View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Institutional Knowledge at Singapore Management University

Singapore Management University
Institutional Knowledge at Singapore Management University

Research Collection School Of Information Systems | School of Information Systems

5-2009

Automatic mining of functionally equivalent code fragments via random testing

Lingxiao JIANG
Singapore Management University, lxjiang@smu.edu.sg

Zhendong SU
University of California, Davis

DOI: https://doi.org/10.1145/1572272.1572283

Follow this and additional works at: https://ink.library.smu.edu.sg/sis_research

Part of the Software Engineering Commons

Citation

JIANG, Lingxiao and SU, Zhendong. Automatic mining of functionally equivalent code fragments via random testing. (2009). *ISSTA '09: Proceedings of the 2009 International Symposium on Software Testing and Analysis, Chicago, July 19-23, 2009*. 81-92. Research Collection School Of Information Systems.
Available at: https://ink.library.smu.edu.sg/sis_research/954

This Conference Proceeding Article is brought to you for free and open access by the School of Information Systems at Institutional Knowledge at Singapore Management University. It has been accepted for inclusion in Research Collection School Of Information Systems by an authorized administrator of Institutional Knowledge at Singapore Management University. For more information, please email libIR@smu.edu.sg.

ISSTA '09: Proceedings of the 2009 International Symposium on Software Testing and Analysis, Chicago, July 19-23, 2009, Pages 81-92.
http://doi.org/10.1145/1572272.1572283

Automatic Mining of Functionally Equivalent Code Fragments via Random Testing*

Lingxiao Jiang
University of California, Davis
jiangl@cs.ucdavis.edu

Zhendong Su
University of California, Davis
su@cs.ucdavis.edu

ABSTRACT

Similar code may exist in large software projects due to some common software engineering practices, such as copying and pasting code and n-version programming. Although previous work has studied syntactic equivalence and small-scale, coarse-grained program-level and function-level semantic equivalence, it is not known whether significant fine-grained, code-level semantic duplications exist. Detecting such semantic equivalence is also desirable because it can enable many applications such as code understanding, maintenance, and optimization.

In this paper, we introduce the first algorithm to automatically mine functionally equivalent code fragments of arbitrary size—down to an executable statement. Our notion of functional equivalence is based on input and output behavior. Inspired by Schwartz's randomized polynomial identity testing, we develop our core algorithm using automated random testing: (1) candidate code fragments are automatically extracted from the input program; and (2) random inputs are generated to partition the code fragments based on their output values on the generated inputs. We implemented the algorithm and conducted a large-scale empirical evaluation of it on the Linux kernel 2.6.24. Our results show that there exist many *functionally equivalent* code fragments that are *syntactically different* (*i.e.*, they are unlikely due to copying and pasting code). The algorithm also scales to million-line programs; it was able to analyze the Linux kernel with several days of parallel processing.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging — *code inspections and walk-throughs, diagnostics*; D.2.7 Distribution, Maintenance, and Enhancement — *Restructuring, reverse engineering, and reengineering*; D.2.m Miscellaneous — *Reusable software*

General Terms: Algorithms, Experimentation

Keywords: code clones, functional equivalence, random testing

1. INTRODUCTION

It is a common intuition that similar code, either syntactically or semantically, is ubiquitous due to some common software development practices, such as copying and pasting code and n-version programming. On one hand, the existence of similar code may indicate potentially high software development and maintenance cost, and a need to reduce the amount of similar code. On the other hand, it is difficult to avoid having similar code because of various resource constraints in development. The abundance of similar code in existing software provides opportunities for one to study its origins, characteristics, and evolution with the potential to improve many aspects of the software development process.

To characterize similar code, the first step is to find it. There have been many techniques for detecting similar code, especially *syntactically similar* code [1, 5, 18, 20]. Large-scale studies have shown that a project may contain more than 20% syntactically similar code and demonstrated applications of detecting similar code such as refactoring, plagiarism detection, and defect detection. However, few studies exist that target *semantically similar* code that may not be syntactically similar. In fact, no study has empirically validated the ubiquitous existence of semantically similar code although it is a common intuition.

In this paper, we propose the first scalable approach for identifying *functionally equivalent* code fragments, where functional equivalence is a particular case of semantic equivalence that concerns the input/output behavior of a piece of code. With such an approach, we are able to discover many functionally equivalent code fragments, covering more than 624K lines of code in the Linux kernel 2.6.24, justifying the common intuition. About 58% of the functionally equivalent code fragments are syntactically different, which shows the need for functionality-aware techniques in addition to syntactic approaches. We have also validated our results by sampling reported equivalent code fragments and running additional random tests on them. About 68% of the sampled code fragments remained in the *same* functional equivalence clusters, and more than 96% of the sampled code fragments remained in *some* clusters, showing probabilistically high accuracy.

Our approach is different from previous studies on finding semantically equivalent code or checking the semantic equivalence between two pieces of code. First, the definition of functional equivalence used in this paper considers only the equivalence of the final output of different code fragments given the same input, and does not consider the intermediate program states. One important practical benefit of this definition is that it focuses on externally observable behavior of a piece of code and is insensitive to code transformations or different implementations for the same behavior. Thus, it may admit more functionally equivalent code.

Second, inspired by Schwartz's randomized polynomial identity testing [33], we apply random testing on arbitrary pieces of code and detect those with the same I/O behavior. The Schwartz-Zippel lemma [33, 39] states that a few random tests are sufficient to decide, with high probability, whether two polynomials are equivalent. Although the lemma only holds for polynomials, we leverage it here for arbitrary code: if two pieces of code always produce the same outputs on a selected number of random inputs, we have high confidence that they are functionally *equivalent*; even if they

*This research was supported in part by NSF CAREER Grant No. 0546844, NSF CyberTrust Grant No. 0627749, NSF CCF Grant No. 0702622, and US Air Force under grant FA9550-07-1-0532. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

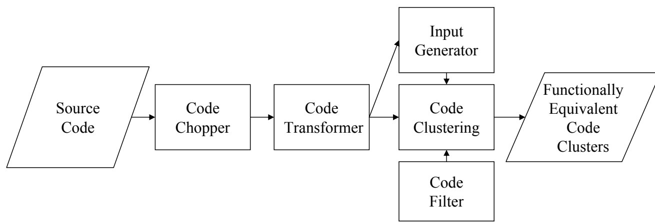


Figure 1: The work flow for mining functionally equivalent code.

may actually differ sometime, such as error-handling and boundary cases, they may still be considered functionally *similar* and provide opportunities for further studies.

Third, to the best of our knowledge, this paper presents the first large-scale study on the existence of functionally equivalent code in million-line software. Many unique optimizations in the implementation made our approach scalable.

In the rest of the paper, we first give an overview of our approach and discuss its algorithmic details (Section 2). Then, Section 3 presents the implementation of our approach, and Section 4 details our empirical evaluation of the approach on both a small sorting benchmark and the Linux kernel. Section 5 discusses some limitations of and future work for our approach. Finally, we present additional related work (Section 6) and conclude (Section 7).

2. ALGORITHM DESCRIPTION

This section presents details of our approach for detecting functional equivalence. We start with an overview of the approach.

2.1 High-level Overview

The main components of our approach are illustrated in Figure 1.

Code chopper. Since we consider functionally equivalent code of various sizes, instead of whole-program or whole-function, we use a *code chopper* to extract code fragments from a program as candidates for functionally equivalent ones. It takes a function definition and parses it into a sequence of statements; then it extracts all possible *consecutive subsequences* from the statement sequence, and each of the subsequences is considered a candidate for functional equivalence. We here illustrate what code fragments may be extracted for the following sample code excerpt from a selection sort algorithm: where the code to the right is the normalized sequence of statements of the code to the left.

```
1  min = i;
2
3  for(j=i+1; j<LENGTH; j++)
4  {
5
6     if(data[j] < data[min])
7        min = j;
8  }
9  if (min > i) {
10    int tmp = data[min];
11    data[min] = data[i];
12    data[i] = tmp; }
```

```
1  min = i;
2  j = i+1;
3  while (1) {
4  if(j >= LENGTH)
5  break;
6  if(data[j] < data[min])
7  min = j;
8  j++; }
9  if(min > i) {
10 tmp = data[min];
11 data[min] = data[i];
12 data[i] = tmp; }
```

When we require the minimum number of *primary statements*[1], contained in a code fragment to be 10, the code chopper will generate three code fragments if the boundaries of statements are respected: the first contains lines 1–12, the second contains lines 2–12, and the third contains lines 3–12. If the boundaries are not respected, the code chopper may generate six code fragments.

Code transformer. Because we define equivalence in terms of I/O behavior, we need to identify the inputs and outputs for each code fragment. This task is straightforward if we only consider coarser-grained code: for a whole program, we could directly use the inputs and outputs of the program; for a whole function, we could use the arguments of the functions as its inputs and its side effects and return values as its outputs. It is not obvious for a code fragment of arbitrary sizes. The *code transformer* exploits the heuristic that inputs should be the variables that are used but not defined in the code, and outputs should be the variables that are defined but not used by the code. Thus, specialized data-flow analyses can be utilized to identify such input and output variables. For example, for the code fragment containing lines 1–12 from the above example, the variables `i` and `data` are used before their first definitions and thus they are identified as the input variables for this code fragment; the variable `data` is the only variable along the control flow paths of the code fragment that is not used after its last definition and thus it is identified as the only output variable.

Since our approach requires executions of all the code fragments, each of the code fragment should be made compilable and executable. For C programs, this can involve many details such as defining all types used in the code, defining functions that are called but not defined in the code, and declaring all variables used in the code. The code transformer also takes care of these details.

Input generator. Since the executions of each code fragment require random inputs, the *input generator* component takes a code fragment and its input variables and generates random values for the input variables. Currently, it does not take the validity of randomly generated inputs *w.r.t.* a code fragment into account since we assume functionally equivalent code fragments should exhibit the same behavior on even invalid inputs. Section 2.5 has more details about the way our input generator works.

Code clustering. The *code clustering* component takes a set of code fragments that have been compiled and random inputs generated by the input generator, executes each code fragment with the same random inputs, and separates two code fragments into two different code clusters whenever the outputs of the two code fragments differ on a same input. In such a way, all the code fragments will be dispatched into a set of code clusters, each of which may be considered a functionally equivalent class, assuming enough executions are performed for the code fragments. Section 2.6 has more details on how the outputs are compared against each other and how the clustering process works.

Code filter. Since the code chopper may extract many code fragments that overlap with each other in terms of the locations of the original source code they correspond to, it may not be interesting to consider them functionally equivalent if two code fragments overlap too much. Thus, a *code filter* can be placed both before and after code clustering to reduce both unnecessary executions of code fragments and false positives.

2.2 Equivalence Definition

We first introduce our definition for the aforementioned functional equivalence. We denote a piece of code as C and its set of *input variables* as $\mathcal{I}$. We also use I to represent a *sequence* of concrete values (also called *an input*) that can be used to instantiate $\mathcal{I}$ and execute C. Similarly, we use $\mathcal{O}$ to denote the set of *output variables* of C, and use O to represent a *set* of concrete values (also called *an output*) that is a concrete instantiation of $\mathcal{O}$. Then, $C(I) = O$ means that the execution of C with the input I generates the output O. We also use $\mathbb{I}$ and $\mathbb{O}$ to represent the sets of all possible inputs and outputs respectively.

Definition 2.1 (Functional Equivalence) *Two code fragments* C_1 *and* C_2 *are functionally equivalent if there exist two permutations* $p_1, p_2 : \mathbb{I} \rightarrow \mathbb{I}$, *such that* $\forall I \in \mathbb{I}$, $C_1(p_1(I)) = C_2(p_2(I))$, *where "$=$" is the standard set equivalence operator.*

The definition has several interesting aspects:

[1] Informally, we consider every expression statement, `if`, `switch`, loop, `return`, `break`, `goto`, and `continue` statements primary. The setting for different applications may be changed.

- Considering the fact that the ordering of input variables for two functionally equivalent code fragments should not matter, the definition allows different permutations of an input for C_1 and C_2. For example, one can see that `x1` in the following code functions the same as `y2` and `y1` functions the same as `x2`, but they appear in different orderings in the headers of `foo` and `bar`. Then, for any given input $I = \langle i_1, i_2 \rangle$, we need to instantiate `<x1,y1>` as $\langle i_1, i_2 \rangle$, but `<x2,y2>` as $\langle i_2, i_1 \rangle$, in order for the two pieces of code to generate the same (in the sense of set equivalence) outputs. Adding the permutation functions in the definition is to allow such different orderings among functionally equivalent code.

```
foo(int x1, int y1) {            bar(int x2, int y2) {
   a1 = x1 + y1;                    a2 = y2 - x2;
   b1 = x1 - y1;                    b2 = y2 + x2;
}                                }
```

  On the other hand, considering that the ordering of input variables in any *individual* code fragment is fixed, the definition requires the *same* permutation functions for all inputs, *i.e.*, p_1 and p_2 should be fixed for all inputs for the same pair of functionally equivalent code fragments.

- An output of C_1 and that of C_2 are compared as sets, instead of sequences. This flexibility accommodates code fragments that perform the same computation but output their results in different orders. The need is also illustrated by the above code example: given an input $I = \langle i_1, i_2 \rangle$, `foo` outputs $\langle i_1 + i_2, i_1 - i_2 \rangle$ according to the sequential ordering of `a1` and `b1` and `bar` outputs $\langle i_1 - i_2, i_1 + i_2 \rangle$, thus it is necessary to compare the outputs as unordered sets so that `foo` and `bar` can be detected as functionally equivalent.

- Considering that different code fragments can perform the same computation with different numbers or types of input variables, the definition defines the behavior of a code fragment C *w.r.t.* an input I (a sequence of concrete primitive values), instead of the input variables $\mathcal{I}$ of C. For example, the following code can be considered functionally equivalent to the above `foo` and `bar`:

```
fun( struct {int x3, y3;} X ) {
   a3 = X.x3 + X.y3;
   b3 = X.x3 - X.y3;
}
```

  Also, despite the differences in their input variables, we can use any $I = \langle i_1, i_2 \rangle$ to instantiate the only input variable `X` in `fun` as `X={`i_1`,`i_2`}` (in the syntax of `C` language).

  Thus, although the definition requires the inputs used for different code fragments to be the same, it does not require them to have the same numbers or types of input variables. Similarly, an output of a code fragment is viewed as a set of concrete primitive values, instead of possibly ordered or complex values for output variables of different types.

- In addition, we assume all side effects of each code fragment can be captured by its output variables and each fragment interacts with its environment only through $\mathcal{I}$ and $\mathcal{O}$.

Section 2.6 and 3 will describe our strategies for realizing the definition for mining functionally equivalent code in practice.

2.3 Code Chopping

As mentioned in Section 2.1, code fragments are extracted from each function to be used for later steps. Given a sequence of n primary statements, there may be $\frac{n(n+1)}{2}$ consecutive subsequences of the statements. Since code fragments that across statement boundaries, such as the fragment containing Line 2–11 from the code snippet in Section 2.1, are syntactically invalid and may not be interesting units for functionality study, our code chopper thus avoids generating such subsequences. Also, we use a parameter *minStmtNum* to exclude code fragments that contain fewer than *minStmtNum* primary statements. An obvious benefit of these two options is that it helps reduce the number of candidate code fragments and relatively improves the scalability of our approach.

Algorithm 1 Code Fragment Generation

```
 1: function CODEGEN(F)
 2: Input: F: a function in a syntax tree
 3: Output: a set of code fragments C = {c_i}
 4:     C ⇐ ∅
 5:     S ⇐ (pre-order traversal of all statements in F)
 6:     for all statement s_i ∈ S, where i is the index of s_i in S do
 7:         c_i ⇐ ⟨⟩        /* empty list */
 8:         for all statement s_j ∈ S, where j ≥ i do
 9:             c_i ⇐ append(c_i, s_j)
10:             if s_j, s_i not in the same statement scope then
11:                 continue
12:             end if
13:             if j − i + 1 ≥ minStmtNum then
14:                 C ⇐ C ∪ {c_i}
15:                 vectorGen(c_i) /* for use in Section 4.3.2 */
16:             end if
17:         end for
18:     end for
19: end function
```

Algorithm 1 illustrates the mechanism of the code chopper. Given a syntax tree of a `C` function, it utilizes a pre-order traversal (S) of the primary statements in the function and a sliding window controlled by a starting point (s_i) and an ending point (s_j) on the statement sequence to generate code fragments that respect statement boundaries (Line 10) and *minStmtNum* (Line 13).

2.4 Code Transformation

The main tasks for the code transformer are to identify the input and output variables of each code fragment extracted by the code chopper and make it compilable and executable.

2.4.1 Input Variables

Since variables in `C` code are often required to be initialized (*i.e.*, defined) before their uses, a variable in a code fragment should get its value from the environment and thus be treated as an input variable if it is not defined in the code fragment before its first use. Hence, comes the following definition:

Definition 2.2 (Input Variables) *A variable v used in a code fragment c is an input variable for c if there is no definition for v before* some *use of v in c, where "before" or "after" is measured along the directions of any control flow path in c.*

Liveness analysis [28] for a function F can tell us which variables should be live at the entry point of F, *i.e.*, undefined before their first uses in F and thus the input variables for F. Similarly, we use a *local* version of liveness analysis for any code fragment c extracted from F to decide which variables are live at the entry point(s) of c and should be the input variables for c. The local liveness analysis is the same as standard backwards-may liveness analysis except that it propagates liveness information only on a subgraph of the control flow graph of F that corresponds to c.

Functions called in c are also live at the entry point(s) but handled differently from variables (*cf.* Section 2.4.4). Undefined labels in `goto` statements imply the target statements are not contained in the code fragment c and we can terminate the executions of c whenever they reach such `goto`s. Thus, we simply transform `goto`s with undefined labels to "`goto _dummy_label;`", and add, as the last statement of c, an empty statement "`_dummy_label: ;`".

2.4.2 Output Variables

Given an arbitrary code fragment c, it is non-trivial to decide which variables hold the data intended by the programmer to be externally observable (*i.e.*, part of its output). We use the following heuristics to make the decision:

- A definition d for a variable v in c should serve some purpose (*i.e.*, to be used somewhere in c or later): if v is used after d, v may not be needed any more since its value has served some purpose; if v is not used after d, it should be an output variable if we want d to be used somewhere later.
- `return` statements in c may indicate that the programmer wants the return value to be part of an output. Thus, we transform all `return` statements in c so that a specially-named variable is assigned the return value before each `return` and considered as an output variable for c.

Definition 2.3 (Output Variables) *A variable v in a code fragment c is an output variable for c if it is a specially-named variable for a* `return` *statement in c or there is no use of v after a definition of v on* some *control flow path in c.*

Reaching definition analysis [28] for a function F can tell us which definitions may reach the exit point of F and thus be the output for F. Similarly, we use a *local* version of reaching definition analysis for any code fragment c extracted from F to decide which variable definitions may reach the exit point(s) of c and should be the output variables for c. The local reaching definition analysis is the same as standard forwards-may reaching definition analysis except that it propagates reaching information only on a subgraph of the control flow graph of F that corresponds to c.

We can also strength Definition 2.3 by changing *some* control flow path to *all* control flow path, then the reaching definition analysis will be a forwards-must analysis, and only those variable definitions that must reach the exit point(s) of c will be included in the set of output variables. The alternative definition will obviously change the output of a code fragment, and may affect the results of mining functionally equivalent code. Section 4.3 will mention an example for this effect in our benchmark program.

2.4.3 Type Definitions

To make a code fragment c compilable, the first thing is to define every type referenced in c. One option is to traverse the code and identify which types are used in c and search in the source files for the definitions of the used types. Since a used type may refer to another type not explicitly used in c, we need to compute a closure of the referenced types. In this paper, we adopt the following simpler option which may include extra unused types: GCC preprocessor is invoked on the original source file f from which c is generated, then the preprocessed file naturally contains all types defined in any file included by f; Thus, as long as as the code chopper includes *all* the preprocessed type definitions with c, the problem is resolved, assuming the original file is compilable.

2.4.4 Function Calls

Each code fragment c may call other functions, some of which are library functions, some of which are functions defined somewhere else in the original source code. Strictly speaking, for two code fragments to be functionally equivalent, we should take the side-effects of the function calls into account and include all those function definitions with c.

In this paper, we take an alternative look at function calls: we view each callee as a random value generator and ignore its side-effects besides assignments through its return values (*i.e.*, the random values). Thus, n function calls in c are viewed in this paper as n extra input variables for c whose values will be generated randomly. Such a strategy helps to limit the execution time of each code fragment and improve the scalability of our approach. As future work, it may be possible to replace function calls with a learned mapping between inputs and outputs for the callees (*i.e.*, a summary of the behavior of the callees) to model them more accurately and modularly but still keep our approach scalable.

2.5 Input Generation

For each execution of a code fragment c, we need to instantiate its input variables with an input that may also be used for other code fragments. To make it easier to instantiate different types of input variables, we only separate input variables into two categories (non-pointers and pointers, and arrays are treated as pointers) and our input generator aims to generate generic values that may be used for all types. We thus encode each input as a sequence of concrete values, each of which may be assigned to a variable of primitive types, and a sequence of $p0$ and $p1$, each of which may indicate a null or non-null pointer.

For example, an input $I = \{48, -1, p1, p0\}$ is able to instantiate variables of different types in the following way: if a variable v is of type `float`, v will be instantiated with 48.0; if v is of type `char`, v will be the character '0' (ASCII code 48); if v is of type

```
struct node {int value; struct node * next;}
```

v will be a struct containing `value=48` and a non-null `next` pointing to another struct, allocated at run-time, containing `value=-1` and a null `next`. If an input contains fewer values than required by a variable or a set of variables, zeros are used. For example, if v is of the following type:

```
struct three {int a; char b; float c;}
```

v will be a struct containing `a=48; b=EOF(-1); c=0.0`, and the $p1$ and $p0$ are not used in this case.

With such an encoding scheme, generating random inputs and instantiating input variables can be separated into two phases. It helps the input generator to generate random inputs independently from any code fragment.

On the other hand, considering certain code specific properties may help the input generator to generate inputs more effectively and help reduce invalid code executions in the following code clustering step. In particular, we consider (1) the probability for generating $p0$ versus $p1$ and (2) the number of generated values in an input that may suit the need of a code fragment the best.

For (1), we consider limiting the probability of generating non-null pointers to avoid generating deeply linked data structure, such as trees, and help limit code execution time. We use exponential decay on the probability of generating $p1$: the more pointer values are added in an input, the more unlikely for $p1$ to occur, *i.e.*, when generating the first pointer value for an input, the probability of generating $p1$ is $\frac{1}{2}$; when generating the second pointer value, the probability of generating $p1$ will be $\frac{1}{2^2}$; and so on.

For (2), we consider generate enough concrete values in an input for input variable instantiation, while generating as few values as possible in order to limit the number of possible permutations of an input that may be required to check functional equivalence as defined in Definition 2.1. We thus statically estimate the possible number of concrete primitive values needed by a code fragment by assuming the "first-level" pointers are non-null and counting the needed values (the counters are initialized to 0):

- If a variable is of a primitive type, the counter will be increased by one.
- If a variable is a struct, the counter will be increased by the number of concrete values needed by all the *non-pointer fields* in the struct. This rule may be recursively applied.
- If a variable is a pointer, the counter will be increased by the number of concrete values needed by a variable of the pointed type, which is then recursively counted.

Algorithm 2 Code Execution and Clustering

```
1: function CODEEXE(𝕀, C)
2: Input: 𝕀: a finite set of inputs
3: Input: C: a finite set of code fragments
4: Output: a set of code clusters ℂ = {C_i}
5:    ℂ ⇐ ∅
6:    for all I ∈ 𝕀 do
7:       for all c ∈ C do
8:          𝕆 ⇐ ∅
9:          for all permutation p of I do
10:            𝕆 ⇐ 𝕆 ⋃ c(p(I)) /* code execution */
11:         end for
12:         for all C_i ∈ ℂ do /* code clustering */
13:            /* 𝕆_i is the representative outputs for C_i */
14:            if 𝕆 ⋂ 𝕆_i ≠ ∅ then
15:               C_i ⇐ C_i ∪ c
16:               break
17:            end if
18:         end for
19:         if ∀C_i ∈ ℂ, c ∉ C_i then
20:            C_{|ℂ|+1} ⇐ {c}
21:            𝕆_{|ℂ|+1} ⇐ 𝕆 /*record representative outputs*/
22:            ℂ ⇐ ℂ ⋃ C_{|ℂ|+1}
23:         end if
24:      end for
25:   end for
26: end function
```

The number of needed pointer values is estimated similarly:

- If a variable is of a primitive type, the counter will be kept the same.
- If a variable is a struct, the counter will be increased by the sum of the number of *pointer fields* and the number of pointer values needed by other *non-pointer fields* in the struct.
- If a variable is a pointer, the counter will be increased by one plus the number of pointer values needed by a variable of the pointed type, which is then recursively counted using these three rules. Note that pointers pointed to by a pointer are recursively counted, but pointer fields in a struct are not. This is what we mean by "first-level" pointers.

Then, the number of values needed in an input for a set of code fragments is determined by the minimum among the estimations for all code fragments. Using such a minimum helps avoid redundant values in inputs and reduce the input permutations and code executions that may be required to check functional equivalence among all the code fragments. Section 3 will also introduce a code fragment grouping strategy that may help to accommodate code fragments that require differently sized inputs.

2.6 Code Execution and Clustering

The goal of the code clustering component is to execute every code fragment generated and transformed in previous steps and separate them into functionally equivalent code clusters. Algorithm 2 illustrates the code execution and clustering process.

Algorithm 2 uses what we call *representative-based partitioning* strategy to make the code execution and clustering process incremental and scalable. Notice that any difference between the outputs of two code fragments should cause the two fragments to be separated into two clusters, and given the output set $\mathbb{O}$ of a code fragment c on a given input I and an existing cluster C_i, we only need to compare $\mathbb{O}$ with the output set $\mathbb{O}_i$ of the representative code fragment in C_i to decide whether c can be put into C_i, avoiding quadratic number of comparisons (Line 13–17). To make the algorithm incremental, we design it in such a way that it does not execute one code fragment on all generated inputs; instead, it tries to execute all code fragment (Line 7) on one input first and partition them into smaller sets (also called code clusters). Thus, the whole set can be gradually partitioned into functionally equivalent clusters with more and more inputs (Line 6). Also, the outputs of the representative code fragment (the first code fragment put into the cluster) are kept (Line 19–23) for comparison with coming code fragments during the incremental partitioning.

A difficulty with the incremental scheme is to find *one* permutation of *all* inputs that satisfies the requirements of Definition 2.1. We observe that it is unlikely for two functionally different code fragments to produce a same output even when *different* permutations for different inputs are allowed. We thus relax Definition 2.1 in Algorithm 2 to reflect the observation and look for a permutation for *each* input independently (Line 9–18). Section 3 also presents a strategy to reduce the complexity introduced by $n!$ permutations.

During output comparison, we use concrete values of output variables except for pointers for which we use $p0$ or $p1$, depending on whether the pointer is null or non-null, and the values pointed to by the pointer (recursions may occur if the pointer is multi-level).

Note that we do not consider input validity: randomly generated inputs may not satisfy implicit invariants required by each code fragment, and thus an execution of a fragment may not generate any output due to problems such as segmentation faults and infinite loops. We treat the outputs of all failed executions as a same special value, and compare the value in the same way as other outputs.

It is also worth mentioning that the executions of different code fragments for the same input can be easily parallelized, and so can the execution and clustering for each code cluster. Thus, the algorithm can be implemented as a parallel program and its degree of parallelism increases as it makes progress on code clustering.

3. IMPLEMENTATION

We have implemented our approach as a prototype, called EQMINER. This section discusses our implementation strategies for EQMINER.

The code chopper, the code transformer, and the input generator are implemented based on CIL [27]. The code clustering component is implemented as Python and Bash scripts.

Data storage. All the code fragments and their inputs and outputs are stored in plain text for convenient inspection. A significant disadvantage is that it may take a large amount of disk space when the number of code fragments is large even if each text file is very small. Also, since a file system often limits the number of subdirectories and files in a directory, we added a layer of file management in the code chopper and the code clustering to split or merge directories as needed. A future improvement will be to store compressed files in database system to avoid slow file operations.

Code compilation. Although it is easy and convenient to make each code fragment independent from each other, it can waste a lot of disk space and take much longer time to compile if many code fragments include common contents, such as the required type definitions. When millions of code fragments are possible, it is worthwhile to extract the common contents from code fragments and compile the common contents just once as shared libraries, and just link the share libraries with much smaller code fragments. Such an optimization was justified in our evaluation on the Linux kernel.

Input generation. The main complexity in Algorithm 2 is to use all possible $n!$ permutations of an input containing n values to execute each code fragment c. We argue that the exponential number of input permutations is largely unnecessary, based on these assumptions: (1) Random orderings of input variables mostly occur when computations on the variables are (conceptually) associative, such as addition and sorting. For such associative computations,

different input permutations should lead to the same output. (2) Most computations in the code fragments are not associative, and when they are not, programmers are more likely to follow certain customs (such as the ordering of involved operands or the flow of computation) to order the input variables, instead of randomly.

In EQMINER, we impose an empirical limit $5!$ on the number of permutations allowed for each input (based on the numbers of input variables from the Linux kernel, *cf.* Section 4.3) Also, we no longer perform regular permutations on an input since it is better to randomly select the limited permutations from all possible permutations. For this purpose, we use *random shuffling* of an input to implement the Line 9 in Algorithm 2 as:

for all (**upto** $5!$) a random shuffle p of I **do**

Also, to allow converting a randomly generated concrete value into different primitive types in C, we limited the range of the value to $[-127, 128]$ so that it can be casted into many types, *e.g.*, `char`, `short int`, `unsigned int`, `float`, etc.

Parallelization. We also observe that certain properties of each code fragment, such as the numbers of input and output variables, may provide opportunities for higher degrees of parallelism. The intuition is that useful functionally equivalent code is likely to execute on similar types of inputs and generate similar types of outputs. For example, different sorting algorithms often take the same array type as their input and output the same sorted array type. Code fragments with significantly different amount input and output variables are much less likely to be functionally equivalent.

Given a large set of code fragments, we first separate them into different groups according to the number of input and output variables they have—the fragments in a same group have the same numbers of input and output variables, then invoke Algorithm 2 on every group in parallel. Alternatively, to prevent missing certain functionally equivalent code, such as those mentioned in Section 2.2, we can group the code fragments based on their estimated numbers of needed concrete values. Besides increased degree of parallelism, another benefit of this grouping strategy is that we can generate inputs containing different numbers of concrete values for different groups so that groups with more input variables can have more values for increased testing accuracy.

4. EMPIRICAL EVALUATION

This section discusses our empirical experience with EQMINER on a benchmark program and the Linux kernel 2.6.24. The evaluations were carried out on a Fedora Core 5 system with a Xeon 3GHz CPU and 16GiB of memory, and a ROCKS computer cluster system with the SGE roll and varying numbers of hosts with an Opteron 2.6GHz CPU and 4GiB of memory.

4.1 Subject Programs

Sorting benchmark. We first used a benchmark program which contains several implementations of different sorting algorithms, including bubble sort, selection sort, recursive and non-recursive quick sort, recursive and non-recursive merge sort, and heap sort, to evaluate the effectiveness and accuracy of EQMINER.

Since the input generator in the prototype does not handle arrays, we wrote the benchmark in a way that the sorting algorithms accept a single struct containing a fixed number (7) of integers as their input variable, but they internally cast the struct to an array before actual sorting. Also, the coexistence of recursive and non-recursive versions was used to evaluate the effects of the way EQMINER handles function calls (*cf.* Section 2.4.4).

There are about 350 lines of code in the program and 200 code fragments were generated when the *minStmtNum* was set to 10.

Linux Kernel 2.6.24. We used the Linux kernel as it is a large project with a relatively long history of development and a large number of participating developers, thus we can (1) evaluate the existence of functionally equivalent code in a popular software and (2) test the scalability of our approach.

The Linux kernel 2.6.24 contains $9,730$ C files of more than 6.2 million lines of code. Since our code chopper requires compilable code to obtain abstract syntax trees and control flow graphs, we only consider a subset of the kernel that is compilable on our Fedora Core 5 system. Specifically, we used the default setting when configuring the kernel before compilation, and saved the intermediate files (preprocessed C files with the suffix `.i`) for chopping.

We obtained $4,750$ preprocessed C files which correspond to about 2.8 million lines of code in the original source code. Each such file is a compilable unit and can be processed by the code chopper independently from other files. The total number of lines of code contained in the preprocessed files is not interesting due to the fact that these preprocessed files contain a lot of commonly used type definitions and function declarations and definitions. On the other hand, it is interesting to know the numbers of functions and statements contained in those functions since they directly affect the number of code fragments generated by the code chopper.

Among the $4,750$ preprocessed files, CIL can successfully parse and construct ASTs and CFGs for $3,748$ files. There are $41,091$ functions with unique names in the $3,748$ preprocessed files, about $\frac{1}{3}$ of the total number of functions (more than 136K) in the Linux kernel [14]. If duplicated functions or functions with the same names in different files are also counted, the number would be more than 67K. We used these 67K functions for our following study and used the containing file and function name and line numbers as the unique identifier for each code fragment.

We calculated the numbers of primary statements contained in each function to estimate the total number of generated code fragments. Data showed that more than 26K functions contain fewer than 10 primary statement, while there are several functions contain more than $1,000$ statements. Since we set the *minStmtNum* to 10, our code chopper ignored the code fragments for those small functions. Without respecting the statement boundaries, the code chopper generated more than 20 million code fragments since more than ten functions contain thousands of statements. The number was reduced to about 6.5M when we respect boundaries.

4.2 Code Execution

An important decision we need to make is which code fragments we should use and how many test cases we should execute for each code fragment. Ideally, the more code fragments, the more functionally equivalent code we may find; the more test cases used, the more accurate the mined functionally equivalent code may be. On the other hand, even if each execution takes only one tenth of a second, it can take more than a week to sequentially execute each of the 6.5M code fragments once. In the following, we apply several heuristic strategies to address the scalability issue.

Code fragment sampling. Our code chopper generates quadratic number of code fragments *w.r.t.* the number of statements in a function. For example, a function named `serpent_setkey` has more than $1,600$ sequential expression statements which led to more than 1.3M fragments for this function only. It appears uninteresting to consider *every* one of them for functional equivalence, but it is still interesting to consider *some* fragments that are "representative" and collectively cover significant portions of the function.

Our data showed that most functions (more than 85%) in the Linux kernel have fewer than 15 statements, implying most functions will have fewer than 120 code fragments generated. Thus, we again used random sampling: we randomly selected up to 100 code fragments from all the code fragments in each function to be used

in our following study. With this strategy, the total number of code fragments that require further executions became $830,319$, more than seven times smaller than the original 6.5M, but still covering more than 1.6 million lines of code. [2]

<u>**Limiting code execution time.**</u> Intuitively, Most code fragments are small and their executions should finish quickly. On the other hand, there are code fragments that can fall into infinite loops if the randomly generated inputs do not satisfy certain requirements of the code. For example, one code fragment from the non-recursive merge sort in our benchmark looks like the following:

```
for(s=0; s < ArrayLength-b; s+=2*b) {
   ...
}
```

Since the variable `b` is identified as an input variable, random values will be fed into `b`. If `0` is used, the increment statement in the `for` loop will never change the value of `s` and cause an infinite loop.

Therefore, we imposed a limit on how long each execution of a code fragment can take. Preliminary evaluations on hundreds of fragments showed that if a fragment ever finishes, it finished within 0.2 second. Thus, we set the limit to 0.5 second and kill the process if it exceeds that limit, and the output of the execution is marked as a failure for later use. This strategy helped save at least an order of magnitude of CPU time based on our experience.

<u>**Limiting the number of test cases.**</u> Our current study focuses on functionally equivalent code, and any output difference between two pieces of code fragments will set them apart based on our code clustering algorithm. Assuming the input space of any code fragment, including invalid ones is uniformly sampled by our random input value generator, we have reason to believe if two code fragments have the same outputs in ten out of *ten* test cases, they are likely to be functionally equivalent. On the other hand, even if uniform sampling is achieved, two code fragments may behave differently only on very rare cases and it may not be practical to distinguish one from another using random executions. This limitation is similar to that of traditional random testing [8,29]. For example, the following two code fragments only differ at the `if` condition and they will only exhibit different behavior when `input` happens to be `23456`, which is very unlikely, even if we did not limit the range of random generated values (*cf.* Section 3).

```
if ( input < 23456 ) {            if ( input < 23457 ) {
  ...                               ...
} else {                          } else {
  ...                               ...
}                                 }
```

Fortunately, we could consider such code fragments functionally *similar*, although not equivalent, since they only differ at rare cases. Section 5 will discuss more about the concept of *similarity*.

Based on the above considerations, we set the limit for the number of test cases to 10. Thus, each code fragment will be executed at most $5! \times 10 = 1200$ times, taking at most 10 minutes.

In the next section, we will look at the properties of the mined functionally equivalent code clusters.

4.3 Functionally Equivalent Code Fragments

4.3.1 Sorting Benchmark

Within 104 minutes of sequential executions (no parallel executions at all), the 200 code fragments (no grouping at the beginning) were partitioned into 69 equivalence clusters. The following summarizes the results of our inspection.

[2] Note that, unlikely the previous numbers for lines of code counted *w.r.t.* the kernel source files, this number was calculated *w.r.t.* preprocessed files for simplicity. Also, it may be better during the sampling process to ensure the selected code fragments accumulatively cover most statements in each function so that we may miss less functionally equivalent code.

Most of the code fragments in the clusters are portions of the functions they belong to, instead of the whole functions. Their appearance in the same clusters are mainly due to two facts: (1) some portions of the different sorting algorithms are indeed functionally equivalent to each other, *e.g.*, portions of the recursive and non-recursive merge sort, and portions of bubble sort and selection sort; (2) some code fragments overlap with each other so much and there is no functional difference among them. While the second kind of functional equivalence is trivial, the first kind is more interesting since the existence of such functionally equivalence code fragments may indicate the need for extracting commonly used building blocks for certain functionalities, which is one of the reasons why we are carrying out the study on functionally equivalent code.

At the whole-function level, EQMINER correctly clustered the fragments that correspond to bubble sort, selection sort, non-recursive merge sort, and non-recursive quick sort into the same cluster. It is not surprising to see recursive merge sort and recursive quick sort were not in the cluster due to the current way EQMINER handles function calls (*cf.* Section 2.4.4). For the heap sort, we noticed that a local variable in the function was identified as an output variable, which tricked the output comparison to view the code fragment differently. This is so because the local variable is defined as a flag that can affect the control flows of the code; it may not be used in one of the paths and is considered an output variable by our local reaching definition analysis (*cf.* Section 2.4.2). After we added a superficial statement that uses the local variable at the end of the heap sort function, the variable was no longer considered an output variable and the corresponding code fragment was then clustered with the code fragments for the other four algorithms. Alternatively, we used the strengthened definition of output variables (*cf.* Section 2.4.2), the above flag variable was no longer an output variable; however, it led to an opposite problem that some functionally significant variables, *e.g.*, the variable storing the sorted data, were left out, causing both false positives and negatives. Better semantic-aware analyses may be needed to help identify output variables more accurately.

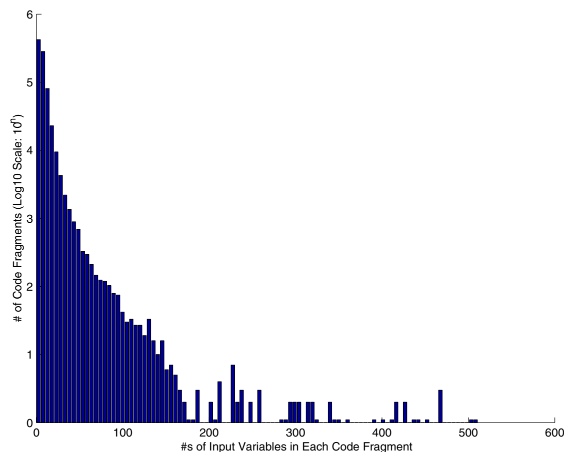
The evaluation based on the sorting algorithms shows the capability of EQMINER on mining functionally equivalent code fragments with satisfying accuracy. On the other hand, it shows the sensitivity of EQMINER on the automatically identified input and output variables. At the function level, it is often easy to improve the validity of the identified input and out variables based on the parameters of the function and its side effects and return values, but it is not obvious how to identify input and output variables for arbitrary portions of the function in general. The def-use analyses used in our approach is a reasonable semantic-aware heuristic, but it will still be worthwhile to investigate other heuristics in the future that can identify functionality-significant variables as inputs and outputs to help reduce both false positive and negative rates.

4.3.2 Linux Kernel

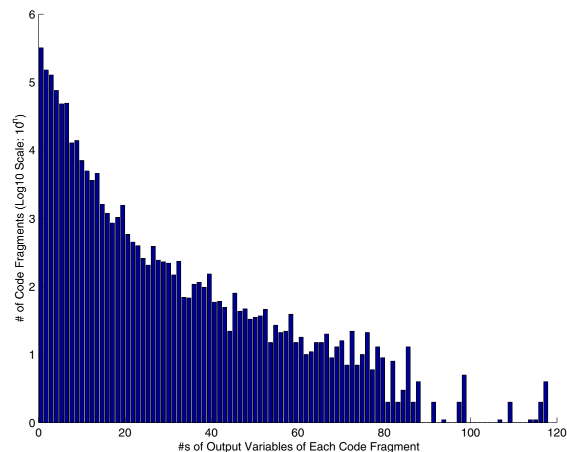
As mentioned in Section 4.2, $830,319$ code fragments were used as candidates for functionally equivalent code.

Based on the numbers of input and output variables every candidate code fragment has, the code fragments were first separated into $2,909$ groups of various sizes. The numbers of input variables in the code fragments range from 0 to 511, and the numbers of output variables range from 0 to 118. Figure 2 show the histograms of the numbers of input and output variables in the code fragments respectively. We can see that most of the code fragments (51%) have fewer than 6 input variables so that our execution strategy on limiting the possible input permutations to $5!$ is not unreasonable.

The sizes of the groups (*i.e.*, the numbers of code fragments in the groups) range from 1 to 76972. The largest group corresponds to code fragments that have 2 input variables and 1 output vari-



(a) Semi-Log Histogram of the Numbers of Input Variables



(b) Semi-Log Histogram of the Numbers of Output Variables

Figure 2: Histograms for Code Fragments.

able. Most groups (more than 90%) contain fewer than 200 code fragments, and more than 1,000 groups contain only one code fragment and there is no need to execute the code fragments in those groups. Also, only 18 groups (fewer than 1%) contain more than 10,000 code fragments. These data suggest that the time required to perform the executions for mining functionally equivalent code should be acceptable on our systems.

The executions of the code fragments in different groups were parallelized. We also parallelized the executions of the code fragments as mentioned at the end of Section 2.6 to speed up the clustering process on groups with large numbers of code fragments.

Since there are many users on our computer cluster system, the number of available hosts on the system varied significantly during the time period when we had our evaluations. Thus, the degree of parallelism was limited, ranging from several to 36 processes at a time. Also, the NFS server which we used to store all the code fragments and related data may have been a bottleneck that limited the actual degree of parallelism. We did not measure how many code executions were parallelized or how much CPU and disk time were. We simply recorded the wall clock time of all the executions, rounded to hours, and that is 189 hours, within 8 days.

The 830,319 code fragments were separated into 269,687 clusters. Most (164,994, more than 60%) of the clusters contain only one fragment, which means most fragments are not functionally equivalent to others. About 30% (82,907) of the clusters contain two to five fragments. Fewer than 1% (1,675) of the clusters contain more than 100 fragments. Many fragments in these large clusters actually overlap with each other, and may be considered trivial. Also, many of these fragments came from functions that are commonly included by many source files. For example, one cluster contained 33,225 fragments, most of which were generated from the same inline, static function `kmalloc` that is included in many preprocessed C files. Although such code fragments are trivially functionally equivalent, it shows the capability of EQMINER on detecting them; When we excluded such code fragments, only 159 non-overlapping fragments were left in the cluster.

Quantity of functionally equivalent code. We used a `code filter` to filter out all but one trivial code fragments in each cluster (which one to keep relies on the order of the occurrence of the overlapping fragments), and remove the cluster if only one code fragment is left in the cluster. We then obtained a set of 32,996 clusters that can be viewed as functional equivalence code clusters, covering about 624K lines of code in the Linux kernel. Figure 3 shows the histogram of the sizes of the clusters. Most of clusters (25,935) contain just two code fragments; very few (14) clusters contain more than 100 code fragments. On the other hand, there are still several clusters containing thousands of code fragments, and the largest one is 3,842.

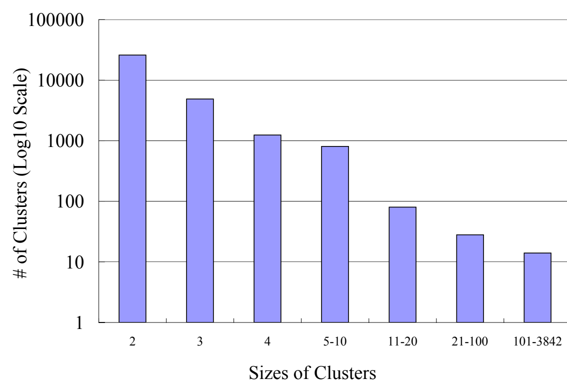


Figure 3: Histogram of the Sizes of Functionally Equivalent Clusters.

The following code represents a common pattern of the code fragments in the largest cluster. A single output variable is identified, and it is assigned a value near the end of the code fragment through an input variable which is introduced due to various reasons, such as a function call or an undefined variable.

```
output = 0;
... /* defs and uses of various variables */
output = input;
```

Assuming the input and output variables identified by EQMINER for these code fragments are appropriate, such code fragments are indeed functionally equivalent according to our definition. However, whether it is really useful to consider them functionally equivalent is still a question worth of future investigation.

Figure 4 shows the spatial distribution of the mined functionally equivalent code in the Linux kernel directories. We can see that the `drivers` directory contains the most functionally equivalent code fragments (more than 35K), while the `block` directory contains the fewest (only 13). This distribution is similar to that of syntactically equivalent code fragments, which we will discuss next.

Differences from syntactically equivalent code. Since many existing techniques can detect syntactically similar code, one may wonder what different results our approach can bring. Answering this question would also help to justify the significance of mining functionally equivalent code fragments in addition to syntactically similar ones. In the following, we thus compare the results of EQMINER with the results from a tree based tool for detecting syntactically similar code–DECKARD [18].

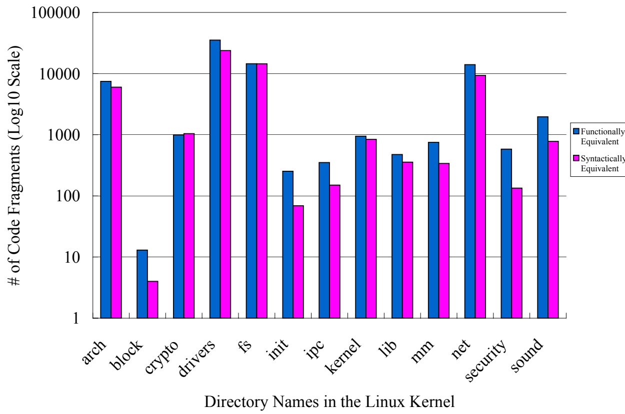


Figure 4: Spatial Distribution of Functionally and Syntactically Equivalent Code in the Linux Kernel.

DECKARD detects syntactically *similar* code by characterizing the syntax tree of a program as a set of vectors and searching for code fragments that have similar characteristic vectors. To have a common ground for comparison, we added the Line 15 in Algorithm 1 to generate a characteristic vector for any code fragment generated by EQMINER, and asked DECKARD to use the set of vectors that correspond to the $830,319$ fragments in our study to search for syntactically *equivalent* code (*i.e.*, set its *similarity* parameter to 1.0, but token level differences are allowed).

On one hand, the spatial distribution of syntactically equivalent code fragments detected by DECKARD is similar to that of functionally equivalent ones (Figure 4). On the other hand, the two sets of code equivalence clusters are different in terms of the code fragments contained in the clusters.

We calculated the percentage of the code fragments that are contained in the clusters reported by EQMINER but not contained in the clusters reported by DECKARD. To our surprise, the percentage was close to 91%, which means only 9% of the functionally equivalent code fragments are also syntactically equivalent. Notice that if there were overlapping code fragments in a cluster, all but one of them (often the first one in the clusters) are removed from the cluster by our code filter, while which one is the first may be different from cluster to cluster, causing the high number of unmatched code fragments between the two sets of clusters. After disabling the code filter, the difference percentage decreased to less than 58%, which means more than 42% functionally equivalent code fragments are syntactically equivalent. We also found that many code fragments reported by DECKARD (more than 36%) were not reported by EQMINER, which means many syntactically equivalent code are functionally different. The still relatively large difference set between the two sets of clusters may be an indication that the two kinds of code detection techniques, functionality-based and syntax-based, can complement each other.

Through a preliminary manual inspection on the difference sets of the clusters, we noticed several categories of the code fragments in the difference set that contribute to the differences:

(1) Many functionally equivalent code is indeed syntactically different. For example, the following two examples are two types of such code that we saw the most; they are functionally equivalent to "`output = input;`" in C programs:

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

(2) Lexical differences cause syntactically equivalent code to function differently. For example, DECKARD considers the code "`if(input < 10) output = 10`" syntactically equivalent to the code "`if(input < 100) output = 100`", while EQMINER considers different. As another example, the following pair of code fragments only differ at a single variable name and are syntactically equivalent, but they are functionally different.

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

(3) False positives produced by EQMINER may have contributed to a large portion of the difference sets. For example, DECKARD recognizes function calls and will consider calls with different numbers of parameters are syntactically different, but EQMINER treats any function call as a random input variable and thus may report false functionally equivalent code fragments. In the following paragraphs, we discuss the accuracy of EQMINER further.

Accuracy. As we mentioned above, we limited the number of test cases executed for each code fragment to 10. This restriction helped improve the performance of EQMINER, but it may cause false positives in the sense that it may incorrectly put functionally different code fragments in a same cluster.

We used additional test cases to evaluate the accuracy of EQMINER. Two different measurements, one stricter than the other, were used to calculate false positive (FP) rates:

First FP Rates: Given a cluster C, its *first false positive rate* $\mathcal{R}_1(C)$ is the number of the code fragments in C that have different outputs from its representative's outputs during the additional testing over the number of all code fragments in C. As a special case, if the former number is just one smaller than the latter, *i.e.*, no code fragments in the cluster are functionally equivalent, we increase the former number by one and thus let $\mathcal{R}_1(C)$ be 100%.

Given a set of clusters $\mathbb{C}$, its *first false positive rate* $\mathcal{R}_1(\mathbb{C})$ is the number of all such code fragments in $\mathbb{C}$ that have different outputs from its corresponding representative's outputs during the additional testing over the total number of all code fragments in $\mathbb{C}$. The special cases when no code fragments in a cluster are functionally equivalent are handled in a way similar to the above.

Second FP Rates: Given a cluster C, its *second false positive rate* $\mathcal{R}_2(C)$ is the number of singleton clusters generated during the additional testing (*i.e.*, the number of code fragments considered functionally nonequivalent to any other code fragments in C) over the number of all code fragments in C.

Given a set of clusters $\mathbb{C}$, its *second false positive rate* $\mathcal{R}_2(\mathbb{C})$ is the number of all such code fragments in $\mathbb{C}$ that are put into singleton clusters during the additional testing over the total number of all code fragments in $\mathbb{C}$.

The first false positive rate is more strict in the sense that it tells how many code fragments in a cluster may not belong to *the* cluster, while the second false positive rate tells how many code fragments in a cluster may not belong to *any* functionally equivalent cluster.

Ideally, we should carry out additional tests for every cluster. Due to the limitation of computing resources, we only focused on 128 clusters each of which contains fewer than 100 code fragments: we randomly selected 50 clusters from the clusters sized between 2 and 4, another 50 clusters from the clusters sized between 5 and 20, and included all of the clusters (28) sized between 20 and 100. We did not choose the clusters in a uniformly random way since the sizes of the clusters are not distributed uniformly (*cf.* Figure 3) and we were trying not to spend too much time on large clusters. The set of clusters we chose contain $1,913$ code fragments, and we denote the whole set as E. We then allowed each code fragment from E to execute with 100 randomly generated inputs.

To compute the first false positive rate $\mathcal{R}_1(E)$, we first executed the representative in every cluster in E with 100 random inputs and recorded their outputs for each of the inputs. Then, we executed all other code fragments in E with the same inputs in parallel. For each code fragment, whenever it generates an output different from the corresponding output of its representative, it is marked as a false positive and its execution is terminated. In addition, if all code fragments in a cluster except for the representative are marked, the representative is also marked as a false positive. This process was finished in about 13 hours on our cluster system, and we obtained the false positive rate $\mathcal{R}_1(E) = 28\%$.

To compute the second false positive rate $\mathcal{R}_2(E)$, we used a different strategy from the above: we simply invoked Algorithm 2 on each of the cluster in E, in parallel, with the number of test cases set to 100. The process was finished in about 16 hours on our cluster system. Although the number of clusters increased to 206 from 128, only 57 code fragments were in singleton clusters, which means the false positive rate $\mathcal{R}_2(E)$ is within 3%.

In addition, we excluded the false positives marked during the calculation of $\mathcal{R}_1(E)$ from E and executed an additional 100 random tests on the rest of E in order to further justify the false positive rates. We noticed that there were additional 69 code fragments marked as false positives, increasing the false positive rate $\mathcal{R}_1(E)$ to 32%. Also, we invoked Algorithm 2 on the resulting clusters (excluding singleton clusters) generated during the calculation of $\mathcal{R}_2(E)$ with additional 100 tests, and we only noticed 6 new singleton clusters, increasing the false positive rate $\mathcal{R}_2(E)$ to 3.5%.

On one hand, the relatively low second false positive rate shows that the code fragments mined by EQMINER are very likely to functionally equivalent to some others. On the other hand, we noticed that several factors may have contributed to the relatively high first false positive rate related to execution failures due to many invalid inputs and inadequate test coverage, besides implementation limitations. Further investigation may help to decide whether directed random testing techniques that combine concrete and symbolic executions [15, 34] can help alleviate our problem which existed in traditional random testing as well [8, 29].

5. DISCUSSIONS

Scalability. As described in Section 4.2, EQMINER employed various heuristics to reduce the expenses of computing resources. Some of the heuristics, such as limiting the number of code fragments by sampling, may lead to missed functionally equivalent code (*i.e.*, false negatives), while others, such as limiting the number of test cases, may increase false positive rates. It will be a challenging and interesting task to scale EQMINER to as many code fragments as possible with as many test cases as possible. It may require the combination of novel techniques and significant engineering efforts to simplify the problem or explore the degree of parallelism in this problem further. Existing program analysis techniques may be of some help. For example, directed testing that combines symbolic and concrete executions [15, 34] may help reduce the number of test cases required for exhibiting the functionality of each code fragment, thus reduce expenses on code executions without increasing false positive rates. Also, program slicing, either static or dynamic [3, 16, 36], may help our code chopper to focus on the most relevant code portions.

Code chopping. In this paper, we generate code fragments mainly based on the syntax of a sequence of statements. In fact, many syntactically consecutive statements may not be semantically related to each other, as shown in [14] that more than half of the functions in the Linux kernel perform more than one unrelated computations. It is intuitively uninteresting to put statements for different computations in a same code fragment and consider it a candidate for functionally equivalent ones. Thus, utilizing program dependency information and generating code fragments based on program slices may help to exclude uninteresting candidates and leave with us more semantic-relevant ones for further consideration; also, since program slices are often smaller than a whole function body, the number of code fragments generated by our code chopping will be smaller and help scale up EQMINER.

Identifying input/output variables. Section 4.3.1 has discussed that EQMINER can be sensitive to the input and output variables identified for each code fragment. The liveness and reaching definition analyses used in EQMINER may include functionally insignificant variables in the sets of input and output variables, causing false positives and negatives. If the code chopping was carried out on program slices, such mis-identifications can be fewer since input and output variables are often more prominent and meaningful along the data flows within slices. Other heuristics, such as statistical learning, may leverage programmers' knowledge and help to identify more appropriate variables as inputs and outputs.

Functional equivalence definition. We in this paper define functional equivalence based on the same I/O behavior, which is different from the traditional concept of *semantic equivalence* defined on program semantics, such as operational semantics. Thus, we in effect do not consider intermediate program states in our definition and have not attempted to detect semantically equivalent code yet. As a result, our approach may not be directly applicable for plagiarism detection, for example, among student programming homework. In principle, we can use functional equivalence to search for semantically equivalent code: first identify the smallest units of code (*e.g.*, a statement) that are functionally equivalent to some others, then look for compositions of such code units that are consecutive and still functionally equivalent to some others. Repeated compositions of consecutive code units may thus form larger code fragments that are semantically equivalent. It would be future work to investigate the feasibility and complexity of such a problem.

We have not explored the concept of functional *similarity* in the sense that we only considered code fragments that are equivalent and have not considered code fragments that are equivalent on certain inputs but different on others. It would be ideal to have a general definition for similarity so that functional differences between code fragments may be quantified and studied further. For example, we may say the following pair of code has a similarity 0.8 since they behave differently on two out of ten inputs:

```
if ( input % 10 == 0 ) {
   output = 0;
} else {
   output = 1;
}
```

```
if ( input % 10 == 1 ) {
   output = 0;
} else {
   output = 1;
}
```

Categorization and application. Categorizing functionally equivalent code fragments may help us to understand the characteristics of the code and understand further about how equivalent code occurs and evolves. One immediate application of such a study will be functionality-based refactoring that helps extract functionally equivalent code into shared libraries for easy reuse. It will be a valuable complement for syntax-based refactoring in the current mainstream. Also, our study can enable semantic-aware code search, in addition to syntax-based search approaches, that may help improve developer productivity. In addition, small functional differences among similar code may be useful for detecting program errors, similar to many other types of inconsistencies that have been used for bug detection [12, 13, 19].

We only performed limited study on the mined code clusters, and have not derived general knowledge about the patterns or characteristics of the code clusters. Future work will investigate in this direction, and aim to increase the confidence on the mined functionally equivalent code fragments and explore their potential applications.

6. RELATED WORK

This section surveys additional closely related work. First, our work is related to checking program equivalence [10], which is a classic problem and is undecidable in general. Definitions of equivalence based on operational semantics have also been proposed long before [30,31]. Definitions based on input and output behavior have also been investigated in the literature [7, 11, 37]. However, to the best of our knowledge, this is the first work that proposes to use random testing for large-scale analysis of functionally equivalence. Although working on different granularities, our technique shares a similar property with any software testing activity [6], such as regression testing, that aims to uncover functional differences among programs and specifications: it guarantees functional differences when it separates code fragments into different clusters, but it can rarely guarantee functionally equivalent.

This work is also related to studies that show much duplicated code exists in large code bases [20,21,25]. Many such duplications can be attributed to poor programming practices since programmers often copy-paste code to quickly duplicate functionality.

There is also a large body of related efforts on detecting similar code, most of which focus on detecting syntactically similar code and can be classified into three categories: (1) *String-based* approaches that use techniques like "parameterized" string matching algorithms [1, 2], (2) *Token-based* approaches that split a program into a token sequence which is scanned for duplicated token subsequences [20, 25], and (3) *Tree-based* approaches that parse a program into parse trees or abstract syntax trees which are scanned for similar subtrees [4, 5, 18, 23, 26, 35]. There are a few semantic-aware techniques for detecting similar code, which can be classified into two categories: (1) *Graph-based* approaches that represent certain semantic information as program dependency graphs, which are then scanned for similar subgraphs [14, 22, 24], and (2) *Birthmark-based* approaches used mainly for detecting illegal theft code where a program is statically or dynamically *fingerprinted* and is checked against the fingerprints of other programs [9, 17, 32, 38]. Different from the above related work, our approach considers the functional behavior of a program and our goal is to detect hidden, but not malicious-intended code equivalence.

7. CONCLUSIONS

We have presented the first scalable algorithm that can automatically discover functionally equivalent code fragments in large, real-world programs. Our insight is that random testing can be used to quickly partition code fragments to detect functional equivalence. We also introduce various analyses and optimizations to enable a large-scale study of code-level functional equivalence in the Linux kernel source code. Our results have shown that functionally equivalent, but syntactically different code fragments commonly exist, which indicates that techniques like ours for detecting functional equivalence is desirable and have many potential applications.

Acknowledgments

We thank Earl Barr, Mark Gabel, Zhongxian Gu, David Hamilton, Andreas, Sæbjørnsen, Jeremiah Willcock, and the anonymous ISSTA reviewers for their detailed and constructive feedback on earlier versions of this paper.

8. REFERENCES

[1] B. S. Baker. On finding duplication and near-duplication in large software systems. In *Working Conference on Reverse Engineering (WCRE)*, pages 86–95, 1995.

[2] B. S. Baker. Parameterized duplication in strings: Algorithms and an application to software maintenance. *SIAM Journal on Computing (SICOMP)*, 26(5):1343–1362, 1997.

[3] S. Bates and S. Horwitz. Incremental program testing using program dependence graphs. In *POPL*, pages 384–396, 1993.

[4] I. D. Baxter, C. Pidgeon, and M. Mehlich. DMS®: Program transformations for practical scalable software evolution. In *ICSE*, pages 625–634, 2004.

[5] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *ICSM*, pages 368–377, 1998.

[6] B. Beizer. *Software Testing Techniques*. The Coriolis Group, 2nd edition, 1990.

[7] M. Bertran, F.-X. Babot, and A. Climent. An input/output semantics for distributed program equivalence reasoning. *Electr. Notes Theor. Comput. Sci*, 137(1):25–46, 2005.

[8] D. L. Bird and C. U. Munoz. Automatic generation of random self-checking test cases. volume 22, pages 229–245, Riverton, NJ, USA, 1983. IBM Corp.

[9] C. S. Collberg and C. D. Thomborson. Software watermarking: Models and dynamic embeddings. In *POPL*, pages 311–324, 1999.

[10] G. Cousineau and P. Enjalbert. Program equivalence and provability. In *MFCS: Symposium on Mathematical Foundations of Computer Science*, 1979.

[11] R. L. Crole and A. D. Gordon. A sound metalogical semantics for input/output effects. In *Computer Science Logic (CSL): 8th workshop*, volume 933 of *LNCS*, pages 339–353, 1995.

[12] I. Dillig, T. Dillig, and A. Aiken. Static error detection using semantic inconsistency inference. In *PLDI*, pages 435–445, 2007.

[13] D. R. Engler, D. Y. Chen, and A. Chou. Bugs as inconsistent behavior: A general approach to inferring errors in systems code. In *SOSP*, pages 57–72, 2001.

[14] M. Gabel, L. Jiang, and Z. Su. Scalable detection of semantic clones. In W. Schäfer, M. B. Dwyer, and V. Gruhn, editors, *ICSE*, pages 321–330, 2008.

[15] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *PLDI*, pages 213–223, 2005.

[16] A. Groce and R. Joshi. Exploiting traces in program analysis. In *TACAS*, volume 3920 of *LNCS*, pages 379–393. Springer, 2006.

[17] H. il Lim, H. Park, S. Choi, and T. Han. Detecting theft of java applications via a static birthmark based on weighted stack patterns. *IEICE Transactions*, 91-D(9):2323–2332, 2008.

[18] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *ICSE*, pages 96–105, 2007.

[19] L. Jiang, Z. Su, and E. Chiu. Context-based detection of clone-related bugs. In *ESEC/FSE*, pages 55–64, New York, NY, USA, 2007. ACM.

[20] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *TSE*, 28(7):654–670, 2002.

[21] M. Kim, V. Sazawal, and D. Notkin. An empirical study of code clone genealogies. In *ESEC/FSE*, pages 187–196, 2005.

[22] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *International Static Analysis Symposium (SAS)*, pages 40–56, 2001.

[23] K. Kontogiannis, R. de Mori, E. Merlo, M. Galler, and M. Bernstein. Pattern matching for clone and concept detection. *Automated Soft. Eng.*, 3(1/2):77–108, 1996.

[24] J. Krinke. Identifying similar code with program dependence graphs. In *Working Conference on Reverse Engineering (WCRE)*, pages 301–309, 2001.

[25] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: A tool for finding copy-paste and related bugs in operating system code. In *OSDI*, pages 289–302, 2004.

[26] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *ICSM*, pages 244–254, 1996.

[27] G. C. Necula, S. Mcpeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of c programs. In *International Conference on Compiler Construction*, pages 213–228, 2002.

[28] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999.

[29] A. J. Offutt and J. H. Hayes. A semantic model of program faults. In *ISSTA*, pages 195–200. ACM Press, 1996.

[30] A. M. Pitts. Operational semantics and program equivalence. In *Applied Semantics, Advanced Lectures*, volume 2395 of *LNCS, Tutorial*, pages 378–412. Springer-Verlag, 2002.

[31] J.-C. Raoult and J. Vuillemin. Operational and semantic equivalence between recursive programs. *Journal of the ACM*, 27(4):772–796, Oct. 1980.

[32] D. Schuler, V. Dallmeier, and C. Lindig. A dynamic birthmark for Java. In *ASE*, pages 274–283, Nov. 2007.

[33] J. T. Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *J. ACM*, 27(4):701–717, 1980.

[34] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *FSE*, pages 263–272, 2005.

[35] V. Wahler, D. Seipel, J. W. von Gudenberg, and G. Fischer. Clone detection in source code by frequent itemset techniques. In *International Workshop on Source Code Analysis and Manipulation*, pages 128–135, 2004.

[36] B. Xin and X. Zhang. Efficient online detection of dynamic control dependence. In *ISSTA*, pages 185–195, New York, NY, USA, 2007. ACM.

[37] V. A. Zakharov. To the functional equivalence of turing machines. *FCT: Fundamentals (or Foundations) of Computation Theory*, 6, 1987.

[38] X. Zhou, X. Sun, G. Sun, and Y. Yang. A combined static and dynamic software birthmark based on component dependence graph. In *International Conference on Intelligent Information Hiding and Multimedia Signal Processing*, pages 1416–1421, 2008.

[39] R. Zippel. An explicit separation of relativised random polynomial time and relativised deterministic polynomial time. *Inf. Process. Lett.*, 33(4):207–212, 1989.