Automatic Mining of Functionally Equivalent Code Fragments via Random Testing

Lingxiao Jiang and Zhendong Su
Cloning in Software Development

How

New Software Product
Cloning in Software Development

Prior Knowledge

How

Specification
Documentation
Code Base
Test Suites
Bug Database

Search
Copy Paste
Modify Compose Reimplement

New Software Product
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

- Run each piece of code with random inputs

• Compare I/O behaviors directly
Our Objectives — Challenges

- Detect functionally equivalent code fragments
- Compare I/O behaviors directly
  - Run each piece of code with random inputs

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

```java
min = i;
j = i+1;
while (j) {
  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;
```
Key 2: Limit Number of Inputs

- **Schwartz-Zippel** lemma: polynomial identities can be tested with few random values
  - Let $D(x) = p_1(x) - p_2(x)$
  - If $p_1(x) = p_2(x)$,
    \[ D(x) \]
  - If $p_1(x) \neq p_2(x)$,
    - $D(x) = 0$ has at most finite number $d$ of roots
    - $\text{Prob}(D(v) = 0)$ is bounded by $d$, for any random value $v$ from the domain of $x$. 

![Diagram of polynomial functions and their differences]
EqMiner

Introduction

Functional Clones

EqMiner w/ Evaluation

Conclusion

Source Code

Code Chopper

Fragment Extraction

Code Transformer

Fragment Compilation

I/O Identification

Code Clustering

Fragment Execution

Output Comparison

Input Generator

Functionally Equivalent Code Clusters

Code Filter

CSSR

Center for Software and Systems Research
Code Chopper

- Sliding windows of various sizes on serialized statements

```java
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;
}
```
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.,

<table>
<thead>
<tr>
<th>Primary value pool (bytes):</th>
<th>100</th>
<th>-78</th>
<th>......</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pointer value pool (0/1):</td>
<td>1</td>
<td>0</td>
<td>......</td>
</tr>
</tbody>
</table>

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

```c
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

  \[ f_1, f_2, f_3, f_4, f_5, f_6, f_7, f_8, f_9, \ldots, f_i, \ldots, f_n \]

  \[ I_1 : \]
Code Clustering

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

\[ f_1, f_2, f_3, f_4, f_5, f_6, f_7, f_8, f_9, \ldots, f_i, \ldots, f_n \]

\[ I_1 : \quad \downarrow O_1 \]

\[ C_1 : f_1 \]
Code Clustering

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

\[ f_1, f_2, f_3, f_4, f_5, f_6, f_7, f_8, f_9, \ldots, f_i, \ldots, f_n \]

\[ I_1 \rightarrow O_2 \]

\[ C_1: f_1 \]

\[ C_2: f_2 \]
Code Clustering

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

\[ f_1, f_2, f_3, f_4, f_5, f_6, f_7, f_8, f_9, \ldots, f_i, \ldots, f_n \]

\[ I_1 : \]

\[ C_1 : f_1 \]

\[ C_2 : f_2 \]

\[ f_3 \]

\[ O_3 \]
**Code Clustering**

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

\[ I_1 : f_1, f_2, f_3, f_4, f_5, f_6, f_7, f_8, f_9, \ldots, f_i, \ldots, f_n \]

\[ O_4 \]

\[ C_1: f_1 \quad C_2: f_2 \quad f_3 \quad C_3: f_4 \]
Code Clustering

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

\[ f_1, f_2, f_3, f_4, f_5, f_6, f_7, f_8, f_9, \ldots, f_i, \ldots, f_n \]

\[ I_1 : \]

\begin{align*}
C_1: f_1 & \quad \quad f_5 \\
C_2: f_2 & \quad \quad f_3, f_6 \\
C_3: f_4 & \quad \quad C_4: f_7 & \quad \quad \vdots \\
& \quad \quad \ldots, f_n \\
\end{align*}
Code Clustering

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

\[ f_1, f_2, f_3, f_4, f_5, f_6, f_7, f_8, f_9, \ldots, f_i, \ldots, f_n \]

\[ I_1 : \]

\[ C_1 : f_1, f_5 \]

\[ C_2 : f_2, f_3, f_6 \]

\[ C_3 : f_4 \]

\[ C_4 : f_7 \]

\[ \ldots, C_k : f_i \]

\[ \ldots, f_n \]

\[ I_2 : \] repeat the same for each intermediate cluster
Code Clustering

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

\[ f_1, f_2, f_3, f_4, f_5, f_6, f_7, f_8, f_9, \ldots, f_i, \ldots, f_n \]

1. \( I_1 \):

   \[
   \begin{align*}
   C_1 & : f_1, f_5 \\
   C_2 & : f_2, f_3, f_6 \\
   C_3 & : f_4 \\
   C_4 & : f_7 \\
   \end{align*}
   \ldots
   \]

2. \( I_2 \): repeat the same for each intermediate cluster

   \[
   O_1 : C_{11} : f_1 \\
   \ldots, fn
   \]
Code Clustering

- Eager partitioning of code fragments for a set of random inputs
  
  $f_1, f_2, f_3, f_4, f_5, f_6, f_7, f_8, f_9, \ldots, f_i, \ldots, f_n$

  $I_1:$

  - $C_1: f_1 \quad C_2: f_2 \quad C_3: f_4 \quad C_4: f_7$
  - $C_k: f_i$

  $I_2:$ repeat the same for each intermediate cluster

  $O_5$

  - $C_{11}: f_1 \quad C_{12}: f_5$
**Code Clustering**

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

\[ f_1, f_2, f_3, f_4, f_5, f_6, f_7, f_8, f_9, \ldots, f_i, \ldots, f_n \]

\[ I_1 : \]

\[ C_1 : f_1, f_5 \]
\[ C_2 : f_2, f_3, f_6 \]
\[ C_3 : f_4 \]
\[ C_4 : f_7 \]
\[ \ldots, C_k : f_i, f_j, \ldots, f_n \]

\[ I_2 : \text{repeat the same for each intermediate cluster} \]

\[ C_{11} : f_1 \]
\[ C_{12} : f_5 \]
\[ \ldots \]
\[ C_{k1} : f_i, f_j \]
\[ C_{k2} : f_l, f_p \]
\[ \ldots, f_n \]
Code Clustering

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

\[ f_1, f_2, f_3, f_4, f_5, f_6, f_7, f_8, f_9, \ldots, f_i, \ldots, f_n \]

\( I_1 : \)

\[ \begin{array}{cccc}
C_1 : f_1 & f_5 \\
C_2 : f_2 & f_3, f_6 \\
C_3 : f_4 \\
C_4 : f_7 \\
\vdots \hspace{2cm} \vdots
\end{array} \]

\( C_k : f_i \ldots, f_n \)

\( I_2 : \) repeat the same for each intermediate cluster

\[ \begin{array}{cccc}
C_{11} : f_1 & C_{12} : f_5 \\
\vdots & \vdots & \vdots & \vdots \\
\vdots & \vdots & \vdots & \vdots \\
C_{k1} : f_i & C_{k2} : f_l & \ldots, f_p & C_{kx} : f_q \ldots, f_n \\
\end{array} \]

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

Introduction

Functional Clones

EqMiner w/ Evaluation

Conclusion

Source Code

Code Chopper

Fragment Extraction

Code Transformer

Fragment Compilation

I/O Identification

Code Clustering

Fragment Execution

Output Comparison

Input Generator

Functionally Equivalent Code Clusters

Code Filter
Results on Sorting Algorithms

- **5** sorting algorithms with both recursive and non-recursive versions
  - \( \sim 350 \) LoC
  - \( \sim 200 \) 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

![Bar chart showing the sizes of clusters](chart.png)
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

Directory Names in the Linux Kernel

- Functionally Equivalent
- Syntactically Equivalent

56% 92K fragments
36% 60K fragments
Differences from Syntactic Clones

- **False positives**
  - Function calls

- **Macro related + few outputs**
  
  ```java
  if ( ALWAYS_FALSE ) {
  
  } else {
  output = input;
  output = input;
  }
  
  output = input + 10;
  output = input + 100;
  
  output = 0;
  if ( output < input ) {
  ...output = output + 1;
  }
  ```

- **Lexical differences**
  
  ```java
  output = input + 10;
  output = input + 100;
  
  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