Detecting Refactorable Clones Using PDG and Program Slicing

By:
Ammar Hamid

Supervisor:
Vadim Zaytsev

Universiteit van Amsterdam — August 27, 2014

Research type

A replication study of Komondoor and Horwitz (2001), titled: “Using Slicing to Identify Duplication in Source Code”.

Research questions

- Can we find code clones of type-3?
  - non-contiguous
  - reordered
  - intertwined
- Are the found clones refactorable into new procedures?

Motivation: why replication study?

- To validate:
  - algorithm
  - experiment results
- To make it easier to:
  - revalidate our results
  - extend our program

Introduction

- Clone types classification
- Program Dependence Graph (PDG)
- Program slicing
- CodeSurfer
- Interaction with the original authors

Clone types – Roy 2007

- **Type-1:** whitespace and comments
- **Type-2:** identifiers, literals, types, layout and comments
- **Type-3:** statements can be changed, added or removed
- **Type-4:** same computation but different syntactic variants

Program Dependence Graph

- Captures the important dependences among program components
- Ignores arbitrary sequencing choices made by programmer
- Nodes represent program statements and predicates
- Edges represent data and control dependencies

Program slicing

- Decomposing program by analyzing data and control flow
- Query types:
    - backward slicing
    - forward slicing
- Backward slicing from node x means to find all the nodes that **influence** the value of node x
- Forward slicing from node y means to find all the nodes that **are influenced** by node y

CodeSurfer

- Code-understanding tool for C and C++ code
- A deep semantic analysis of a program
- Usage: interactively or programmatically

The original study

- PDG and program slicing for clone detection
- Represent each procedure using PDG representation
- Program slicing to filter out any statements that are irrelevant for clone detection

Algorithm description

- **Step 1:** find reachable procedures from the main program execution
- **Step 2:** find pair of *expression-typed* nodes with equivalent syntactic structure
- **Step 3:** find clones
- **Step 4:** group clones

Example of type-3 clone

Procedure A:

```
int foo(void) {
    int i = 1;

    bool z = true;

    int j = i + 1;
    int count;

    int unused = 10;

    for (count=0; count<10; count++)
    {
        j = j + 5;
        . . .
    }

    int k = i + j - 1;
     . . .
    return k;
}
```

Procedure B:

```
int bar(void) {
    int a = 1;

    int t = 10;

    int s;
    int b = a + 1;

    bool w = true;

    for (s=0; s<10; s++)
    {
        . . .
        b = b + 5;
    }

    . . .
    int c = a + b - 1;
    return c;
}
```

Interaction with the original authors

- Successfully contacted Raghavan Komondoor
- He was kind enough to share his code, including its documentation

The original implementation

- CodeSurfer version 1.8
- Scheme program (6123 LOC)
- C++ program (4380 LOC)

Our implementation: first attempt

- Run the original code sent by Raghavan
- Didn't work out because:

1. It used a very old CodeSurfer 1.8
2. Not supported anymore

Our implementation: second attempt

- Porting the original code to use CodeSurfer 2.3
- Didn't work out because:

1. version 1.8 and 2.3 use different (custom) Scheme interpreters (STk and STklos)
2. modularity issues

Our implementation: third attempt

- A new implementation from scratch
- Scheme (536 LOC)
- Ruby (161 LOC)

Changes to the original study

- Only reachable procedures
- **No** forward slicing

Example of why no forward slicing is needed

Procedure A:

```
fp3 = lookaheadset + tokensetsize;
for (i = lookaheads(state); i < k; i++) {
    fp2 = lookaheadset;
    fp1= LA + i * tokensetsize;
    while (fp2 < fp3)
        *fp2++ |= *fp1++; ++
}
```

Procedure B:

```
fp3 = base + tokensetsize;
while((j = *rp++) >= 0) {
    fp1 = base;
    fp2 = F + j * tokensetsize;
    while(fp1 < fp3)
        *fp1++ |= *fp2++;
}
```

This is a refactoring strategy

Analyzed programs

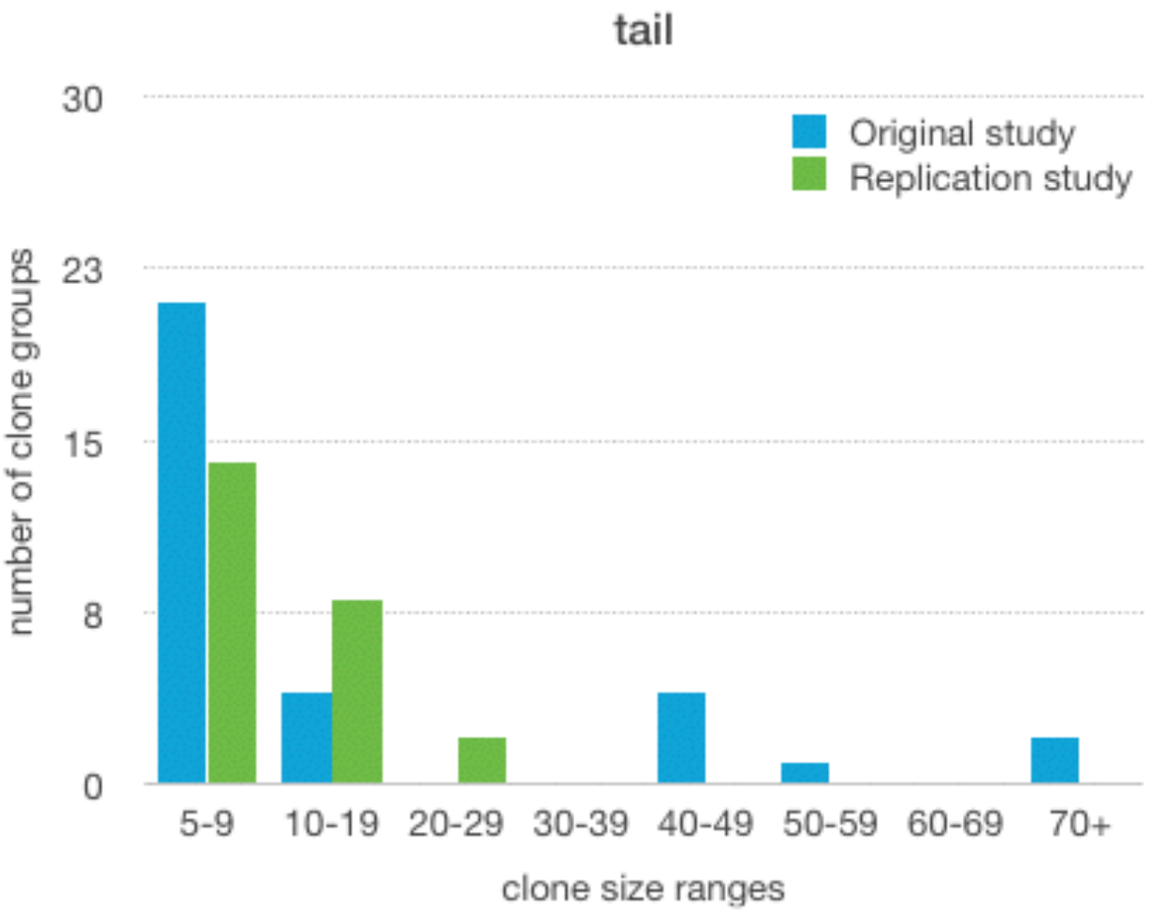
Study	Program	LOC	PDG nodes	Elapsed time		
				Scheme	C++	Ruby
Original	tail	1569	2580	40 sec.	3 sec.	NA
Replication	tail	1668	3052	5 sec.	NA	1 sec.
Original	sort	2445	5820	10 min.	7 sec.	NA
Replication	sort	2499	6891	30 sec.	NA	1 sec.
Original	bison	11540	28548	1 hour 33 min.	1 minute 5 sec.	NA
Replication	bison	10550	33820	2 hours 6 min.	NA	42 sec.

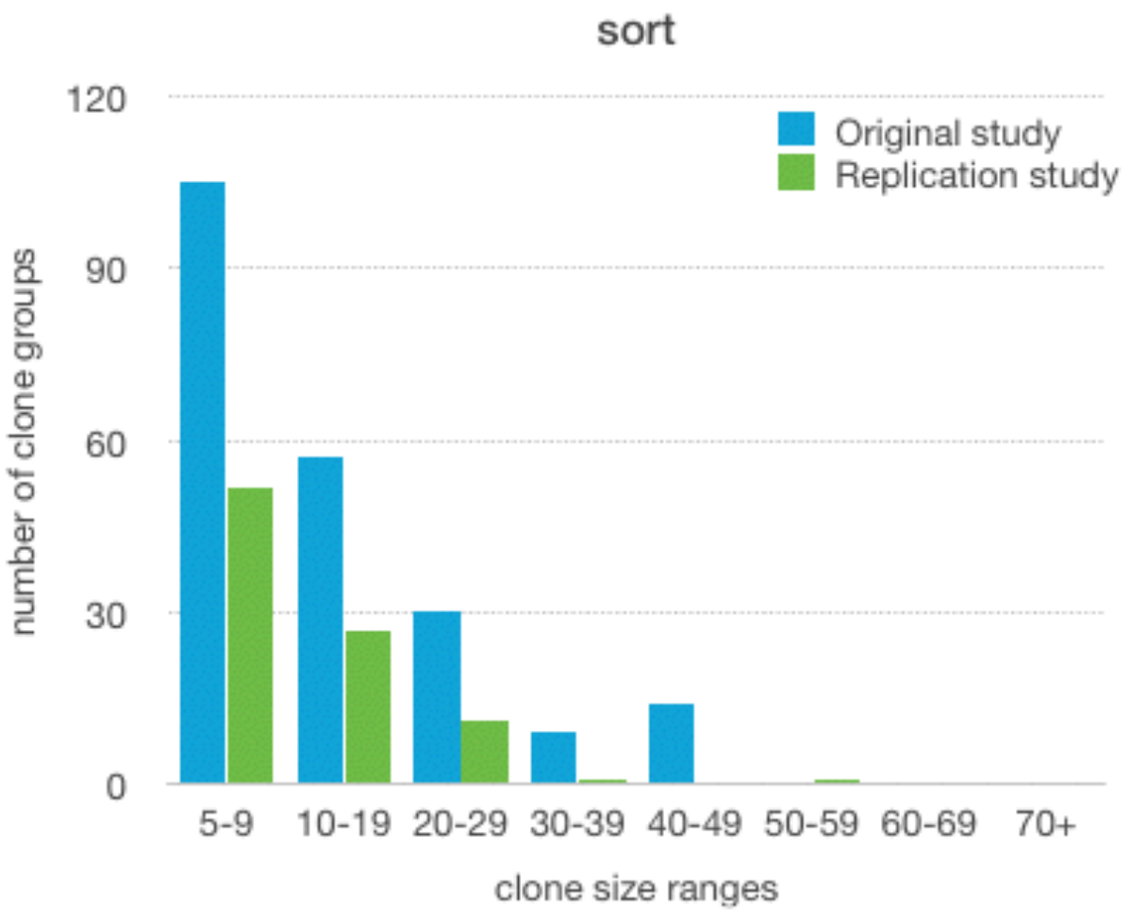
Table 4.1: Comparison on program size, number of nodes, implementation and elapsed time

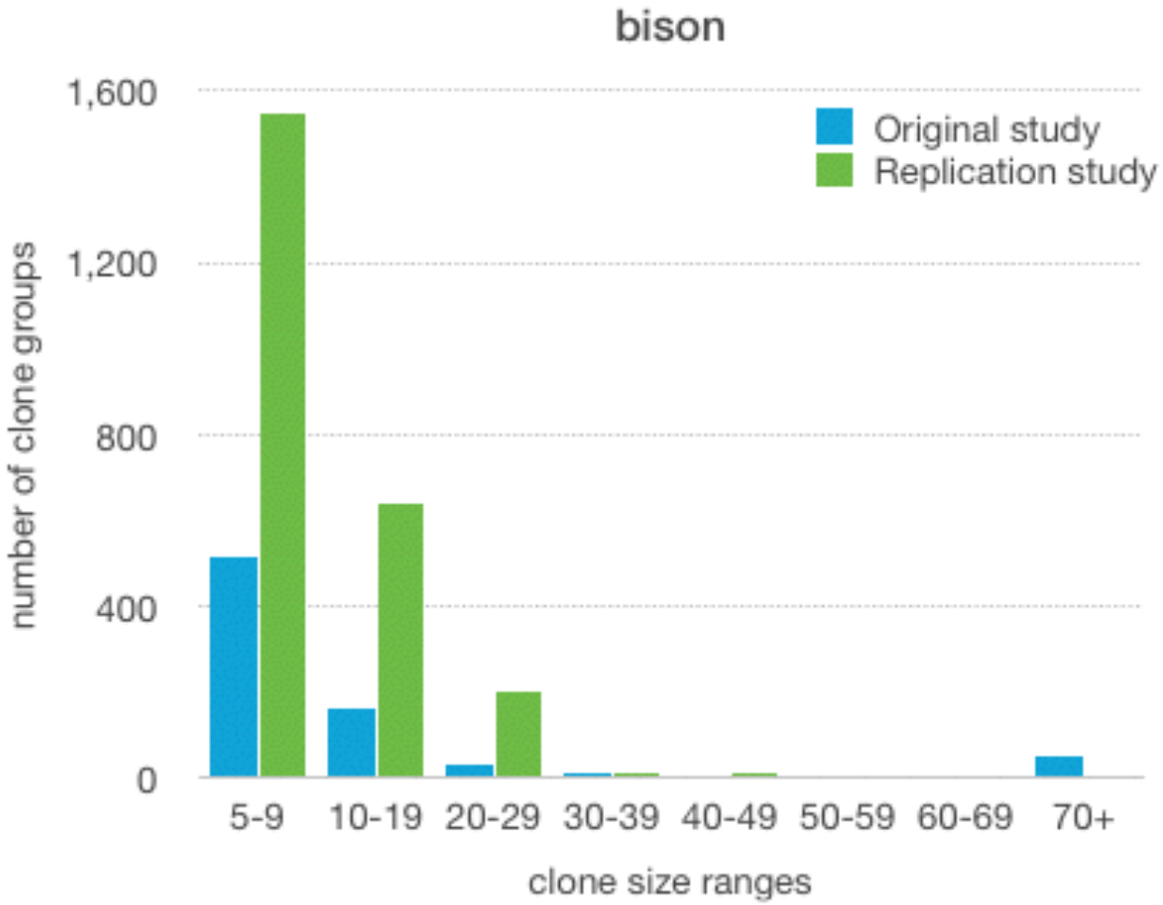
Detailed results

Study	Program	Number of nodes in a clone							
		5-9	10-19	20-29	30-39	40-49	50-59	60-69	70+
Original	tail	21	4	0	0	4	1	0	2
Replication	tail	14	8	2	0	0	0	0	0
Original	sort	105	57	30	9	14	0	0	0
Replication	sort	52	27	11	1	0	1	0	0
Original	bison	513	164	34	16	9	9	6	49
Replication	bison	1545	638	201	15	14	6	0	1

Table 4.2: Detailed comparison results between the original and replication study


tail
30
23
15
8
0
number of clone groups
Original study
Replication study
5-9
10-19
20-29
30-39
40-49
50-59
60-69
70+
clone size ranges


sort
120
90
60
30
0
number of clone groups
Original study
Replication study
5-9
10-19
20-29
30-39
40-49
50-59
60-69
70+
clone size ranges


bison
1,600
1,200
800
400
0
number of clone groups
Original study
Replication study
5-9
10-19
20-29
30-39
40-49
50-59
60-69
70+
clone size ranges

Hypotheses

- Altered algorithm [insignificant]
- Manual inspection [random for `bison`]
- Bison running time in the original study is suspiciously short
- Clone size are longer than many functions (group 70+)
- Clone size **vs** clone group on Figure 7 of the original (pp. 11)
- 10 KLOC is harder to test than 1 KLOC

Conclusion

- **Yes**, the [replicated] algorithm works
  - PDG & slicing are suitable for type-3 clone detection
  - CodeSurfer has improved over the years
- Quantitative differences
  - evidence of correctness
  - impossible to compare in detail [lack of original data]
- **Yes**, the clones are refactorable

Contributions

- GitHub: https://github.com/ammarhamid/clone-detection
  - code and intermediate results
  - makes it easier to revalidate, replicate, and extend
  - a tool to use for future work
- SATToSE 2014 in L'Aquila, Italy
  - presented extended abstract and early results
  - pre-proceedings: http://grammarware.github.io/sattose/SATToSE2014.pdf **[pp. 56–59]**
  - post-proceedings: [WIP @ **CEUR**]

Questions