bidirectional)
- Softmax Layer
  - Generate a probability distribution of the programmers



Long Short-Term Memory Networks

Recurrent neural networks (RNNs)

- Handle sequential input
- Add feedback loops to remember information



LSTMs add memory cells

- Sequential long-term dependencies
- Use gates to control flow of information

What should I remember?

What should I ignore?

What should I forget?

Results

Using only AST features (No lexical or layout features)

	Python (25 programmers)	Python (70 programmers)	C++ (10 programmers)
Random Forest	86.00	72.90	75.90
Linear SVM	77.20	61.28	73.50
LSTM	92.00	86.36	80.00
BiLSTM	96.00	88.86	85.00

So what?

- Learn better AST representations without feature engineering
- Language independent - any programming language that supports ASTs

Future work

- Combine with Random Forests and fuller feature sets
  - Better results or just overlap with other features?

What about other languages?

Porting requires AST parser and lexical/layout features









Similar accuracies so far (on GCJ dataset)
Results with just AST vary

Train on one language test on another?

- This is something we'd like to try
- Need either universal intermediate AST representation or pairwise
- Babblefish project (doesn't appear to be ready yet)



Interesting Software Engineering Insights

What about attributing groups?

Looked at CODEFORCES β Sponsored by Telegram team programming competition

Teams compete on sets of problems

Preliminary results:

118 Codeforces teams, at least 20 submissions each

- 10-fold cross-validation: 67.2% accuracy
- 20-fold cross-validation: 67.8% accuracy

Difficult because they are likely splitting up the problems completely

Future work: code repositories

Difficult vs. Easy Tasks

Implementing harder functionality makes programming style more unique.

Same set of 62 programmers	Classification Accuracy
Solving 7 Easy Problems	90%
Solving 7 Harder Problems	95%

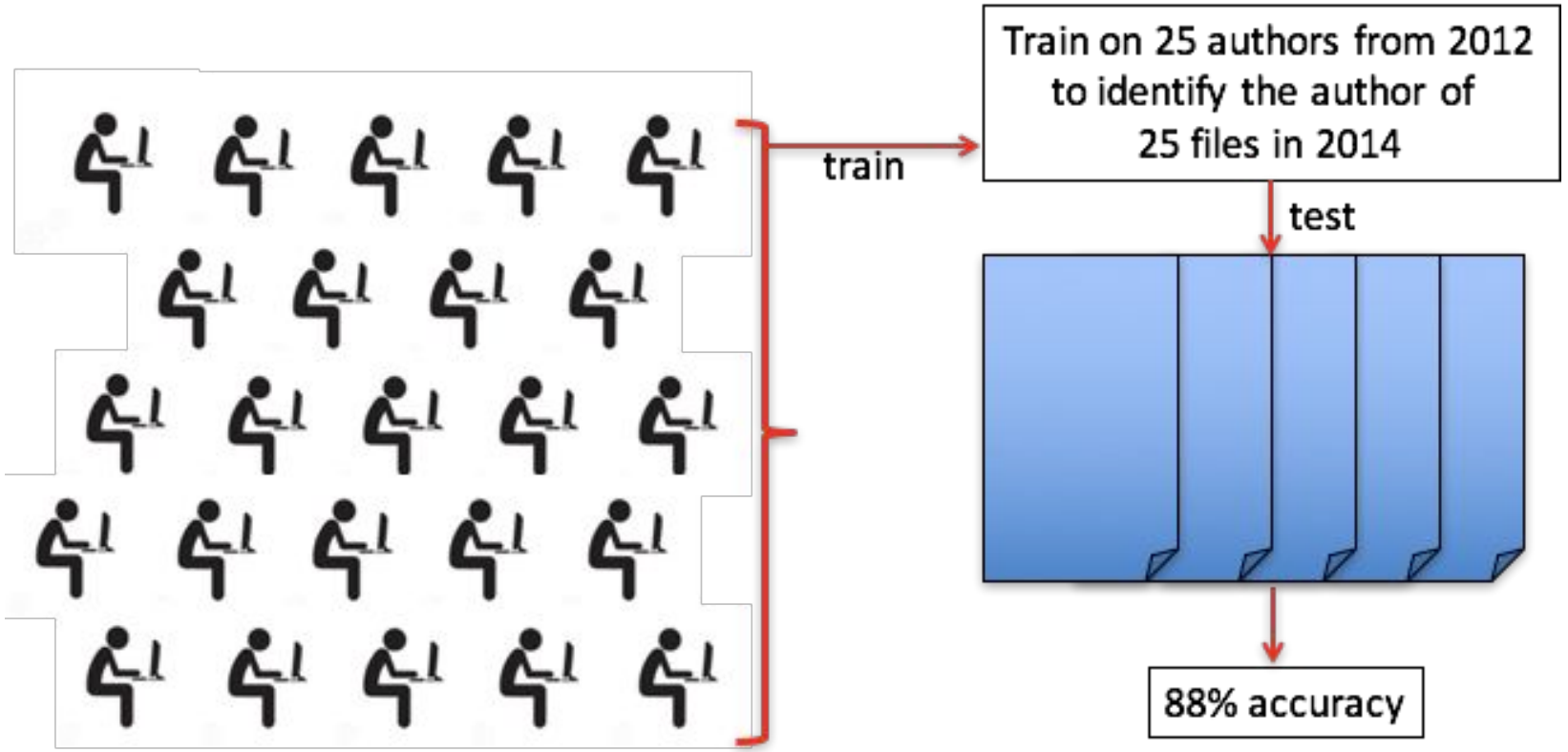
Effect of Programming Skill?

Programmers who got further in the GCJ Contest were easier to attribute.

Same set of 62 programmers	Classification Accuracy
Less Advanced Coders	80%
More Advanced Coders	95%

How does coding style change over time?

- 92% accuracy, train and test on 2012
- 88% accuracy, train on 2012, test on 2014



Coding style by country?



GCJ files (in javascript) written by programmers in Canada and China

- 84 files
- 91.9% classification accuracy

Future Applications

- Find malicious code authors
  - anonymous contributors
- Write better obfuscators
  - target AST directly
- Find authors who write vulnerable code
  - open source code
- Find who to recruit directly
  - from git commits

Thanks to collaborators

Bander Alsulami, Edwin Dauber, Richard Harang, Andrew Liu, Spiros Mancoridis, Arvind Narayanan, Frederica Nelson, Mosfiqur Rahman, Dennis Rollke, Konrad Rieck, Gregory G. Shearer, Clare Voss, Michael J. Weisman, Fabian Yamaguchi

Contact information and Q & A

Aylin Caliskan

@aylin_cim
aylin@gwu.edu

Rachel Greenstadt

@ragreens
greenstadt@gmail.com

Source code authorship attribution: https://github.com/calaylin/bda

Javascript authorship attribution:
https://github.com/dns43/CodeStylometry/tree/master/SCAA/src

Binary authorship attribution: https://github.com/calaylin/bda

Comparison to related work

Related Work	Author Size	Instances	Average LOC	Language	Fetaures	Method	Result
MacDonell et al.	7	351	148	C++	lexical & layout	Case-based reasoning	88.0%
Frantzeskou et al.	8	107	145	Java	lexical & layout	Nearest neighbor	100.0%
Elenbogen and Seliya	12	83	100	C++	lexical & layout	C4.5 decision tree	74.7%
Shevertalov et. al.	20	N/A	N/A	Java	lexical & layout	Genetic algorithm	80%
Frantzeskou et al.	30	333	172	Java	lexical & layout	Nearest neighbor	96.9%
Ding and Samadzadeh	46	225	N/A	Java	lexical & layout	Nearest neighbor	75.2%
Ours	**35**	**315**	**68**	**C++**	**lexical & layout & syntactic**	**Random forest**	**100.0%**
Ours	**250**	**2,250**	**77**	**C++**			**98.0%**
Ours	**1,600**	**14,400**	**70**	**C++**			**93.6%**

Comparison to related work

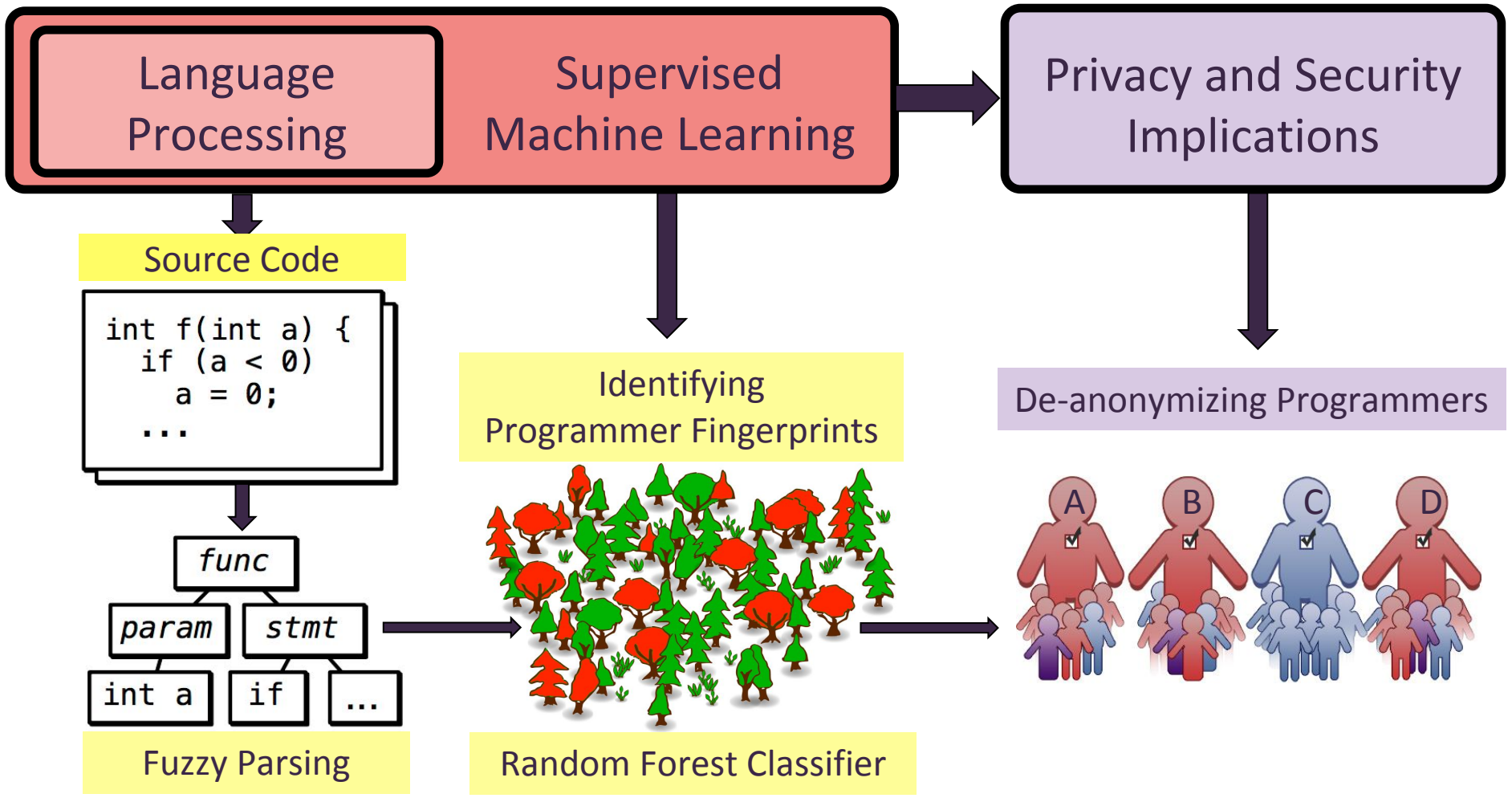
Related Work	Author Size	Instances	Average LOC	Language	Fetaures	Method	Result
Frantzeskou et al.	30	333	172	Java	lexical & layout	Nearest neighbor	96.9%
Ding and Samadzadeh	46	225	N/A	Java	lexical & layout	Nearest neighbor	75.2%
Ours	35	315	68	C++	lexical & layout & syntactic	Random forest	100.0%
Ours	250	2,250	77	C++			98.0%
Ours	1,600	14,400	70	C++			93.6%

Comparison to related work

Related Work	Author Size	Instances	Average LOC	Language	Fetaures	Method	Result
Frantzeskou et al.	30	333	172	Java	lexical & layout	Nearest neighbor	96.9%
Ding and Samadzadeh	46	225	N/A	Java	lexical & layout	Nearest neighbor	75.2%
Ours	35	315	68	C++	lexical & layout & syntactic	Random forest	100.0%
Ours	250	2,250	77	C++			98.0%
Ours	1,600	14,400	70	C++			93.6%

Comparison to related work

Related Work	Author Size	Instances	Average LOC	Language	Fetaures	Method	Result
MacDonell et al.	7	351	148	C++	lexical & layout	Case-based reasoning	88.0%
Frantzeskou et al.	8	107	145	Java	lexical & layout	Nearest neighbor	100.0%
Elenbogen and Seliya	12	83	100	C++	lexical & layout	C4.5 decision tree	74.7%
Shevertalov et. al.	20	N/A	N/A	Java	lexical & layout	Genetic algorithm	80%
Frantzeskou et al.	30	333	172	Java	lexical & layout	Nearest neighbor	96.9%
Ding and Samadzadeh	46	225	N/A	Java	lexical & layout	Nearest neighbor	55.2%
Ours	**35**	**315**	**68**	**C++**	**lexical & layout & syntactic**	**Random forest**	**100.0%**
Ours	**250**	**2,250**	**77**	**C++**		**Random forest**	**98.0%**

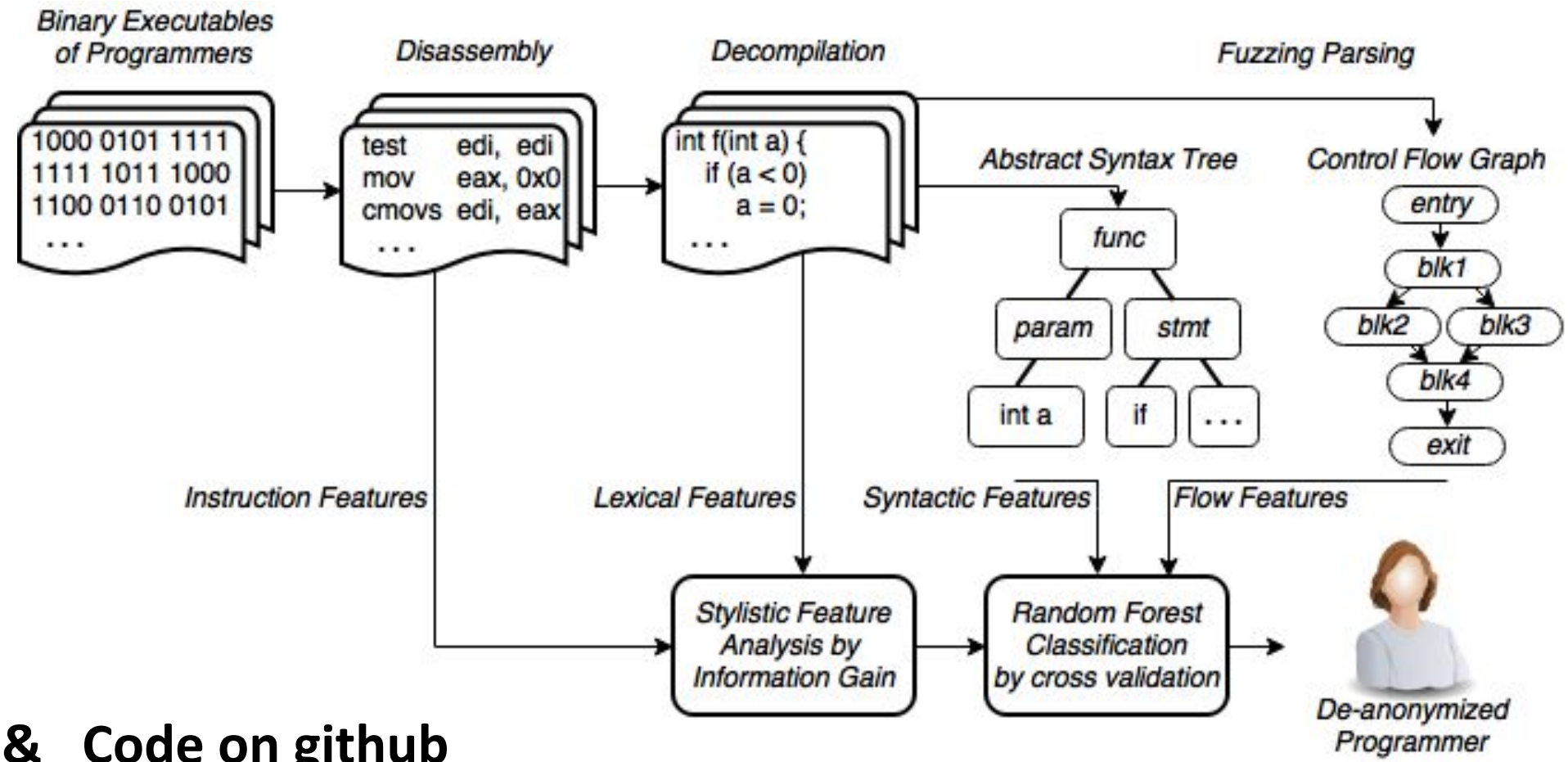


Publications

Usenix 2015:

Aylin Caliskan-Islam, Richard Harang, Andrew Liu, Arvind Narayanan, Clare Voss, Fabian Yamaguchi, and Rachel Greenstadt.

De-anonymizing Programmers via Code Stylometry. 24th Usenix Security Symposium (Usenix 2015).



Publications & Code on github

NDSS 2018:

Aylin Caliskan, Fabian Yamaguchi, Edwin Dauber, Richard Harang, Konrad Rieck, Rachel Greenstadt, and Arvind Narayanan. When Coding Style Survives Compilation: De-anonymizing Programmers from Executable Binaries (NDSS 2018).

Usenix 2015:

Aylin Caliskan-Islam, Richard Harang, Andrew Liu, Arvind Narayanan, Clare Voss, Fabian Yamaguchi, and Rachel Greenstadt.
De-anonymizing Programmers via Code Stylometry. 24th Usenix Security Symposium (Usenix 2015).

Source code stylometry

- Everyone learns coding on an individual basis, as a result code in a unique way,

  which makes de-anonymization possible.

- Software engineering insights
  - programmer style changes while implementing sophisticated functionality
  - differences in coding styles of programmers with different skill sets

- Identify malicious programmers.

Case 2: Obfuscation

- Who is the programmer of this obfuscated source code?
- Code is obfuscated to become unrecognizable.
- Our authorship attribution technique is impervious to common off-the-shelf source code obfuscators.

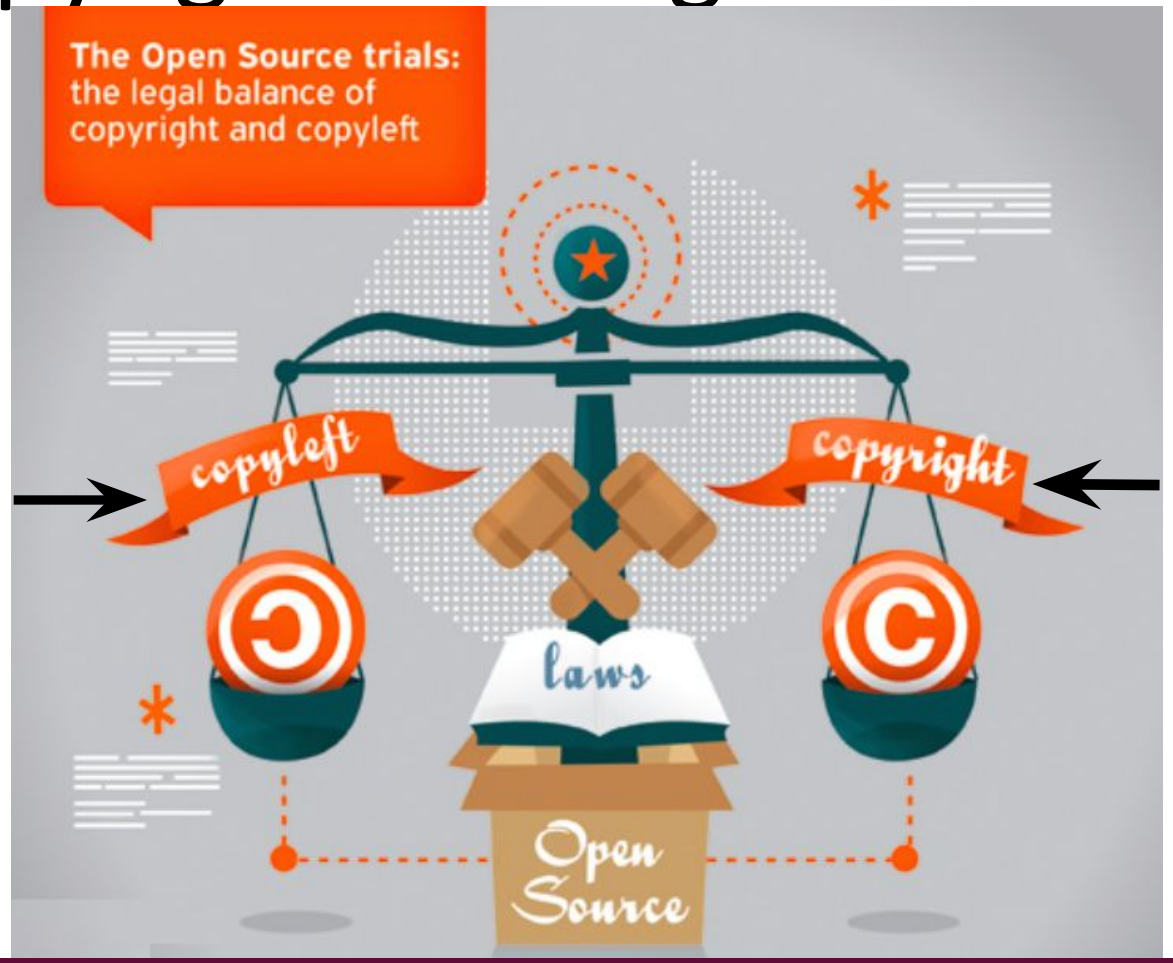
Case 3: Copyright investigation

- Copyleft programs are free but licensed
- Did this programmer take a copyleft code and distribute it commercially?
  - ***Jacobsen vs Katzer (Java Model Railroad Interface)***
- Two-class machine learning classification task
  - Class 1: the copyleft code
  - Class 2: the copyright code



Case 3: Copyright investigation

Jacobsen → ← **Katzer**



30 pairs of authors each with 9 program files	Classification Accuracy
Two-class task	100%

Case 5: Coding style throughout years

- Is programming style consistent?
- If yes, we can use code from different years for authorship attribution.

2012

```
int main()
{
    freopen("a.in", "r", stdin);
    freopen("a.out", "w", stdout);

    int tt;
    scanf("%d", &tt);

    for(int t = 0; t < tt; t++)
    {
        int n;
        scanf("%d", &n);
        ...
```

2014

```
int main()
{
    freopen("a.in", "r", stdin);
    freopen("a.out", "w", stdout);

    int TT;
    scanf("%d", &TT);
    for(int T = 0; T < TT; T++)
    {
        printf("Case #%d: ", T+1);


        ...
```

Case 5: Coding style throughout years

- Is programming style consistent?
- If yes, we can use code from different years for authorship attribution.

2012

```
int main()
{
    freopen("a.in", "r", stdin);
    freopen("a.out", "w", stdout);

    int tt;
    scanf("%d", &tt);

    for(int t = 0; t < tt; t++)
    {
        int n;
        scanf("%d", &n);
        ...
```



2014

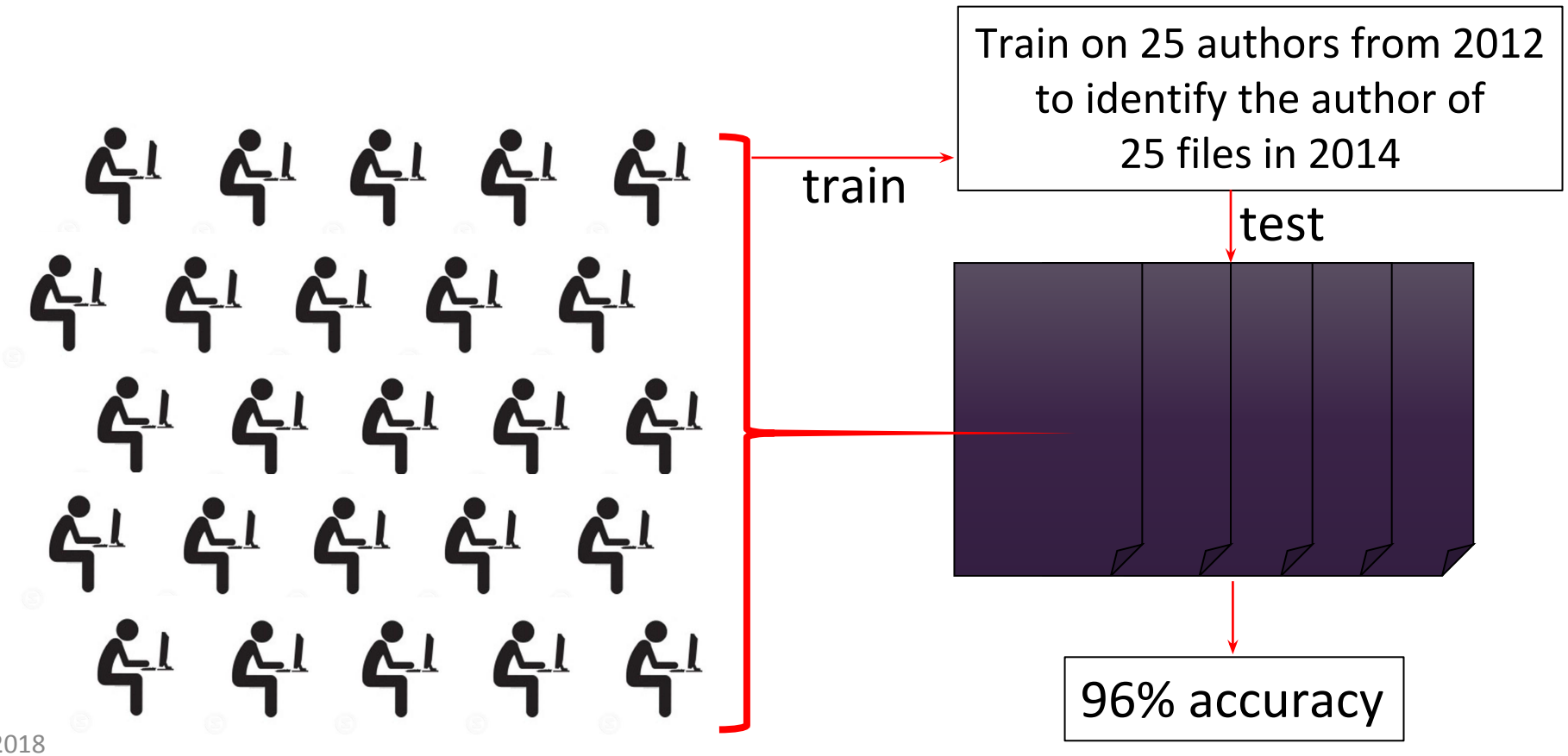
```
int main()
{
    freopen("a.in", "r", stdin);
    freopen("a.out", "w", stdout);

    int TT;
    scanf("%d", &TT);
    for(int T = 0; T < TT; T++)
    {
        printf("Case #%d: ", T+1);


    ...
```

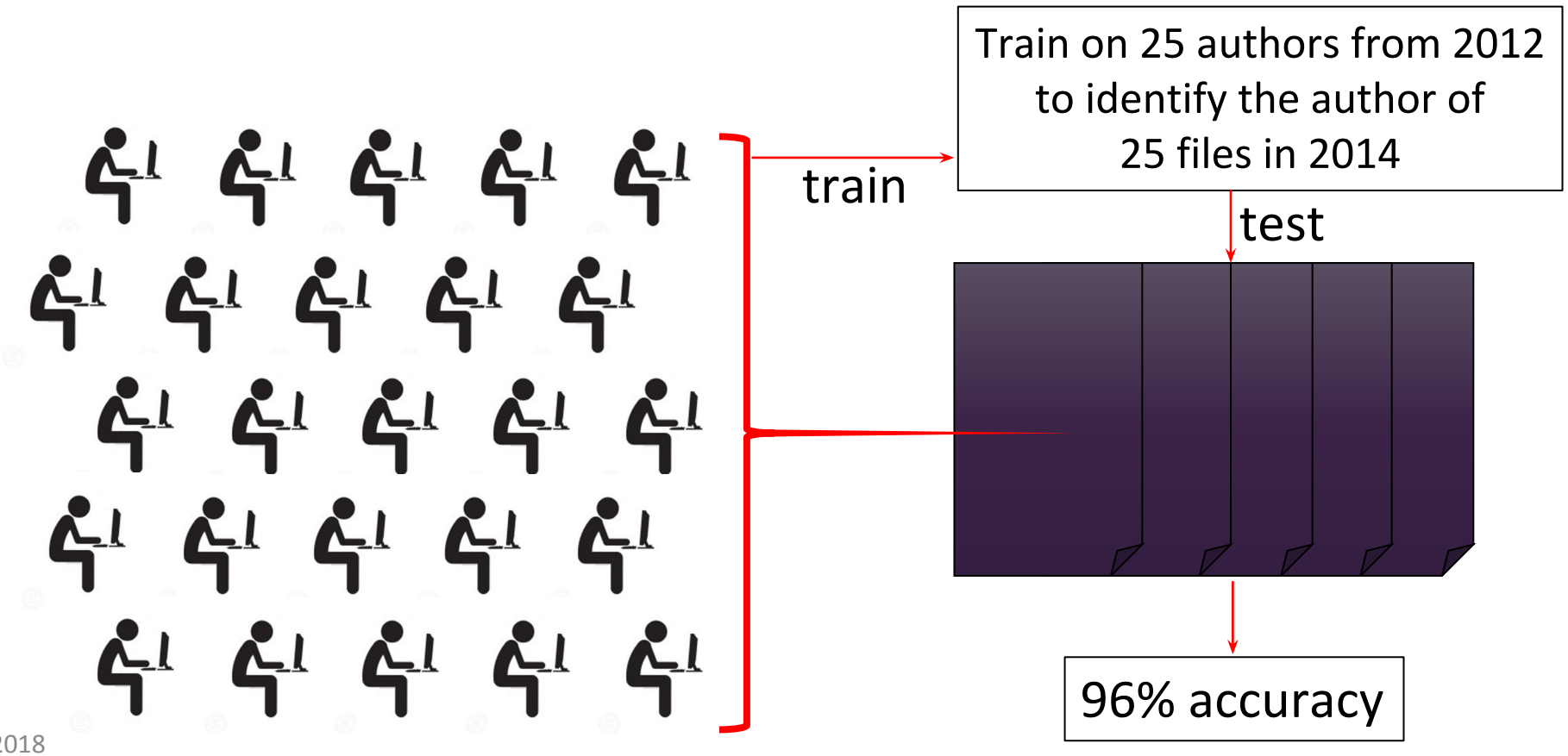
Case 5: Coding style throughout years

- Coding style is preserved up to some degree throughout years.



Case 5: Coding style throughout years

- 98% accuracy, train and test in 2014
- 96% accuracy, train on 2012, test on 2014



Case 6: Difficult tasks & advanced coders

- Insights about programmers and coding style:
  - Implementing harder functionality makes programming style more unique

Same set of 62 authors	Classification Accuracy
Solving 7 easy problems	98%
Solving 7 more difficult problems	99%

Case 6: Difficult tasks & advanced coders

- Insights about programmers and coding style.
  - Better programmers have more distinct coding style

Two sets of 62 authors	Classification Accuracy
Less advanced programmers	97%
More advanced programmers	98%

Case 7: Generalizing the approach - python

Feature set: Using ‘only’ the Python equivalents of syntactic features

Application	Programmers	Instances	Result
Python programmer de-anonymization	229	2,061	53.9%
Top-5 relaxed classification	229	2,061	75.7%
Python programmer de-anonymization	23	207	87.9%
Top-5 relaxed classification	23	207	99.5%

Comparison to related work

600 contestants – C++

code jam

System.out.println("hello, world!");

preprocessing

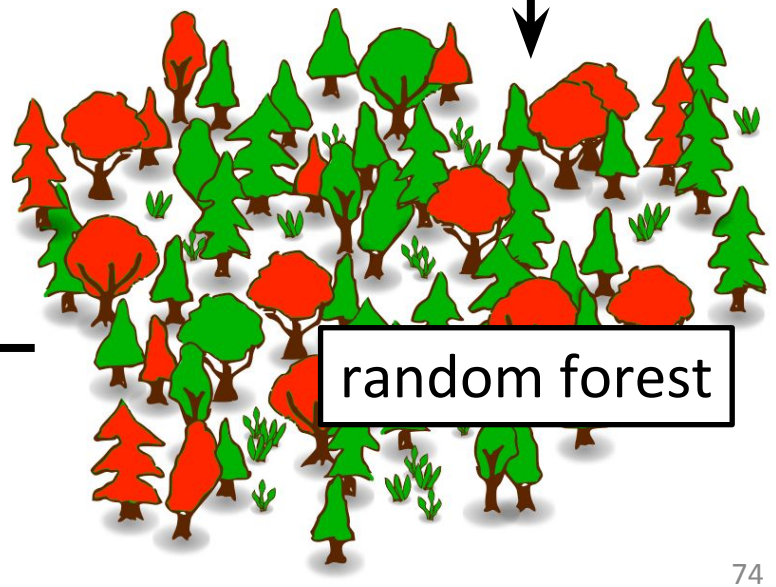


Hex Rays
IDA Pro v6.3

fuzzy AST parser

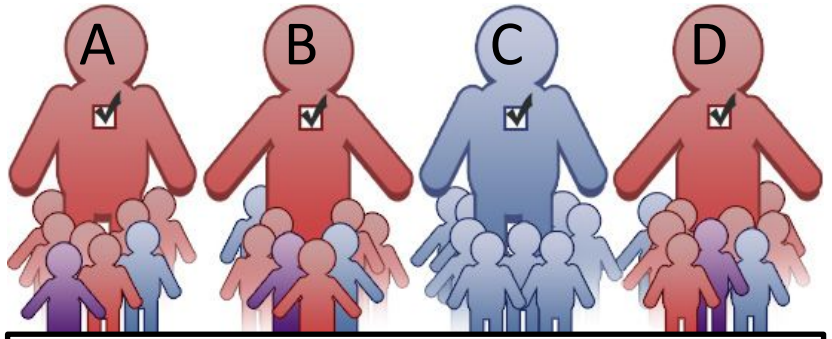
joern

extract features



random forest

majority vote

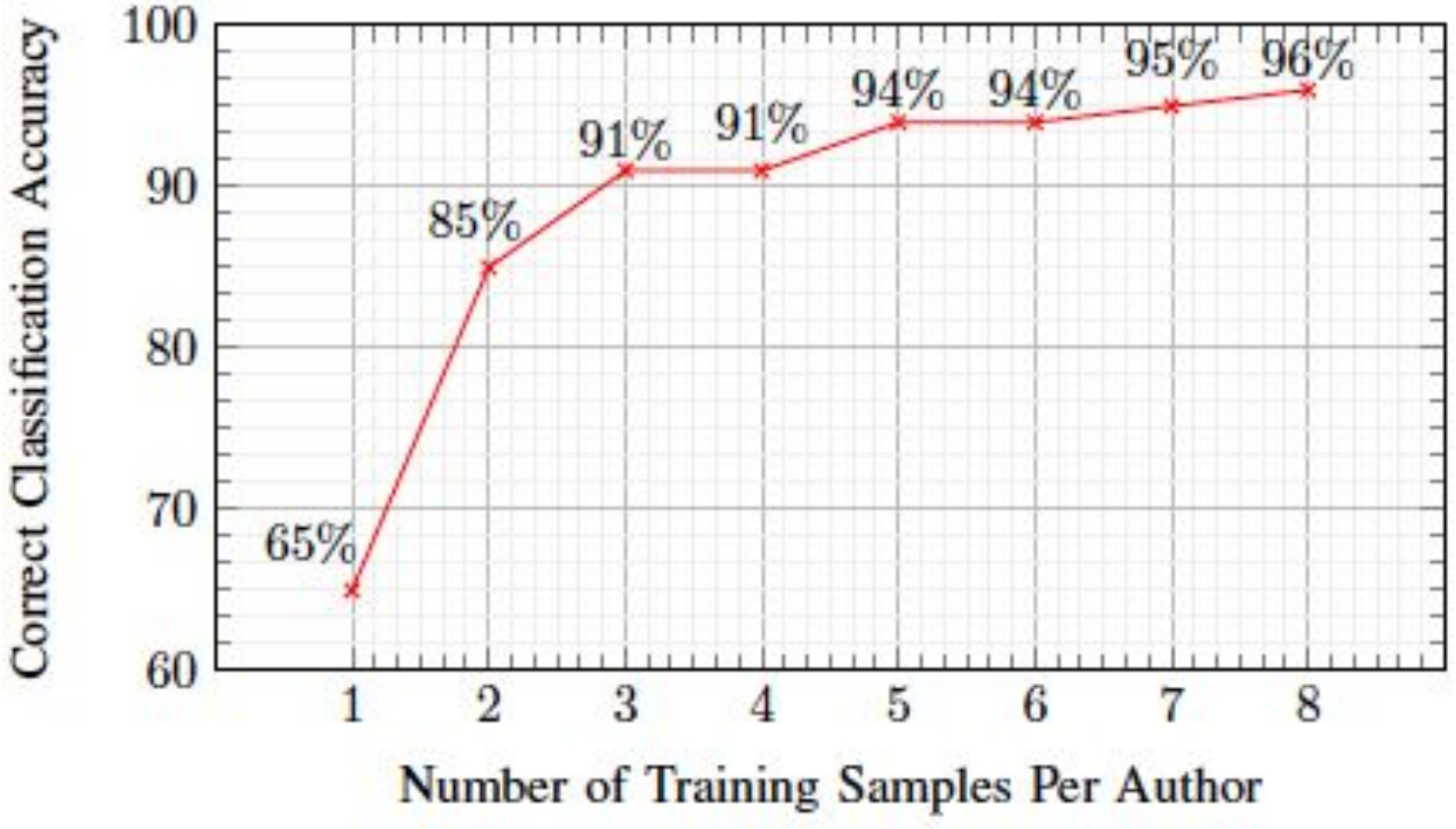


de-anonymized programmers

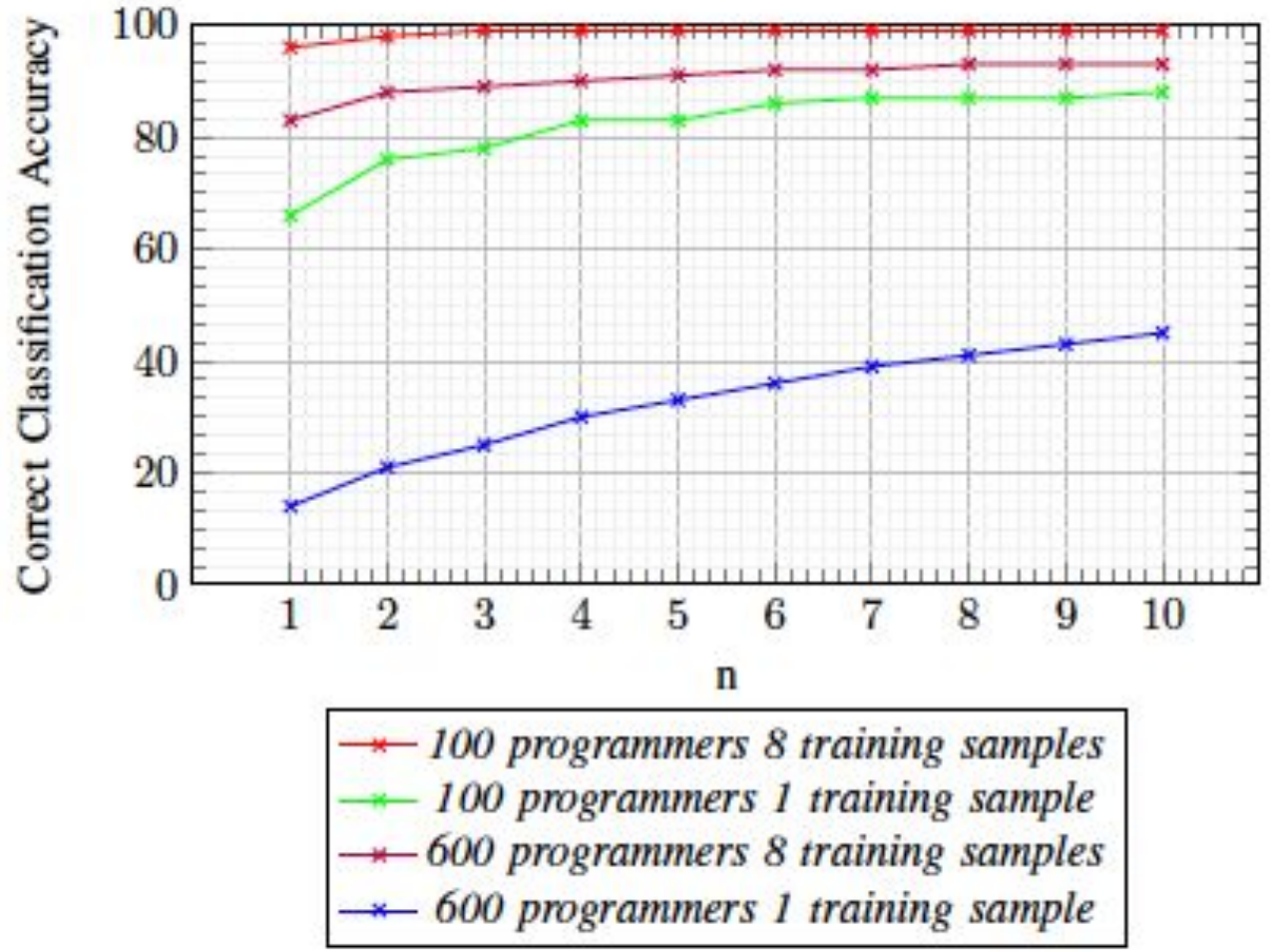
Comparison to related work

Related Work	Number of Programmers	Number of Training Samples	Classifier	Accuracy
Rosenblum et al.	**20**	**8-16**	**SVM**	**77%**
This work	20	8	SVM	90%
This work	20	8	Random forest	99%
Rosenblum et al.	191	8-16	SVM	51%
This work	191	8	Random forest	92%
This work	**600**	**8**	**Random forest**	**83%**

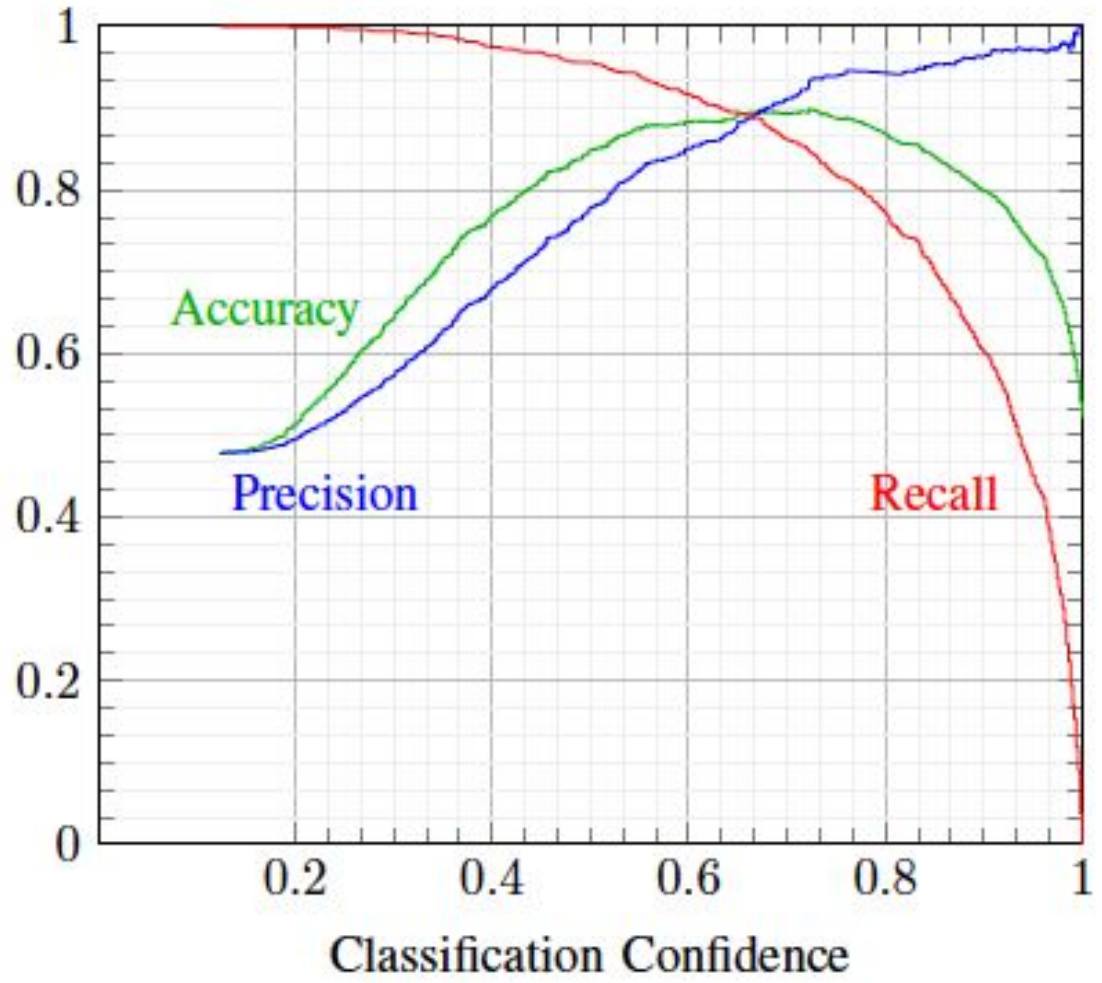
Amount of Training Data Required for De-anonymizing 100 Programmers



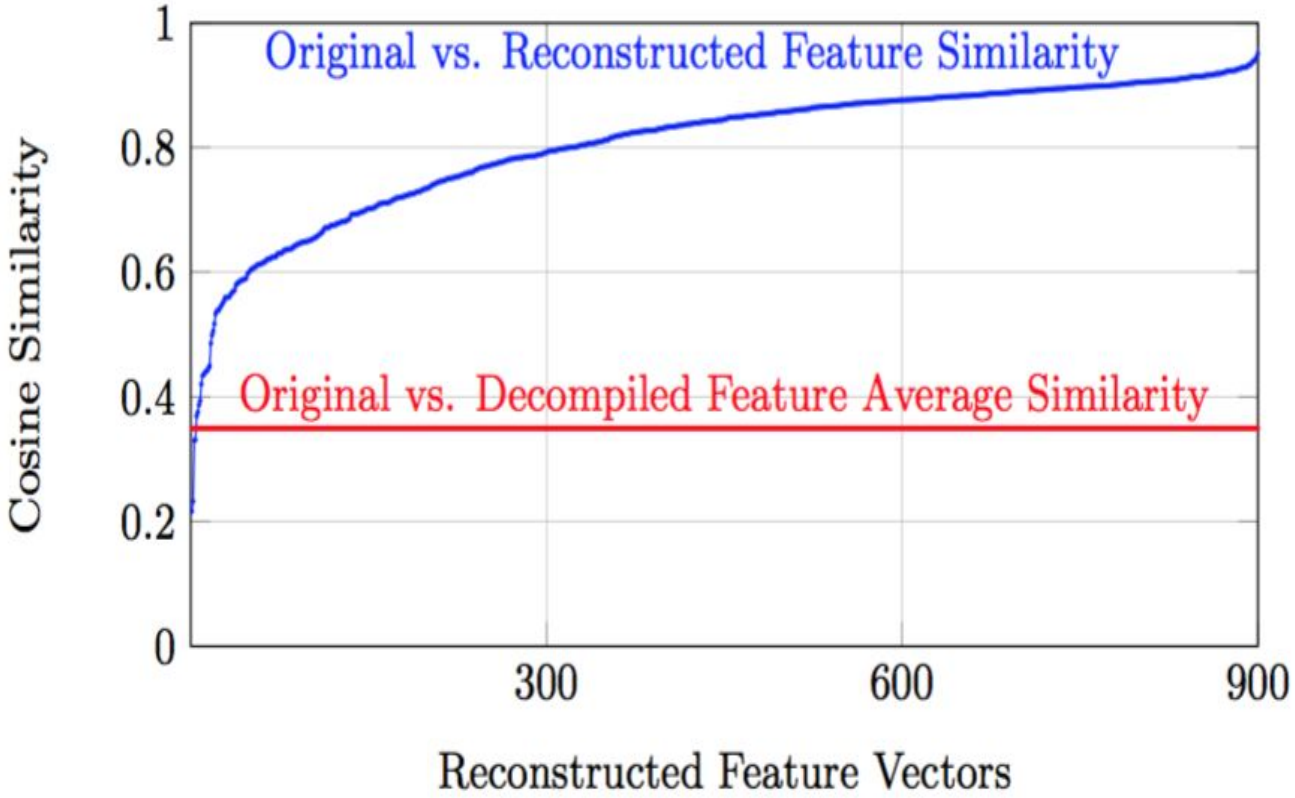
Reducing Suspect Set Size: Top-n Classification



Open world:
Classification thresholds for verification



Reconstructing original features



- Original vs predicted features
  - Average cos similarity: 0.81
- Original vs decompiled features
  - Average cos similarity: 0.35

This suggests that original features are transformed but not entirely lost in compilation.

Ongoing work - DARPA

- Malware author attribution
- Dataset with ground truth
- Automated malware analysis

Future work

- De-anonymizing collaborative code
  - Group fingerprint vs individual fingerprint

- Anonymizing source code
  - Obfuscation is not designed for anonymization

Underground forums

- Micro-text
- L33t sp34k
- Multi-lingual
- Products
- Doppelgänger Finder
  - Carders

