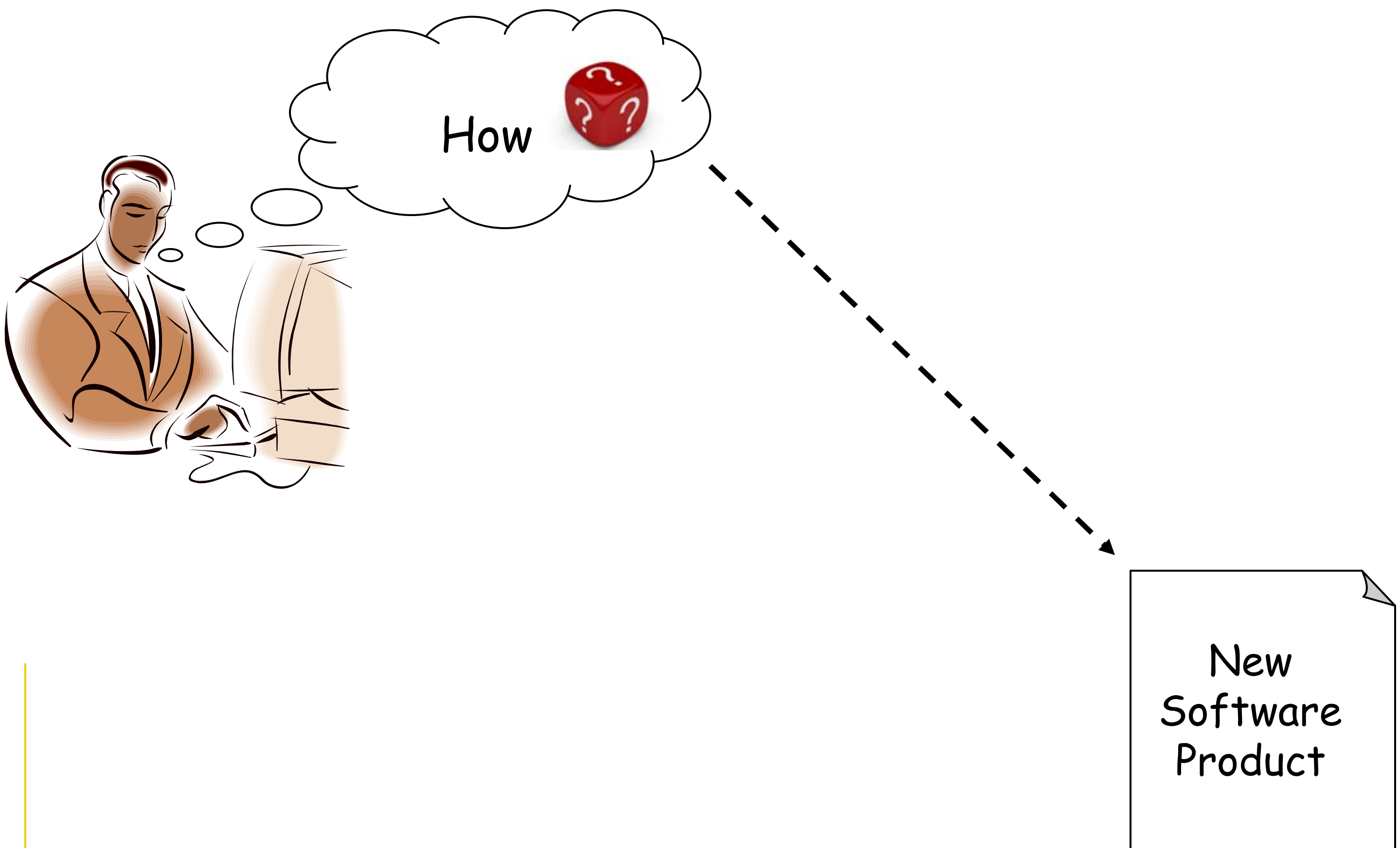
Automatic Mining of Functionally Equivalent Code Fragments via Random Testing

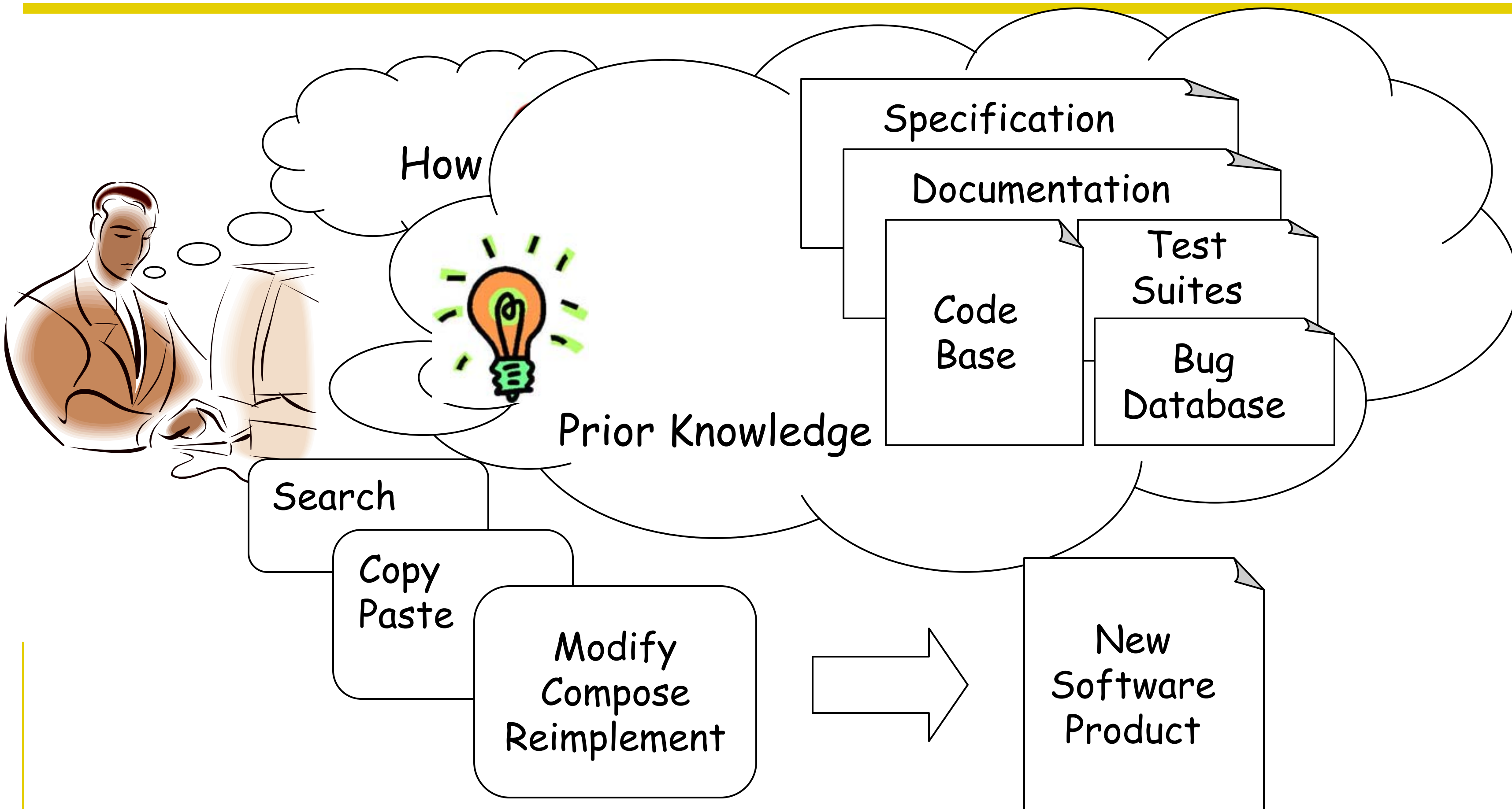
Lingxiao Jiang and Zhendong Su

CSSR
Center for Software and Systems Research
UC DAVIS
COMPUTER SCIENCE

Cloning in Software Development



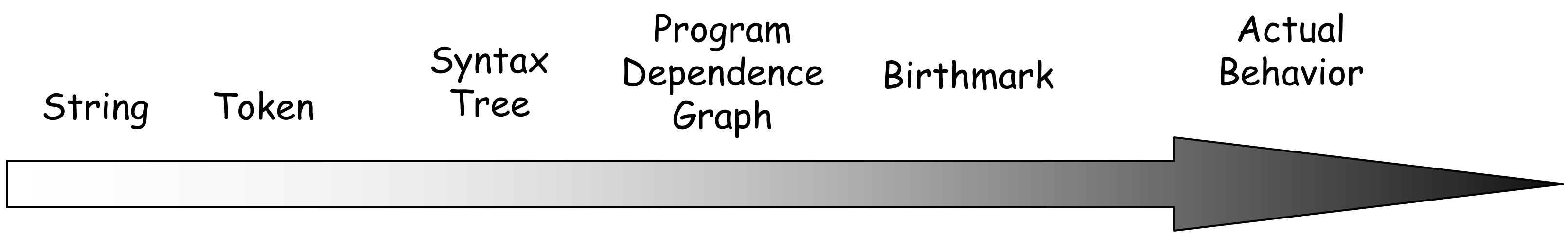
Cloning in Software Development



Applications of Clone Detection

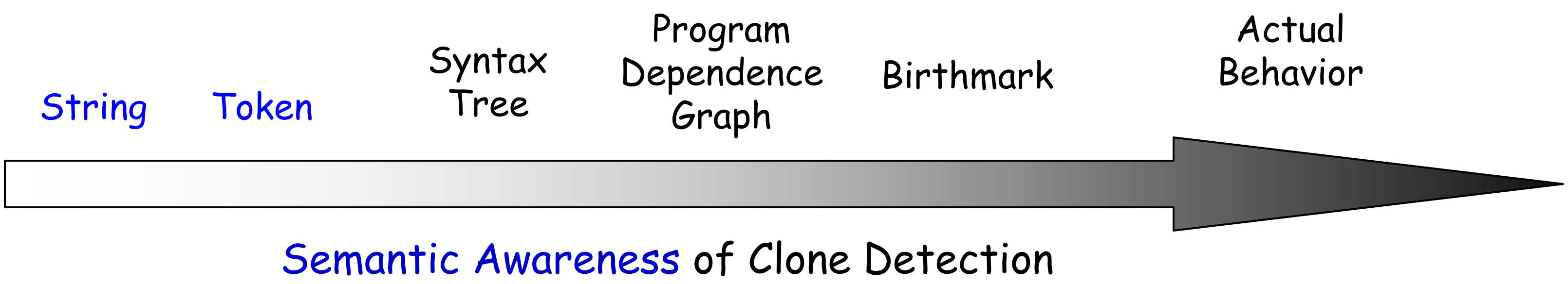
- Refactoring
- Pattern mining
- Reuse
- Debugging
- Evolution study
- Plagiarism detection

A Spectrum of Clone Detection



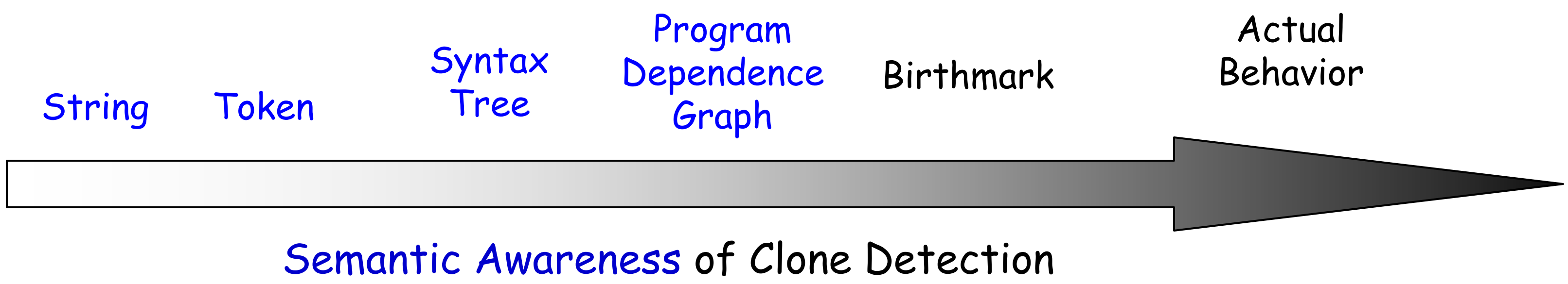
Semantic Awareness of Clone Detection

A Spectrum of Clone Detection



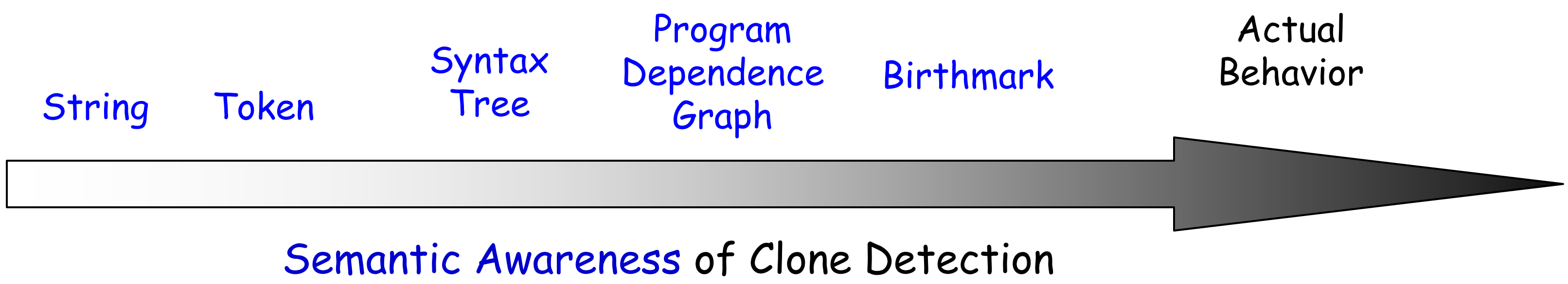
- 1992: Baker, parameterized string algorithm
- 2002: Kamiya et al., CCFinder
- 2004: Li et al., CP-Miner
- 2007: Basit et al., Repeated Tokens Finder

A Spectrum of Clone Detection



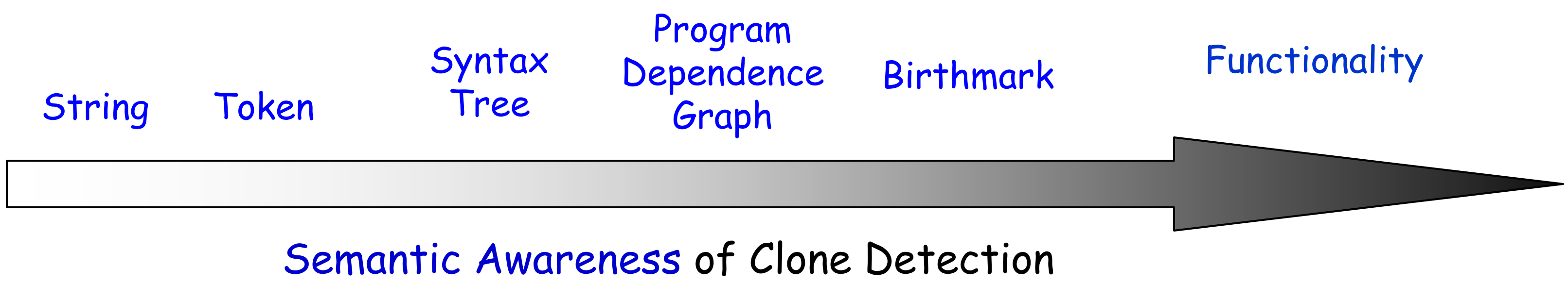
- 1998: Baxter et al., CloneDR
- 2004: Wahler et al., XML-based
- 2007: Jiang et al., Deckard
- 2000, 2001: Komondoor et al.
- 2006: Liu et al., GPLAG
- 2008: Gabel et al.

A Spectrum of Clone Detection



- 1999: Collberg et al., Software watermarking
- 2007: Schuler et al., Dynamic birthmarking
- 2008: Lim et al., Static birthmarking
- 2008: Zhou et al., Combined approach

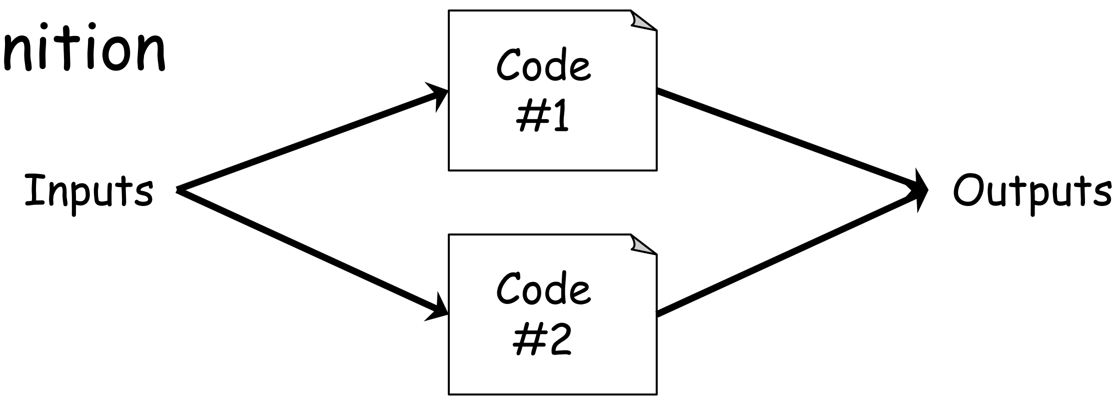
A Spectrum of Clone Detection



- Functional equivalence
  - How extensive is its existence

Functional Equivalence

- Definition



- Applicability: arbitrary piece of code
  - Source and binary
  - From whole program to whole function to code fragments
- Example: sorting algorithms
  - Bubble, selection, merge, quick, heap

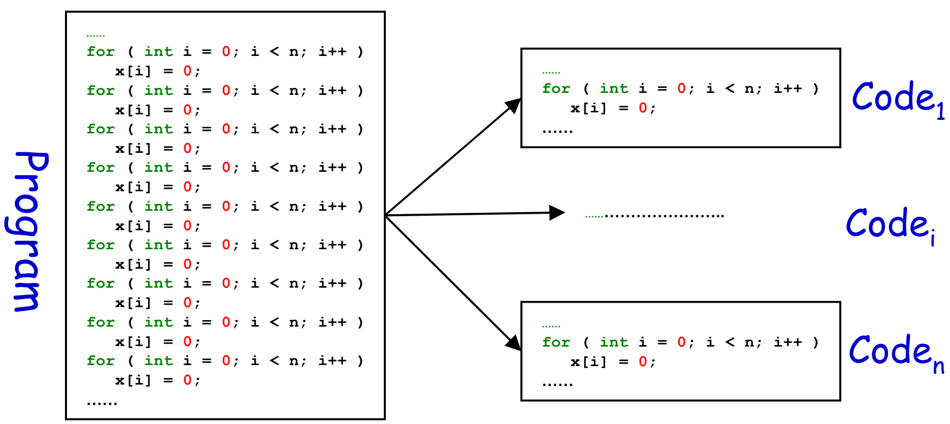
Previous Work on Program Equivalence

- [Cousineau 1979; Raoult 1980; Zakharov 1987; Crole 1995; Pitts 2002; Bertran 2005; Matsumoto 2006; Siegel 2008; …]

- Many based on formal semantics
- Consider whole programs or functions only
  - Not arbitrary code fragments
- Check equivalence among given pieces of code
  - Not scalable detection

Our Objectives

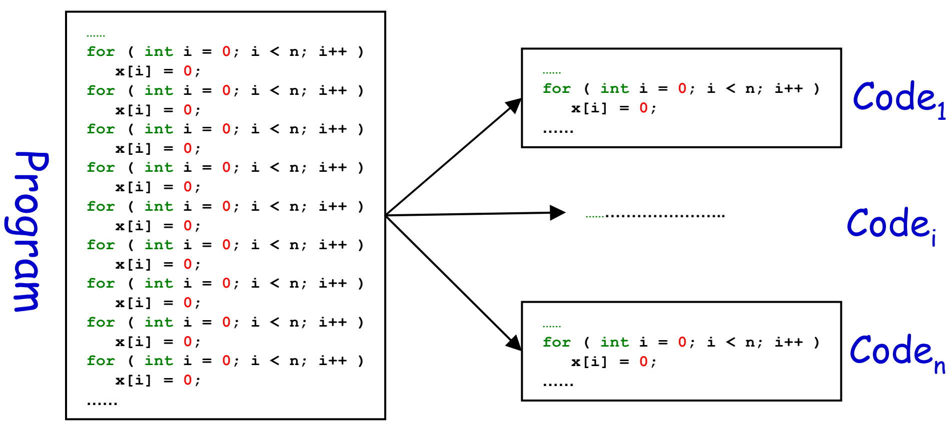
- Detect functionally equivalent code fragments



- Compare I/O behaviors directly
  - Run each piece of code with random inputs

Our Objectives — Challenges

- Detect functionally equivalent code fragments



- Large number of code fragments
- Unclear I/O interfaces

- Compare I/O behaviors directly
  - Run each piece of code with random inputs

- Huge number of code executions

Key 1: Semantic-Aware I/O Identification

- Identify input and output variables based on data flows in the code:
  - Variables used before defined are inputs
  - Variables defined but may not used are outputs

```
min = i;
j = i+1;
while (1) {
if(j >= LENGTH)
break;
if(data[j] < data[min])
min = j;
j++; }
if(min > i) {
tmp = data[min];
data[min] = data[i];
data[i] = tmp; }
```

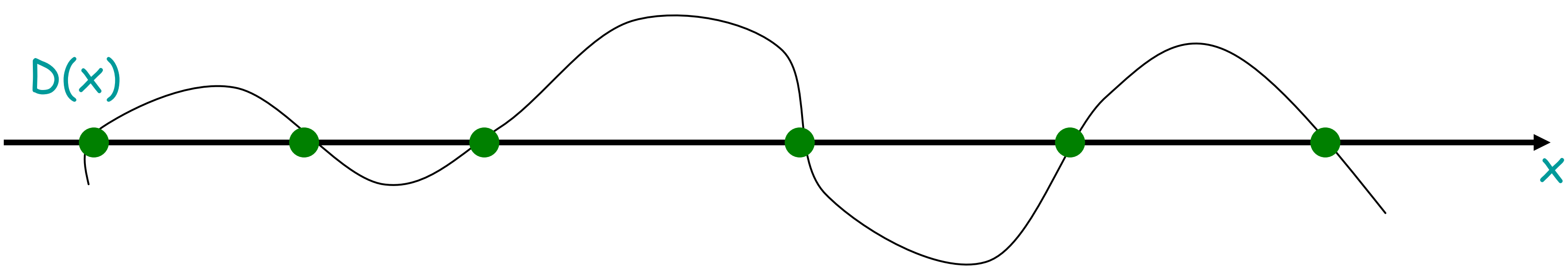
Input variables: `i` and `data`

Output variables: `data`

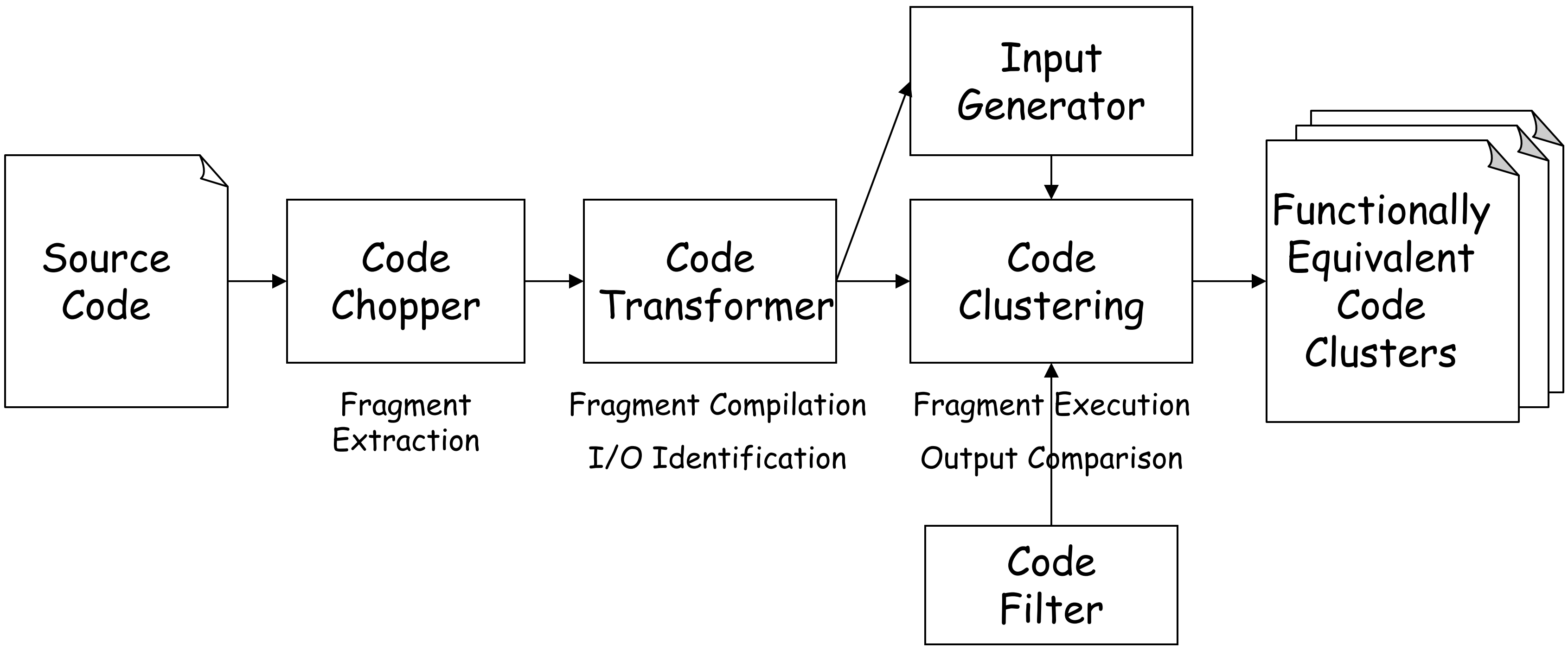
Key 2: Limit Number of Inputs

- Schwartz-Zippel lemma: polynomial identities can be tested with few random values
  - Let $D(x)$ be $p_1(x) - p_2(x)$
  - If $p_1(x) = p_2(x)$,

  - If $p_1(x) \neq p_2(x)$,
    - $D(x) = 0$ has at most finite number d of roots
    - Prob ($D(v) = 0$) is bounded by d, for any random value v from the domain of x.



EqMiner


Source Code
Code Chopper
Fragment Extraction
Code Transformer
Fragment Compilation
I/O Identification
Input Generator
Code Clustering
Fragment Execution
Output Comparison
Code Filter
Functionally Equivalent Code Clusters

Code Chopper

- Sliding windows of various sizes on serialized statements

```
min = i;

for(j=i+1; j<LENGTH; j++)
{

    if(data[j] < data[min])
        min = j;
}
if (min > i) {
    int tmp = data[min];
    data[min] = data[i];
    data[i] = tmp; }
```

```
min = i;
j = i+1;
while (1) {
if(j >= LENGTH)
break;
if(data[j] < data[min])
min = j;
j++; }
if(min > i) {
tmp = data[min];
data[min] = data[i];
data[i] = tmp; }
```

Code Transformer

- Declare undeclared variables, labels
- Define all used types
- Remove assembly code
- Replace goto, return statements
- Replace function calls
  - Replace each call with a random input variable
  - Ignore side effects, only consider return values
- Read inputs
- Dump outputs

Input Generation

- In order to share concrete input values among input variables for different code fragments, separate the generation into two phases:
  1. Construct bounded memory pools filled with random primary values and pointers. E.g.,

Primary value pool (bytes):	100	-78	......
Pointer value pool (0/1):	1	0	......

  2. Initialize each variable with values from the pools. E.g.,

```
struct { int x, y; } X;
```

Input variables: `X* x; int* y;`

```
x = malloc(sizeof(X));
x.x = 100; x.y = -78;
y = 0;
```

Code Clustering

- Eager partitioning of code fragments for a set of random inputs

f1, f2, f3, f4, f5, f6, f7, f8, f9, …, fi, …, fn

I_1 :

Code Clustering

- Eager partitioning of code fragments for a set of random inputs

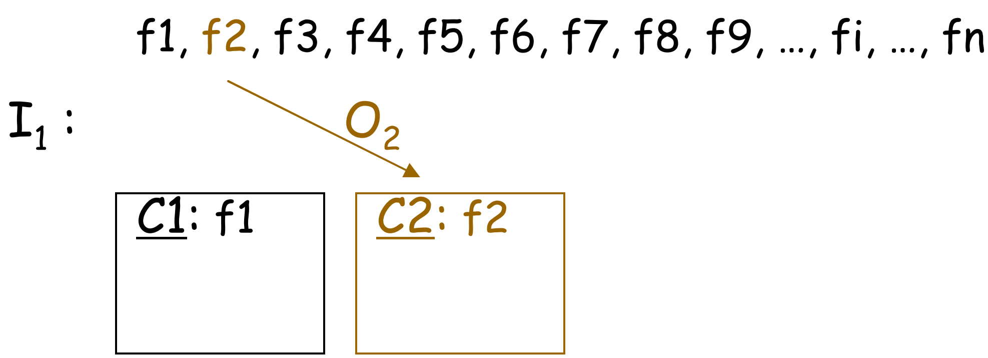
f1, f2, f3, f4, f5, f6, f7, f8, f9, …, fi, …, fn

I_1 : ↓ O_1

C1: f1

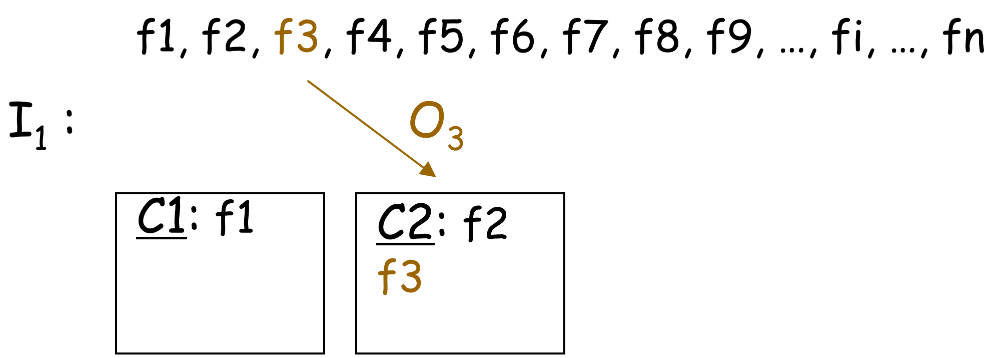
Code Clustering

- Eager partitioning of code fragments for a set of random inputs



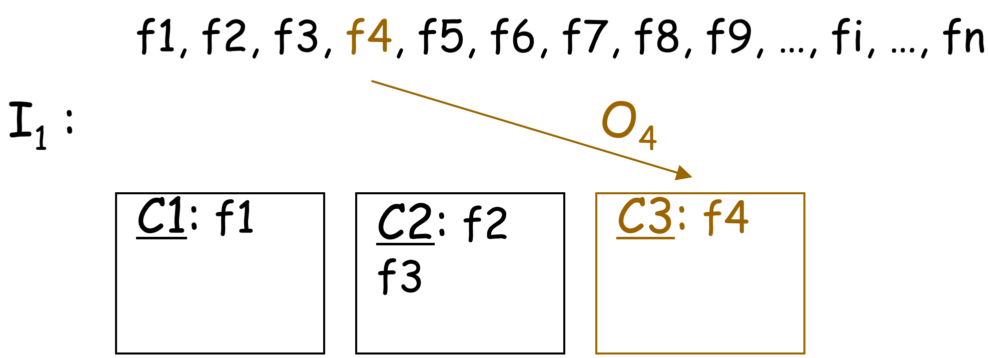
Code Clustering

- Eager partitioning of code fragments for a set of random inputs



Code Clustering

- Eager partitioning of code fragments for a set of random inputs



Code Clustering

- Eager partitioning of code fragments for a set of random inputs

f1, f2, f3, f4, f5, f6, f7, f8, f9, …, fi, …, fn

I_1 :

C1: f1 f5	C2: f2 f3, f6	C3: f4	C4: f7	……	Ck: fi …, fn

Code Clustering

- Eager partitioning of code fragments for a set of random inputs

f1, f2, f3, f4, f5, f6, f7, f8, f9, …, fi, …, fn

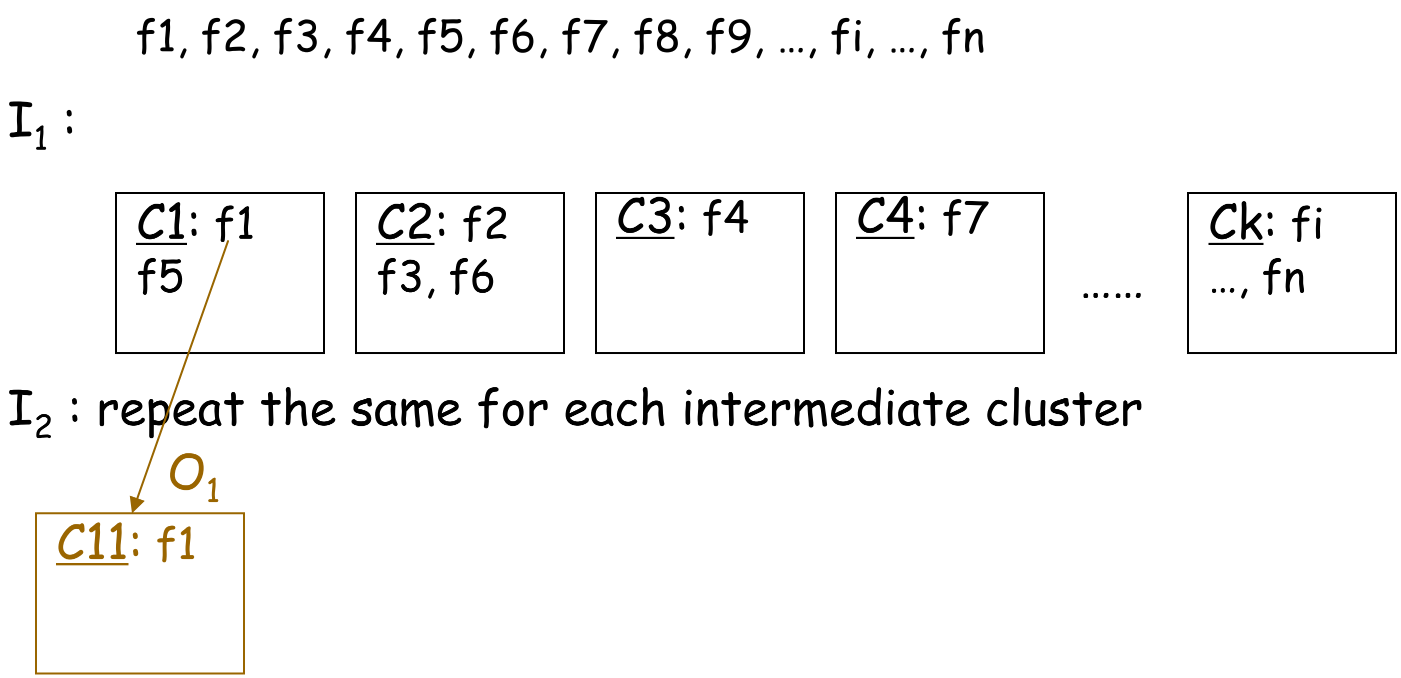
I_1 :

C1: f1 f5	C2: f2 f3, f6	C3: f4	C4: f7	……	Ck: fi …, fn

I_2 : repeat the same for each intermediate cluster

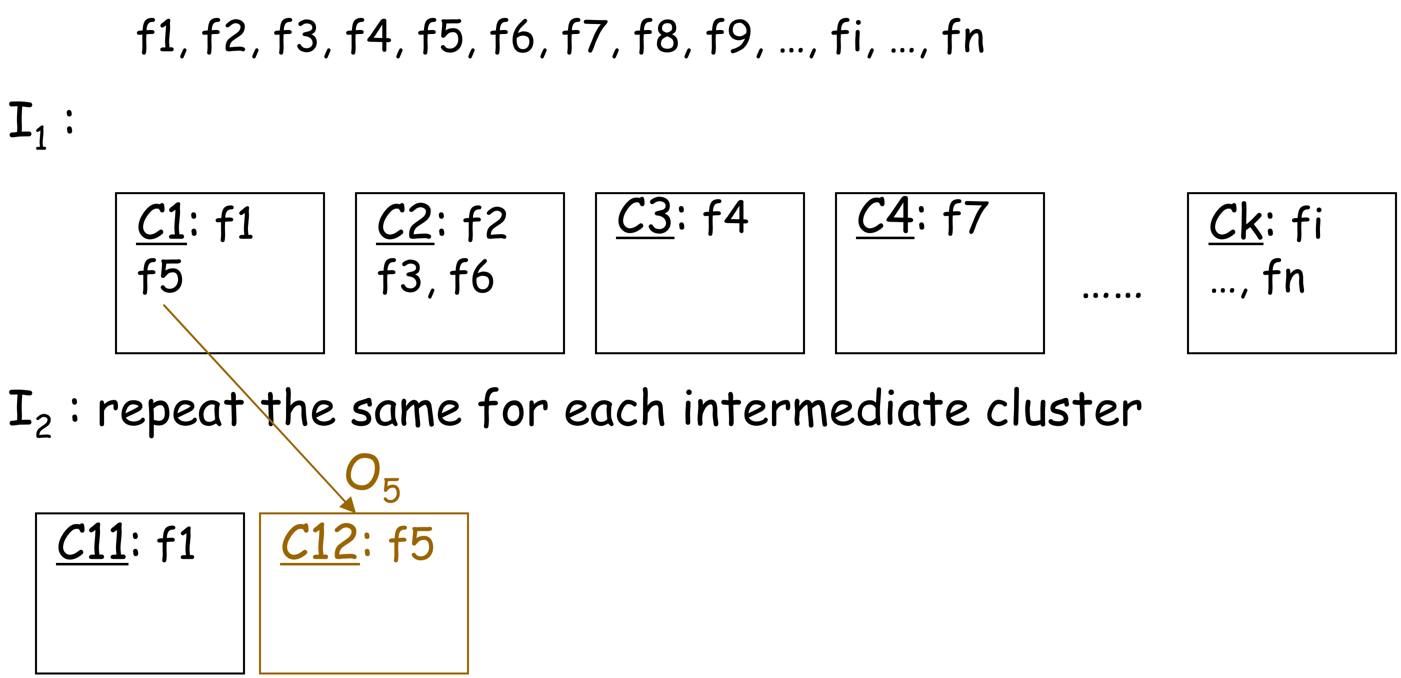
Code Clustering

- Eager partitioning of code fragments for a set of random inputs



Code Clustering

- Eager partitioning of code fragments for a set of random inputs



Code Clustering

- Eager partitioning of code fragments for a set of random inputs

f1, f2, f3, f4, f5, f6, f7, f8, f9, …, fi, …, fn

I_1 :

C1: f1 f5	C2: f2 f3, f6	C3: f4	C4: f7	……	Ck: fi …, fn

I_2 : repeat the same for each intermediate cluster

C11: f1	C12: f5	……	Ck1: fi fj	Ck2: fl …, fp	Ckx: fq …, fn

Code Clustering

- Eager partitioning of code fragments for a set of random inputs

f1, f2, f3, f4, f5, f6, f7, f8, f9, …, fi, …, fn

I_1 :

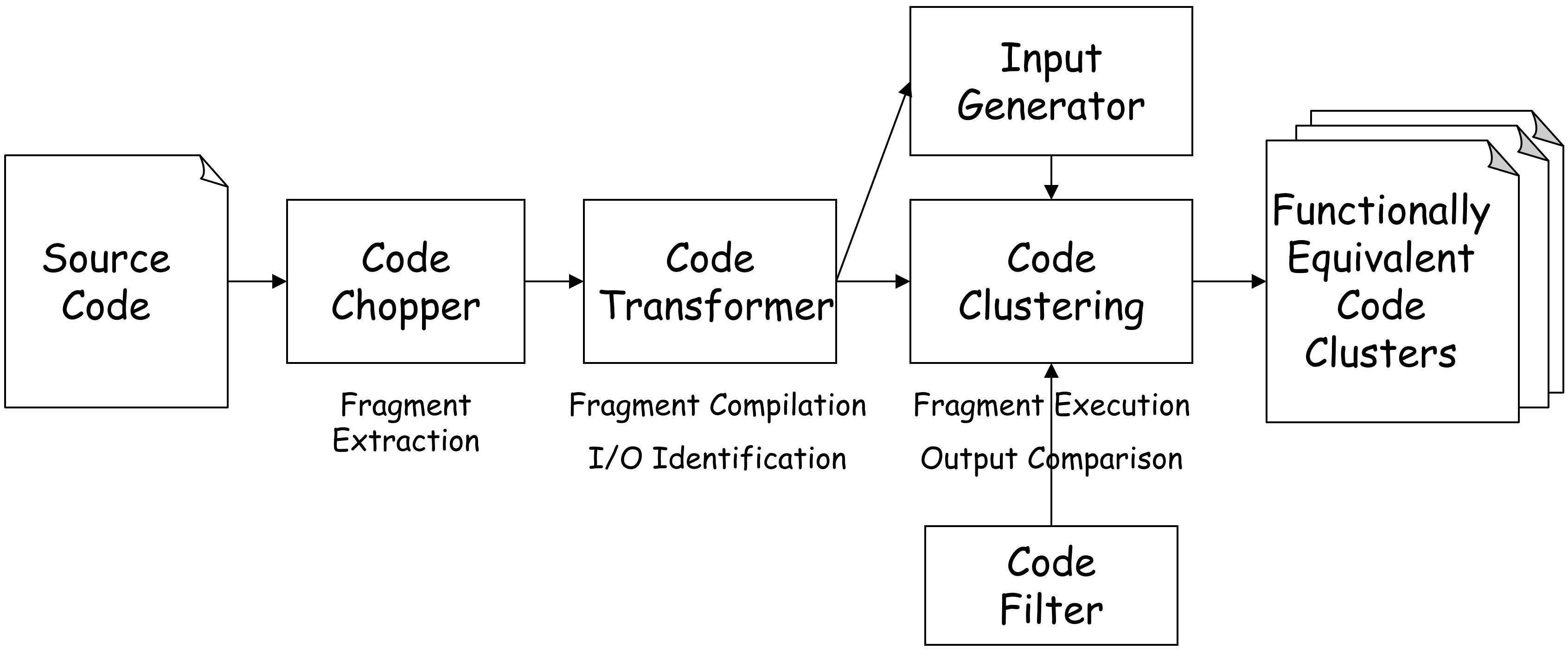
C1: f1 f5 | C2: f2 f3, f6 | C3: f4 | C4: f7 | …… | Ck: fi …, fn

I_2 : repeat the same for each intermediate cluster

C11: f1 | C12: f5 | …… | Ck1: fi fj | Ck2: fl …, fp | Ckx: fq …, fn

I_s : until only one code fragment is left for each cluster, or until a reasonable number s of inputs are used

EqMiner

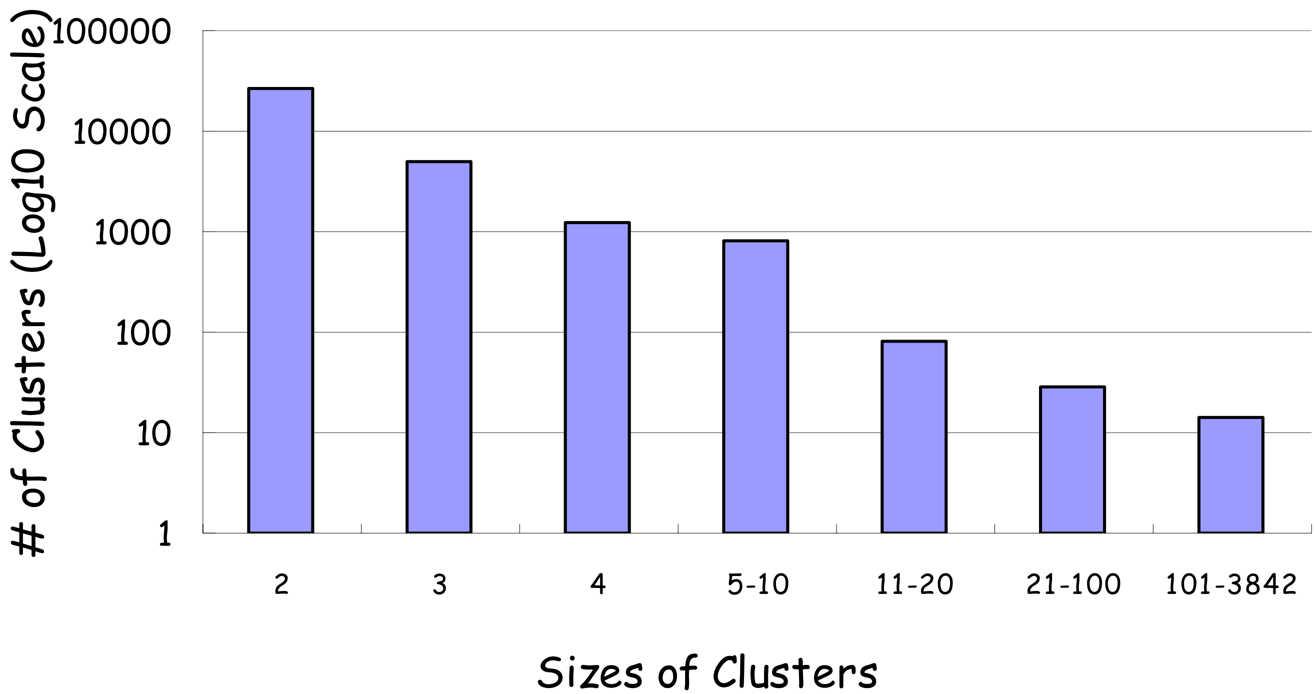

Input Generator
Source Code
Code Chopper
Code Transformer
Code Clustering
Functionally Equivalent Code Clusters
Fragment Extraction
Fragment Compilation
I/O Identification
Fragment Execution
Output Comparison
Code Filter

Results on Sorting Algorithms

- 5 sorting algorithms with both recursive and non-recursive versions
  - ~350 LoC
  - ~200 code fragments
- s = 10
  - 69 clone clusters reported
    - Most are portions of the algorithms
    - 4 non-recursive versions are in a same cluster

Results on the Linux Kernel

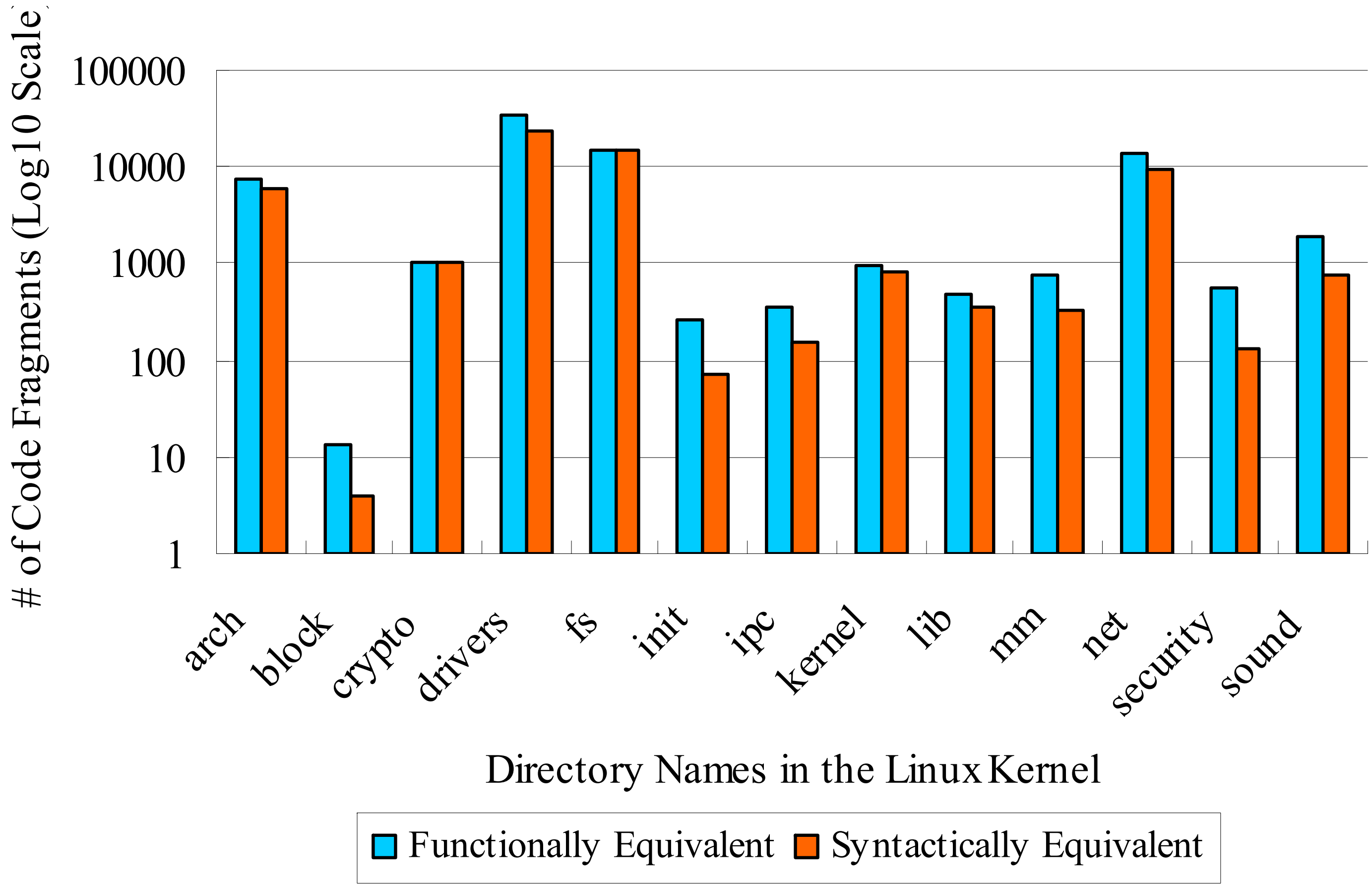
- s = 10
  - >800K code fragments were separated into 32K non-trivial clusters

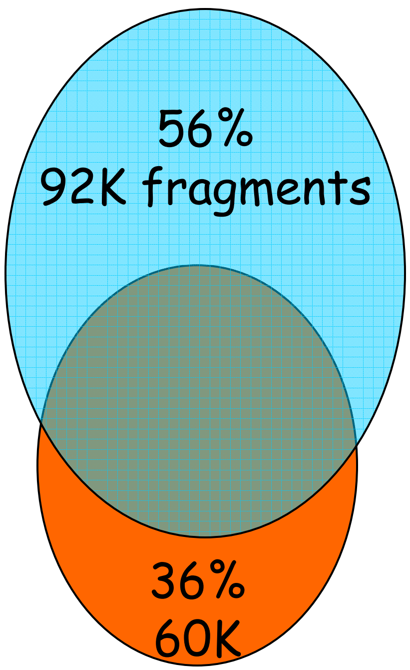


Results on the Linux Kernel

- s = 10
  - >800K code fragments were separated into 32K non-trivial clusters
- Additional 100 for 128 semi-randomly selected clusters
  - 3% of all of the code fragments became singletons
- 100 more tests
  - 0.5% additional

Differences from Syntactic Clones





Differences from Syntactic Clones

- False positives
  - Function calls
- Macro related + few outputs

```
if ( ALWAYS_FALSE ) {
   ......
} else
    output = input;
```

```
output = input;
```

- Lexical differences

```
output = input + 10;
```

```
output = input + 100;
```

```
output = 0;
if ( output < input ) {
   ...
   output = input + 1;
}
```

```
output = 0;
if ( output < input ) {
   ...
   output = output + 1;
}
```

Conclusion & Future Work

- First scalable detection of functionally equivalent code based on random testing
- Confirm the existence of many functional clones which complement syntactic clones
  - Enable further studies on functional clone patterns
  - Explore utilities of functional equivalent code

Thank you!

Questions?

jiangl@cs.ucdavis.edu