2017 IEEE/ACM 39th International Conference on Software Engineering

# Automated Transplantation and Differential Testing for Clones

Tianyi Zhang, Miryung Kim
University of California, Los Angeles
{tianyi.zhang, miryung}@cs.ucla.edu

***Abstract*—Code clones are common in software. When applying similar edits to clones, developers often find it difficult to examine the runtime behavior of clones. The problem is exacerbated when some clones are tested, while their counterparts are not. To reuse tests for similar but not identical clones,** GRAFTER **transplants one clone to its counterpart by (1) identifying variations in identifier names, types, and method call targets, (2) resolving compilation errors caused by such variations through code transformation, and (3) inserting stub code to transfer input data and intermediate output values for examination. To help developers examine behavioral differences between clones,** GRAFTER **supports fine-grained differential testing at both the test outcome level and the intermediate program state level.**

**In our evaluation on three open source projects,** GRAFTER **successfully reuses tests in 94% of clone pairs without inducing build errors, demonstrating its automated code transplantation capability. To examine the robustness of** GRAFTER**, we systematically inject faults using a mutation testing tool,** MAJOR**, and detect behavioral differences induced by seeded faults. Compared with a static cloning bug finder,** GRAFTER **detects 31% more mutants using the test-level comparison and almost 2X more using the state-level comparison. This result indicates that** GRAFTER **should effectively complement static cloning bug finders.**

## I. INTRODUCTION

Code reuse via copying and pasting is a common practice in software development [1]–[3]. Prior studies show that up to 25% of code in modern software contains *code clones*—code similar to other code fragments elsewhere [4]–[6]. Manually adapting clones is error-prone. Chou et al. show that a large portion of operating system bugs is introduced by manual porting mistakes between clones [7]. Juegens et al. find that "nearly every second unintentionally inconsistent change to a clone leads to a fault" [8]. Therefore, developers may want to examine and contrast runtime behavior of clones. Fischer finds that developers want to see how reused code works in terms of runtime behavior [9]. Holmes et al. find that developers want to leverage existing tests to validate reused code [10]. We also find that industrial developers rely on regression testing to check for inconsistent or missing edits on clones [11].

However, the situation is exacerbated due to a lack of tests, where some clones are tested while their counterparts are not. In fact, our study shows that, in 46% of studied clone pairs, only one clone is tested by existing tests, but not its counterpart (to be detailed in Section IV). No existing techniques can help programmers reason about runtime behavior differences of clones, especially when clones are not identical and when clones are not tested. In the absence of test cases, developers can only resort to static analysis techniques to examine clones [2, 3, 11]–[13], but these techniques are limited to finding only pre-defined types of cloning bugs such as renaming mistakes or control-flow and data-flow inconsistencies.

This paper presents a test reuse and differential testing framework for clones, called GRAFTER. Given a pair of clones and an existing test suite, GRAFTER helps programmers cross-check runtime behavior by exercising the clones using the same test. Test reuse for clones is challenging because clones may appear in the middle of a method without a well-defined interface (i.e., explicit input arguments and return type), which also makes it hard to directly adapt test for reuse. Such intra-method clones are often found by widely-used clone detectors such as Deckard [2] or CCFinder [14]. GRAFTER identifies input and output parameters of a clone to expose its de-facto interface and then grafts one clone in place of its counterpart to exercise the grafted clone using the same test.

Similar to how organ transplantation may bring incompatibility issues between a donor and its recipient, a grafted clone may not fit the context of the target program due to variations in clone content. For example, if a clone uses variables or calls methods that are not defined in the context of its counterpart, simply copying a clone in place of another will lead to compilation errors. To ensure *type safety* during grafting, GRAFTER performs inter-procedural analysis to identify variations in referenced variables and methods. It then adapts the grafted clone using five transplantation rules to handle the variations in referenced variables, types, and method calls. Finally, it synthesizes stub code to propagate input data to the grafted clone and then transfers intermediate outputs back to the recipient. GRAFTER supports differential testing at two levels: test outcomes (i.e., *test-level comparison*) and intermediate program states (i.e., *state-level comparison*). During differential testing, GRAFTER does not assume that all clones should behave similarly nor considers that all behavioral differences indicate bugs. In fact, a prior study on clone genealogies [15] indicates that many syntactically similar clones are used in different contexts and have intended behavioral differences. The purpose of differential testing in GRAFTER is rather to illuminate and expose behavioral differences at a fine-grained level *automatically* and *concretely* by pinpointing which variables' states differ in which test.

We evaluate GRAFTER on 52 pairs of nonidentical clones from three open-source projects: Apache Ant, Java-APNS, and Apache XML Security. GRAFTER successfully grafts and

1558-1225/17 $31.00 © 2017 IEEE
DOI 10.1109/ICSE.2017.67

```
1  public class Copy extends Task{
2   private IncludePatternSet includes;
3
4   public void setIncludes(String patterns){
5     ...
6     if(patterns != null && patterns.length() > 0){
7   -   StringTokenizer tok=new StringTokenizer(patterns,",");
8   -   while(tok.hasMoreTokens()){
9   -     includes.addPattern(tok.next());
10  -   }
11  +   String[] tokens = StringUtils.split(patterns, ",");
12  +   for(String tok : tokens){
13  +     includes.addPattern(tok);
14  +   }
15    }
16  }
17    ...
18  }
19
20  public class IncludePatternSet{
21   public Set<String> set;
22   public void addPattern(String s) { set.add(s); }
23    ...
24  }
```

(a) Correctly edited clone in the Copy class

```
1  public class Delete extends Task{
2   private ExcludePatternSet excludes;
3
4   public void setExcludes(String patterns){
5     ...
6     if(patterns != null && patterns.length() > 0){
7   -   StringTokenizer tok=new StringTokenizer(patterns,",");
8   -   while(tok.hasMoreTokens()){
9   -     excludes.addPattern(tok.next());
10  -   }
11  +   String[] tokens = StringUtils.split(patterns, ".");
12  +   for(String tok : tokens){
13  +     excludes.addPattern(tok);
14  +   }
15    }
16  }
17    ...
18  }
19
20  public class ExcludePatternSet{
21   public Set<String> set;
22   public void addPattern(String s) { set.add(s); }
23    ...
24  }
```

(b) Inconsistenly edited clone in the Delete class

Fig. 1: Similar edits to update the use of `StringTokenizer` API to `StringUtils.split` in `Copy` and `Delete`.

```
1  @Test
2  public void testCopy(){
3    Task copyTask = FileUtils.createTask(FileUtils.COPY);
4    ...
5    copyTask.setIncludes("src/*.java, test/*.java");
6    JobHandler.fireEvent(copyTask);
7    assertTrue(checkFileCopied());
8  }
```

Fig. 2: A test case for the `Copy` class.

reuses tests in 49 out of 52 pairs of clones without inducing compilation errors. Successfully reusing tests in 94% of the cases is significant, because currently no techniques enable test reuse for nonidentical clones appearing in the middle of a method. GRAFTER inserts up to 33 lines of stub code (6 on average) to ensure type safety during grafting, indicating that code transplantation and data propagation in GRAFTER are not trivial. To assess its fault detection capability, we systematically seed 361 mutants as artificial faults using the MAJOR mutation framework [16]. We use Jiang et al.'s static cloning bug finder [2] as a baseline for comparison. By noticing runtime behavioral discrepancies, GRAFTER is more robust at detecting injected mutants than Jiang et al.—31% more using the test-level comparison and almost 2X more using the state-level comparison. GRAFTER's state-level comparison also narrows down the number of variables to inspect to three variables on average. Therefore, GRAFTER should complement static cloning bug finders by enabling runtime behavior comparison. Our grafting technology may also have potential to assist code reuse and repair [17]–[20].

The rest of the paper is organized as follows. Section II illustrates a motivating example. Section III describes how GRAFTER reuses tests from its counterpart clone by grafting a clone. Section IV describes the evaluation of GRAFTER and comparison to Jiang et al. Section V discusses threats to validity and Section VI describes related work.

## II. MOTIVATING EXAMPLE

This section motivates GRAFTER using an example based on Apache Ant. The change scenario is constructed by us to illustrate the difficulty of catching cloning bugs. Figure 1 shows the pair of inconsistently edited clones, one from the `setIncludes` method in the `Copy` class (lines 6-15 in Figure 1a) and the other from the `setExcludes` method in the `Delete` class (lines 6-15 in Figure 1b). These clones are syntactically similar but not identical—the left program uses a field `includes` of type `IncludePatternSet` while the right program uses a field `excludes` of type `ExcludePatternSet`. The `Copy` class implements the task of copying files matching the specified file pattern(s). On the other hand, `Delete` removes files that do not match the pattern(s). Methods `setIncludes` and `setExcludes` both split the input string by a comma and add each pattern to a pattern set, `includes` and `excludes` respectively. Figure 2 shows a test case, `testCopy`, which creates a `Copy` object, specifies two copied file patterns as a string `"src/*.java, test/*.java"`, and then checks if all java files in the `src` folder and the `test` folder are copied to a target directory. However, the `Delete` class is not tested by any existing test.

`StringTokenizer` is a legacy class and its usage is now discouraged in new code. Therefore, Alice updates the use of `StringTokenizer` API to `StringUtils.split` in both `Copy` and `Delete` in Figure 1. However, she accidentally changes the separator from `','` to `'.'` in `Delete` (line 11 in Figure 1b). Such mistake is difficult to notice during manual inspection, as these programs are similar but not identical. An existing cloning bug finder by Jiang et al. would fail to find the mistake, as it checks for only three pre-defined cloning bug types via static analysis [2]: renaming mistakes, control construct inconsistency, and conditional predicate inconsistency. Accidentally replacing the separator does not belong to any of the pre-defined cloning bug types.

```
1  public class Copy extends Task{
2    private IncludePatternSet includes;
3  +  private ExcludePatternSet excludes;
4
5    public void setIncludes(String patterns){
6      ...
7      /* this is stub code inserted for data transfer*/
8  +   ExcludePatternSet excludes_save = excludes;
9  +   excludes = new ExcludePatternSet();
10 +   excludes.set = includes.set;
11
12     /* the original code is replaced with the grafted code
           from setExcludes*/
13 -   if(patterns != null && patterns.length() > 0){
14 -     String[] tokens = StringUtils.split(patterns, ",");
15 -     for(String tok : tokens){
16 -       includes.addPattern(tok);
17 -     }
18 -   }                                   Original Clone (Deleted)
19 +   if(patterns != null && patterns.length() > 0){
20 +     String[] tokens = StringUtils.split(patterns, ".");
21 +     for(String tok : tokens){
22 +       excludes.addPattern(tok);
23 +     }
24 +   }                                   Grafted Clone (Inserted)
25
26     /* this is stub code inserted for data transfer*/
27 +   includes.set = excludes.set;
28 +   excludes = excludes_save;
29   }
30 }
```

Fig. 3: GRAFTER grafts the clone in `Delete` (lines 19-24) in place of the original clone in `Copy` (lines 13-18) for test reuse. GRAFTER inserts stub code (highlighted in yellow).

To reuse the same test `testCopy` for `Delete`, GRAFTER grafts the clone from `Delete` in place of the original clone in `Copy`, as shown in Figure 3. As the grafted code uses an undefined variable `excludes`, GRAFTER also ports its declaration to `Copy.java`. GRAFTER ensures that the grafted clone receives the same input data by populating `excludes` with the value of `includes` (lines 8-10) and transfers the value of `excludes` back to `includes` (lines 27-28). Therefore, the value of `excludes` can flow into the same assertion check of the original test. Additional stub code generated by GRAFTER is highlighted in yellow in Figure 3.

After grafting, GRAFTER then runs `testCopy` on both clones and finds that the test now fails on `Delete`, because the string is not split properly. To help Alice further diagnose failure symptoms, GRAFTER shows that `tokens` has a list { `"src/*.java"`, `"test/*.java"`} in `Copy` but { `"src/*"`, `"java, test/*"`, `"java"` } in `Delete` due to a wrong split. GRAFTER also shows that this difference has propagated to corresponding variables `includes` and `excludes`.

## III. CLONE GRAFTING AND TESTING

GRAFTER takes a clone pair and an existing test suite as input and grafts a clone from the donor to the recipient to make the test(s) of the recipient executable for the donor. A *donor* program is the source of grafting and a *recipient* program is the target of grafting. GRAFTER does not require input clones to be identical. Clones could be the output of an existing clone detector [14, 21] or be supplied by a human. For example, lines 6-15 in `setIncludes` and lines 6-15 in `setExcludes` are clones found by DECKARD [2] in Figure 1. `Delete.java` is the *donor* program and `Copy.java` is the *recipient* program, as Alice wants to reuse the test of `Copy.java` for `Delete.java`.

GRAFTER works in four phases. GRAFTER first analyzes variations in local variables, fields, and method call targets referenced by clones and their subroutines (Phase I). It also matches corresponding variables at the entry and exit(s) of clones, which is used for generating stub code and performing differential testing in later phases. GRAFTER ports the donor clone to replace its counterpart clone and declares undefined identifiers (Phase II). To feed the same test input into the grafted clone, GRAFTER populates the input data to newly ported variables and transfers the intermediate output of the grafted clone back to the test for examination (Phase III). Finally, it runs the same test on both clones and compares test outcomes and the intermediate states of corresponding variables at the exit(s) of clones. We use Figure 1 as a running example throughout this section.

### A. Variation Identification

The goal of Phase I is to identify mappings between method call targets, local variables, and fields at the entry and exit(s) of each clone. GRAFTER leverages inter-procedural analysis to find identifiers referenced by each clone and its subroutines. It then determines which referenced identifiers are defined in the donor but not in the recipient.

There are three goals with respect to finding variable mappings at the entry and exit(s) of each clone. First, we need to identify variables used by the donor clone but not defined in the recipient clone, so GRAFTER can port their declarations in Phase II to ensure type safety and avoid compilation errors. Second, we need to decide the data flowing in and out of the clone at the entry and exit(s), so we can insert stub code to populate values between corresponding variables in Phase III. Third, we compare the states of corresponding variables at clone exit(s) for fine-grained differential testing in Phase IV.

These goals are achieved by capturing the *consumed* variables at the entry of the clone region and the *affected* variables at the exit(s) of the clone region in the control flow graph. A variable is *consumed* by a clone if it is used but not defined within the clone. A variable is *affected* by a clone if its value could be potentially updated by the clone. The consumed variables are associated with the data flowing into the clone and the affected variables are associated with the updated data flowing out of the clone. GRAFTER performs a combination of def-use analysis and scope analysis to identify consumed and affected variables.

Given a clone $F$ and its container method $M$ and class $C$, *consumed variables* at the clone's entry can be approximated:

$$Consumed(F, M, C) = (Def(C) \cup Def(M) \setminus Def(F)) \cap Use(F)$$

Similarly, given a clone $F$, affected variables at an exit point $P$ can be approximated as following:

$$Affected(F, P) = Use(F) \cap In\text{-}Scope(P)$$

To assist the derivation of *consumed* and *affected* variables, we define three functions. $Def(F)$ returns the set of variables declared within the fragment $F$. $Use(F)$ returns the set of variables used within the fragment. $In\text{-}Scope(P)$ returns the set of variables at a program point $P$. The set of affected variables is an over-approximation of the variables that could be updated by a clone at runtime. This set may include variables only read but not mutated by the clone. However, it is guaranteed to include all variables potentially updated by the clone, thus capturing all data flowing out of it.

Consider `setIncludes` in Figure 1. Figure 4 shows an inter-procedural control flow graph. Nodes represent corresponding program statements, solid edges represent control flow, and dashed edges represent method invocation. The gray nodes in Figure 4 represent the clone region $F$ (line 6 and lines 11-15 in Figure 1) in `setIncludes`. The CFG edge entering the clone region is labeled with `<entry>` and the two edges exiting the clone region are labeled with `<exit1>` and `<exit2>`. Each CFG node is labeled with the variables defined and used within the corresponding statement. For example, $Def(F)$ includes `tokens` and `tok`. Variable `patterns` is not included because it is declared as a method parameter in Figure 1, which is before the clone region $F$ (line 6 and lines 11-15). $Use(F)$ returns `includes`, `patterns`, `tokens`, and `tok`. The figure does not show the scope of individual variables but we associate each variable with its scope and visibility. For example, $In\text{-}Scope(< exit_2 >)$ returns `patterns` and `tokens`. Putting these definitions together, the resulting set of consumed variables at the entry of the clone is {`includes`, `patterns`}. The resulting sets of affected variables at the two exit edges are the same: {`includes`, `patterns`}.

By comparing the two sets of consumed variables, {`includes, patterns`} and {`excludes, patterns`} at the entry of clones using name similarity, we find that `includes` and `excludes` are corresponding variables. Therefore, GRAFTER knows that it must port the declaration statement of the field `excludes`. The name similarity is computed using the Levenshtein distance [22], i.e., the minimum number of single-character insertions, deletions, or substitutions required to change one string into the other. The lower the distance is, the more similar two field names are. This mapping information is used to guide the process of redirecting data into the grafted clone and back to the recipient in Phase III. For example, GRAFTER populates the value of `includes` to `excludes` at the entry and transfers the updates on `excludes` back to `includes` at the exit (to be detailed in Section III-C).

### B. Code Transplantation

Simply copying and pasting a clone in place of its counterpart in the recipient could lead to compilation errors due to variations in clone content. Phase II applies five transplantation rules to ensure type safety during grafting. The rules are *sound* in the sense that the resulting grafted code is guaranteed to compile. To ensure type safety, our rules do not convert

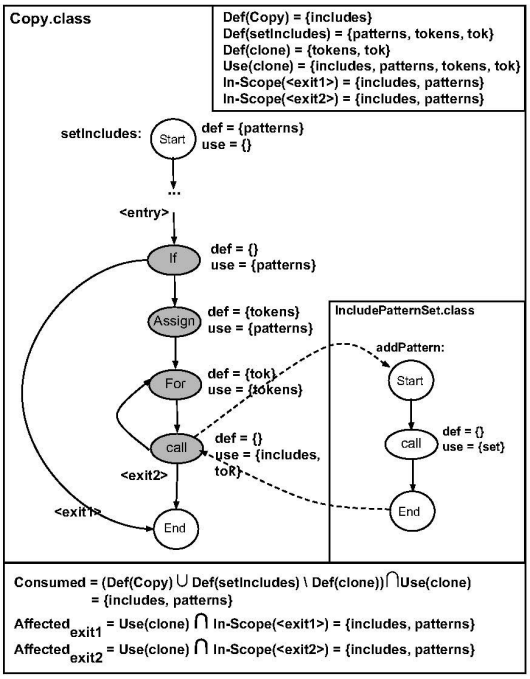



Fig. 4: An inter-procedural control flow graph for the `setIncludes` method in Figure 1. The CFG nodes in the clone region are colored in gray. There is one `entry` edge and two `exit` edges of the clone.

objects, if their types are not castable or structurally equivalent. We conservatively choose not to graft clones referencing such unrelated types and give the user a warning instead.

**Variable Name Variation.** If the grafted clone uses a variable undefined in the recipient, GRAFTER moves its definition from the donor to the recipient. Consider Figure 1 where `setIncludes` and `setExcludes` use different variables, `includes` and `excludes`. When grafting the clone in `setExcludes` to `setIncludes`, GRAFTER adds the definition of `excludes` in line 3 of `Copy.java` in Figure 3. In particular, if the grafted clone uses a variable that has already been defined with a different type in the recipient, GRAFTER still ports the definition but renames it and all its references by appending `_graft` to avoid a naming conflict.

**Method Call Variation.** If the grafted clone calls a method undefined in the recipient, GRAFTER ports its definition from the donor to the recipient. Similar to the rule above, GRAFTER renames it if it leads to a naming conflict.

**Variable Type Variation.** If the grafted clone uses a different type compared with its counterpart, GRAFTER generates stub code to convert the object type to the one compatible with the recipient. For example, `includes` and `excludes` in Figure 1 have different types, `IncludePatternSet` and `ExcludePatternSet`. Simply assigning `includes` to `excludes` leads to a type error. Thus, GRAFTER preserves the original value of `excludes` in line 8, creates a new `ExcludePatternSet` instance in line 9, and populates the field `sets` from the `IncludePatternSet` object to the

**Algorithm 1:** Heuristics for transfering variable values

**Input** : Let `v1` and `v2` be a pair of mapped variables. In this algorithm, each variable symbol is an abstraction, containing the name, type, and field information, which guides the generation of stub code.

**Output**: Let `code` be the stub code to tranfer the value of `v1` to `v2`. It starts with an empty string and ends with a sequence of statements generated using a few heuristics.

```
   Algorithm transfer(v1, v2)
1  |  code := ""
2  |  if v1.name == v2.name then
3  |  |  return ""
4  |  if v1.type == v2.type or v1.type is castable to v2.type then
5  |  |  return "v2.name = v1.name;"
6  |  if v1.type structurally equivalent to v2.type then
7  |  |  code + = "v2.name = new v2.type();"
8  |  |  match := stableMatching (v1.fields, v2.fields)
9  |  |  foreach f_i, f_j in match do
10 |  |  |  code + = transfer (f_i, f_j)
11 |  |  return code
   Procedure stableMatching(s_1, s_2)
12 |  match := ∅
13 |  unmatch := s_2
14 |  while unmatch is not empty do
15 |  |  f_2 := next field in unmatch
16 |  |  foreach f_1 in s_1 do
17 |  |  |  if f_1.type == f_2.type or f_1.type is castable to f_2.type or f_1.type
   |  |  |  is structurally equivalent to f_2.type then
18 |  |  |  |  if f_1 ∈ match.keys then
19 |  |  |  |  |  f'_2 := match.get(f_1)
20 |  |  |  |  |  d_1 := levenshteindistance (f_1.name, f_2.name)
21 |  |  |  |  |  d_2 := levenshteindistance (f_1.name, f'_2.name)
22 |  |  |  |  |  if d_1 < d_2 then
23 |  |  |  |  |  |  match.put(f_1, f_2)
24 |  |  |  |  |  |  unmatch.add(f'_2)
   |  |  |  |  else
25 |  |  |  |  |  match.put(f_1, f_2)
26 |  return match
```

`ExcludePatternSet` object in line 10 in Figure 3.

**Expression Type Variation.** The data type of an expression can be different based on the variables, operators, and method targets used in the expression. Such variation can cause type incompatibility in the returned object if it appears in the `return` statement. GRAFTER first decomposes the return statement `return X;` into two statements, one storing the expression value to a temporary variable `Type temp = X;` and the other returning the temporary value `return temp;`. GRAFTER applies the Variable Type Variation rule above on `temp` to convert its type to a compatible type in the recipient.

**Recursion.** If both container methods in the donor and recipient have recursive calls in the clone region, GRAFTER updates the recursive call targets in the grafted clone.

### C. Data Propagation

In medicine, surgeons reattach blood vessels to ensure the blood in the recipient flows correctly to the vessels of the transplanted organ. Similarly, GRAFTER adds stub code to ensure that (1) newly declared variables consume the same input data as their counterparts in the recipient and (2) the updated values flow back to the same test oracle.

Given each mapped variable pair $v_1$ and $v_2$ in Phase II, GRAFTER generates stub code to propagate the value of $v_2$ to $v_1$ at the entry of the clone and to transfer the updated value of $v_1$ back to $v_2$ at the exit. In Algorithm 1, the main function, `transfer`, takes two variables $v_1$ and $v_2$ as input and produces a sequence of program statements for data propagation. The symbols $v_1$ and $v_2$ in Algorithm 1 abstract their variable name, type, and field information.

**Heuristic A.** Given two variables $v_1$ and $v_2$ with the same name and type, there is no need to propagate the value from $v_1$ to $v_2$. In Figure 1, both clones use the method parameter `patterns` and the references to `patterns` in the grafted code are automatically resolved to the same parameter in the recipient. Algorithm 1 returns an empty string in this case.

**Heuristic B.** Given two variables $v_1$ and $v_2$ of the same type or castable types due to subtyping, the value of $v_2$ can be directly assigned to $v_1$ without inducing type casting errors. Algorithm 1 adds an assignment statement.

**Heuristic C.** Given $v_1$ of type $t_1$ and $v_2$ of type $t_2$, if $t_1$ and $t_2$ are *structurally equivalent*, we propagate corresponding sub fields from $v_2$ to $v_1$. Two types are structurally equivalent if (1) they have the same number of fields, and (2) for each field in one type, there exists a field in another type that has either the same or structurally equivalent type. For example, `IncludePatternSet` and `ExcludePatternSet` are structurally equivalent because both have only one sub-field `set` of type `Set<String>` in Figure 1. To propagate data at the clone entry, GRAFTER first preserves the original `ExcludePatternSet` object in line 8, creates a new `ExcludePatternSet` instance in line 9, and then populates the field `set` from the `IncludePatternSet`'s `set` field in line 10 in Figure 3. At the clone exit, the updates on `excludes` are transferred to `includes` by setting field `set` in line 27 and the original reference is restored to `excludes` in line 28.

Because GRAFTER allows the fields of structurally equivalent types to have different names and orders, GRAFTER identifies n-to-n sub-field matching. This problem can be viewed as a stable marriage problem (SMP) and is solved using the Gale-Shapley algorithm [23]. The `stableMatching` procedure in Algorithm 1 establishes field mappings based on type compatibility and name similarity. The `stableMatching` procedure takes two sets of fields, $s_1$ and $s_2$ as input. It creates an empty map `match` and adds all fields in $s_2$ to `unmatch`. For each field $f_2$ in `unmatch`, GRAFTER compares it with any field $f_1$ in $s_1$. If $f_1$ and $f_2$ have the same or structurally equivalent types and their name similarity is greater than the current match $f_1$ to $f'_2$ (if any), $f_2$ is a better match than $f'_2$. GRAFTER puts a mapped pair ($f_2$, $f_1$) to `match` and adds $f'_2$ to `unmatch`. This process continues until `unmatch` is empty.

**Heuristic D.** If a data type or its field is an array or a collection such as `List` or `Set` of the same or structurally equivalent type, GRAFTER synthesizes a loop, in which each iteration populates corresponding elements using Algorithm 1.

### D. Differential Testing

GRAFTER supports behavior comparison at two levels.

**Test Level Comparison.** GRAFTER runs the same test on two clones and compares the test outcomes. If a test succeeds on one clone but fails on the other, behavior divergence is noted.

**State Level Comparison.** GRAFTER runs the same test on two clones and compares the intermediate program states

for *affected* variables at the exit(s) of the clones. GRAFTER instruments code clones to capture the updated program states at the exit(s) of clones. GRAFTER uses the XStream library[1] to serialize the program states of affected variables in an XML format. Then it checks if two clones update corresponding variables with the same values. State-level comparison is more sensitive than test outcome comparison.

GRAFTER is publicly available with our experiment dataset.[2] Its GUI allows users to experiment with clone grafting and revert edits after examining runtime behavior. Therefore, the inserted stub code is not permanent and does not need to be comprehended by users. Using GUI, users can easily discard tests that do not preserve the desired semantics.

## IV. EVALUATION

Our evaluation investigates three research questions.

- RQ1: How successful is GRAFTER in transplanting code?
- RQ2: How does GRAFTER compare with a static cloning bug finder in terms of detecting behavioral differences?
- RQ3: How robust is GRAFTER in detecting unexpected behavioral differences caused by program faults?

We use DECKARD [21] to find intra-method clones (i.e., clone fragments appearing in the middle of a method) from three open-source projects. Apache Ant is a software build framework. Java-APNS is a Java client for the apple push notification service. Apache XML Security is a XML signature and encryption library. Ant and XML Security are well-known large projects with regression test suites. In Table I, LOC shows the size in terms of lines of code. Test# shows the number of JUnit test cases. Branch and Stmt show branch and statement coverage respectively, both measured by JaCoCo.[3] Pair# shows the number of selected clone pairs.

Because GRAFTER's goal is to reuse tests for nonidentical clones, we include clones meeting the following criteria in our dataset: (1) each clone pair must have at least one clone exercised by some tests, (2) clones must not be identical, because grafting is trivial for identical clones, (3) each clone must have more than one line, (4) each clone must not appear in uninteresting areas such as import statements and comments. These are not restrictions on GRAFTER's applicability, rather we target cases where GRAFTER is designed to help (e.g., tests exist for reuse) and where grafting clone is hard (e.g., clones with variations and without well-defined interfaces).

| Subject | LOC | Test# | Branch | Stmt | Pair# |
|---|---|---|---|---|---|
| ant-1.9.6 | 267,048 | 1,864 | 45% | 50% | 18 |
| Java-APNS-1.0.0 | 8,362 | 103 | 59% | 67% | 7 |
| xmlsec-2.0.5 | 121,594 | 396 | 59% | 65% | 27 |

TABLE I: Subject Programs

### A. Grafting Capability

We use GRAFTER to graft clones in each pair in both directions. A pair of clones is considered successfully grafted if and only if there is no compilation error in both directions. Table II shows 52 clone pairs in our dataset. Column Type shows types of code clones in our dataset based on a well-known clone taxonomy [24, 25]. Type I clones refer to identical code fragments. Type II clones refer to syntactically identical fragments except for variations in names, types, method call targets, constants, white spaces, and comments. Type III clones refer to copied code with added and deleted statements. Because variations in variable names, types, method calls, and constants are all grouped as Type II clones, we enumerate individual kinds of variations in column Variation. Since grafting identical code is trivial, our evaluation focuses on Type II and III clones not to artificially inflate our results.

Column Tested indicates whether both clones are covered by an existing test suite (i.e., *full*) or only one of the two is covered (i.e., *partial*). Success shows whether GRAFTER successfully grafts clones without inducing compilation errors. $\Delta$ shows lines of stub code inserted by GRAFTER and Prpg shows the kinds of heuristics applied for data propagation, described in Section III-C. Branch and Stmt show the branch and statement coverage over the cloned regions in each pair before and after test reuse respectively.

GRAFTER successfully grafts code in 49 out of 52 pairs (94%). In 3 cases, GRAFTER rejects transplantation, because it does not transform objects, unless they are hierarchically related or structurally equivalent to ensure type safety. In Table II, the corresponding rows are marked with ✗ and —. On average, 6 lines of stub code is inserted. GRAFTER synthesizes assignment statements for 30 cases, in 3 of which require type transformation between objects with structurally equivalent types. GRAFTER synthesizes a loop to propagate data between two arrays of structurally equivalent objects in pair#24. Even when the mapped variables have the same type and name, additional stub code may be required, when one clone references extra identifiers undefined in another context (e.g., pairs#47, 48, 52). In some cases, the generated stub code is over 30 lines long, indicating that naïve cut and paste is definitely inadequate for ensuring type safety and data transfer. GRAFTER automates this complex stub code generation.

By reusing tests between clones, GRAFTER roughly doubles the statement coverage and the branch coverage of partially tested clone pairs. For partially tested Type II clone pairs, statement coverage improves from 49% to 98% and branch coverage improves from 34% to 68%. For partially tested Type III clone pairs, we observe similar improvement (36% to 68% and 27% to 49%). For fully tested clones pairs, statement coverage is still improved by augmenting tests.

### B. Behavior Comparison Capability

We use GRAFTER to detect behavioral differences on the 49 pairs of successfully grafted clones. We hypothesize that by noticing fine-grained behavior differences at runtime, GRAFTER can detect potential cloning bugs more effectively than Jiang et al. [2] that detect three pre-defined bug types:

- Rename Mistake: in Figure 5a, the right clone performs a null check on `l_stride` but then dereferences `r_stride`

[1] http://x-stream.github.io/
[2] http://web.cs.ucla.edu/~tianyi.zhang/grafter.html
[3] http://www.eclemma.org/jacoco/

| ID | Type | Variation | Tested | Graft | | | Branch | | Stmt | | Behavior Comparison | | | Mutation | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | Success | Δ | Prpg | Before | After | Before | After | Test | State | Jiang | Test | State | Jiang |
| 1 | II | var, call | full | ✓ | 9 | B | 75% | 75% | 100% | 100% | 0/8 | 0/4 | ✗ | 10/28 | 12/28 | 12/28 |
| 2 | II | var | full | ✓ | 8 | B | 100% | 100% | 100% | 100% | 0/10 | 0/2 | ✗ | 4/4 | 4/4 | 4/4 |
| 3 | II | var | full | ✓ | 8 | B | 50% | 50% | 75% | 75% | 0/1 | 0/2 | ✗ | 2/4 | 2/4 | 4/4 |
| 4 | II | var, call | full | ✓ | 9 | B | 50% | 50% | 75% | 75% | 0/1 | 0/3 | ✗ | 2/6 | 2/6 | 6/6 |
| 5 | II | var, call | full | ✓ | 10 | A, B | 100% | 100% | 100% | 100% | 0/13 | 2/7 | ✗ | 4/8 | 4/8 | 8/8 |
| 6 | II | var | full | ✓ | 6 | B | 100% | 100% | 100% | 100% | 0/38 | 2/3 | ✗ | 12/12 | 12/12 | 0/12 |
| 7 | II | var | full | ✓ | 23 | B | 100% | 100% | 100% | 100% | 0/38 | 5/5 | ✗ | 20/22 | 20/22 | 6/22 |
| 8 | II | var | full | ✓ | 8 | B | 50% | 50% | 88% | 88% | 0/36 | 2/2 | ✗ | 2/6 | 6/6 | 4/6 |
| 9 | II | type, call | full | ✓ | 16 | A | 33% | 66% | 50% | 80% | 2/3 | 4/7 | ✗ | — | — | — |
| 10 | II | var | full | ✓ | 3 | A, B | 88% | 100% | 93% | 100% | 0/37 | 8/9 | ✗ | 13/30 | 30/30 | 12/30 |
| 11 | II | var | full | ✓ | 7 | B | 100% | 100% | 100% | 100% | 0/14 | 3/5 | ✗ | 4/4 | 4/4 | 4/4 |
| 12 | II | call | full | ✓ | 0 | B | 50% | 50% | 75% | 75% | 60/157 | 3/4 | ✗ | — | — | — |
| 13 | II | var, type, call, lit | full | ✓ | 6 | A, C | 63% | 63% | 100% | 100% | 1/1 | 10/11 | ✗ | — | — | — |
| 14 | II | var, type, call, lit | full | ✓ | 6 | A, C | 63% | 63% | 100% | 100% | 1/1 | 10/11 | ✗ | — | — | — |
| 15 | II | type, call | full | ✗ | — | — | 33% | — | 45% | — | — | — | — | — | — | — |
| 16 | II | var, type, call | full | ✓ | 4 | A, B | 100% | 100% | 100% | 100% | 15/45 | 2/7 | ✗ | — | — | — |
| 17 | II | var, type | full | ✓ | 3 | B | 25% | 25% | 14% | 14% | 54/54 | 4/5 | ✓ | — | — | — |
| 18 | II | var | full | ✓ | 6 | B | 75% | 100% | 75% | 100% | 0/116 | 0/3 | ✗ | 4/6 | 4/6 | 0/6 |
| 19 | II | lit | full | ✓ | 0 | A | 25% | 25% | 17% | 17% | 0/1 | 1/4 | ✗ | 2/6 | 2/6 | 6/6 |
| 20 | II | type, lit | full | ✓ | 0 | A | 66% | 66% | 80% | 80% | 4/4 | 2/2 | ✗ | — | — | — |
| 21 | II | lit | full | ✓ | 0 | A | 75% | 75% | 83% | 83% | 0/2 | 0/4 | ✗ | 2/6 | 2/6 | 4/6 |
| 22 | II | var | full | ✓ | 9 | B | 50% | 50% | 71% | 71% | 0/307 | 0/5 | ✗ | 2/24 | 4/24 | 6/24 |
| 23 | II | call | full | ✓ | 0 | A | 100% | 100% | 100% | 100% | 160/168 | 2/3 | ✗ | — | — | — |
| 24 | II | type, lit | full | ✓ | 11 | A, C, D | 70% | 70% | 77% | 77% | 1/1 | 1/4 | ✗ | — | — | — |
| Type II (full) | | | | 23/24 (96%) | 7 | | 65% | 68% | 80% | 84% | 9/23 (39%) | 16/23 (70%) | 1/23 (4%) | 83/166 (50%) | 108/166 (65%) | 76/166 (46%) |
| 25 | II | var | partial | ✓ | 6 | B | 50% | 100% | 50% | 100% | 0/1 | 1/29 | ✗ | 2/4 | 4/4 | 4/4 |
| 26 | II | var | partial | ✓ | 6 | B | 50% | 100% | 50% | 100% | 0/1 | 1/29 | ✗ | 2/4 | 3/4 | 4/4 |
| 27 | II | var | partial | ✓ | 6 | B | 50% | 100% | 50% | 100% | 0/1 | 1/29 | ✗ | 3/4 | 4/4 | 0/4 |
| 28 | II | lit | partial | ✓ | 0 | A | 25% | 50% | 42% | 84% | 0/1 | 2/2 | ✗ | 4/12 | 12/12 | 4/12 |
| 29 | II | var | partial | ✓ | 6 | B | 50% | 100% | 50% | 100% | 0/4 | 1/3 | ✗ | 2/2 | 2/2 | 0/2 |
| 30 | II | var, type | partial | ✓ | 6 | B | 50% | 100% | 50% | 100% | 0/4 | 1/3 | ✓ | 2/2 | 2/2 | 1/2 |
| 31 | II | var, type | partial | ✓ | 6 | B | 50% | 100% | 50% | 100% | 0/4 | 1/3 | ✓ | 2/2 | 2/2 | 1/2 |
| 32 | II | var | partial | ✓ | 3 | A, B | 50% | 100% | 50% | 100% | 0/5 | 0/1 | ✗ | 2/2 | 2/2 | 2/2 |
| 33 | II | var, lit | partial | ✓ | 3 | B | 25% | 50% | 50% | 100% | 0/1 | 1/2 | ✗ | 0/8 | 2/8 | 2/8 |
| 34 | II | type, call | partial | ✓ | 0 | A | 25% | 50% | 50% | 100% | 0/21 | 5/7 | ✗ | 6/20 | 7/20 | 4/20 |
| 35 | II | var, type, call | partial | ✗ | — | — | 50% | — | 50% | — | — | — | — | — | — | — |
| 36 | II | var, type, call, lit | partial | ✗ | — | — | 25% | — | 50% | — | — | — | — | — | — | — |
| 37 | II | var, lit, call | partial | ✓ | 4 | B | 25% | 50% | 50% | 100% | 2/2 | 1/2 | ✗ | — | — | — |
| 38 | II | var, lit | partial | ✓ | 3 | B | 25% | 50% | 50% | 100% | 0/1 | 1/2 | ✗ | 1/8 | 6/8 | 2/8 |
| Type II (partial) | | | | 12/14 (86%) | 4 | | 34% | 68% | 49% | 98% | 1/12 (8%) | 11/12 (92%) | 2/12 (17%) | 26/68 (38%) | 46/68 (68%) | 24/68 (36%) |
| Type II Total | | | | 35/38 (92%) | 6 | | 59% | 68% | 73% | 87% | 10/35 (29%) | 27/35 (77%) | 3/35 (9%) | 109/234 (47%) | 154/234 (66%) | 100/234 (43%) |
| 39 | III | call, extra | full | ✓ | 2 | A | 100% | 100% | 100% | 100% | 10/21 | 4/6 | ✓ | 22/26 | 22/26 | 10/26 |
| 40 | III | call, lit, extra | full | ✓ | 33 | A | 38% | 38% | 68% | 68% | 1/3 | 2/8 | ✓ | — | — | — |
| 41 | III | type, extra | full | ✓ | 0 | A | 70% | 70% | 100% | 100% | 0/4 | 6/11 | ✓ | 18/30 | 23/30 | 8/30 |
| 42 | III | var, extra | full | ✓ | 0 | A | 51% | 68% | 70% | 81% | 33/156 | 8/13 | ✓ | — | — | — |
| Type III (full) | | | | 4/4 (100%) | 9 | | 54% | 65% | 77% | 84% | 3/4 (75%) | 4/4 (100%) | 4/4 (100%) | 40/56 (71%) | 45/56 (80%) | 18/56 (32%) |
| 43 | III | var, call, extra | partial | ✓ | 32 | B | 36% | 64% | 50% | 88% | 0/3 | 1/5 | ✓ | 15/29 | 29/29 | 7/29 |
| 44 | III | call, extra | partial | ✓ | 8 | A | 33% | 66% | 43% | 86% | 2/2 | 3/8 | ✓ | — | — | — |
| 45 | III | var, extra | partial | ✓ | 10 | B | 20% | 40% | 20% | 40% | 14/14 | 2/3 | ✓ | — | — | — |
| 46 | III | var, extra | partial | ✓ | 6 | B | 25% | 50% | 25% | 50% | 14/14 | 2/3 | ✓ | — | — | — |
| 47 | III | call, extra | partial | ✓ | 1 | A | 25% | 50% | 46% | 100% | 0/2 | 4/5 | ✓ | 3/38 | 3/38 | 0/38 |
| 48 | III | extra | partial | ✓ | 1 | A | 25% | 50% | 30% | 70% | 0/4 | 1/4 | ✓ | 0/4 | 0/4 | 2/4 |
| 49 | III | var, lit, extra | partial | ✓ | 4 | A, B | 17% | 50% | 30% | 70% | 4/4 | 4/4 | ✗ | — | — | — |
| 50 | III | var, lit, extra | partial | ✓ | 4 | A, B | 17% | 50% | 30% | 70% | 4/4 | 5/5 | ✓ | — | — | — |
| 51 | III | var, lit, extra | partial | ✓ | 1 | A | 17% | 50% | 30% | 70% | 4/4 | 5/5 | ✓ | — | — | — |
| 52 | III | lit, extra | partial | ✓ | 3 | A | 17% | 50% | 30% | 70% | 1/1 | 3/3 | ✓ | — | — | — |
| Type III (partial) | | | | 10/10 (100%) | 7 | | 27% | 49% | 36% | 68% | 7/10 (70%) | 10/10 (100%) | 9/10 (90%) | 18/71 (25%) | 32/71 (45%) | 9/71 (13%) |
| Type III Total | | | | 14/14 (100%) | 8 | | 39% | 60% | 56% | 81% | 10/14 (71%) | 14/14 (100%) | 13/14 (93%) | 58/127 (46%) | 77/127 (61%) | 27/127 (21%) |
| Type II & III Total | | | | 49/52 (94%) | 6 | | 50% | 72% | 67% | 83% | 18/47 (38%) | 39/47 (83%) | 14/47 (29%) | 167/361 (46%) | 231/361 (64%) | 127/361 (35%) |

TABLE II: Evaluation Benchmark

due to a renaming mistake.

- Control-flow Construct Inconsistency: in Figure 5b, the left clone is enclosed in an `if` statement, while the right clone is enclosed in a `for` loop.
- Conditional Predicate Inconsistency: in Figure 5c, though both clones are in `if` branches, the `if` predicates are different: one calls `strncmp` which takes three arguments, while the other calls `strcmp` which takes two arguments.

In Table II, Behavior Comparison shows whether GRAFTER detects behavioral differences in each clone pair. Test shows how many tests exhibit different test outcomes. State shows how many variables exhibit state differences using GRAFTER's state-level comparison. Jiang shows whether Jiang et al. detect cloning bugs ✓or not ✗.

GRAFTER detects test-level differences in 20 pairs of clones and detects state-level differences in 41 pairs. On the other hand, Jiang et al. detect differences only in 16 pairs because they ignore behavioral differences at runtime caused by using different types and calling different methods. For example, pair#9 from Apache Ant in Figure 7 uses different object

```
if(l_stride!=NULL){
  mps_cdiv_q(X1,X1,
        l_stride->value);
}
```

```
if(l_stride!=NULL){
  mps_cdiv_q(X1,X1,
        r_stride->value);
}
```

(a) Renaming mistake.

```
if(cmd_type==READ_M2){
  msgbuf[xa_count*3]=0;
  msg(DBG_XA1, ...);
}
```

```
for(i=0;i<count;i++){
  msgbuf[i*3]=0;
  msg(DBG_SQ1, ...);
}
```

(b) Control-flow construct inconsistency.

```
if(length>=9&& strcmp(
   buffer,"EESOXSCSI",9){
  buffer+=9;
  length+=9;
}
```

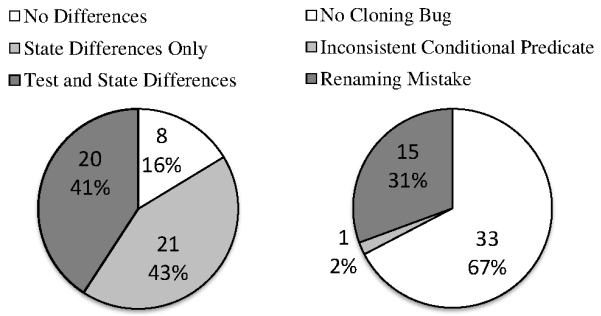
```
if(length>=11&& strncmp(
   buffer,"CUMANNASCSI2"){
  buffer+=11;
  length+=11;
}
```

(c) Conditional predicate inconsistency.

Fig. 5: Three examples of cloning bugs by Jiang et al.

types, `TarFileSet` and `ZipFileSet`. Given the same input variables `p` and `o`, the clones enter different branches due to different runtime type checks (i.e., `instanceof` predicates). Because these runtime checks are syntactically isomorphic and there are no renaming mistake, Jiang et al. report no inconsistency. 13 of 15 renaming mistakes detected by Jiang et al. are in Type III clones, because Jiang et al. compare unique identifiers in each clone to detect renaming mistakes and added statements often lead to extra variable counts. In other words, by definition, they consider almost all Type III clones as cloning bugs.

Figure 6a shows that state-level difference is noted in 84% of pairs, while test outcome difference is noted in 41% of pairs. State-level comparison being more sensitive than test-level comparison is expected, because some program states are not examined by test oracle checking. As GRAFTER focuses its comparison scope to only affected variables, the size of state-level comparison is manageable, three variables on average.




(a) GRAFTER (b) Jiang et al.

Fig. 6: Comparison between GRAFTER and Jiang et al.

### C. Fault Detection Robustness

To systematically assess the robustness of GRAFTER in detecting unexpected behavioral differences caused by program faults, we use the MAJOR mutation framework to inject 361 mutants into 30 pairs of clones. The 19 pairs where the test-level comparison already exhibits differences without adding mutants are marked with — and excluded from the study in order not to over-inflate our results. Each mutant represents an artificial cloning bug and it is injected to only one clone in each pair. We then use GRAFTER to check whether behavioral difference is exhibited at runtime. Table III shows eight kinds of mutants injected by MAJOR. A mutant is detected by GRAFTER's test-level comparison, if GRAFTER exposes test outcome differences in one or more tests after injecting the mutant. A mutant is detected by GRAFTER's state-level comparison, if there is an affected variable with a state value different from its corresponding variable's state value.

In Table II, columns in Mutation show the mutation experiment results. Test shows how many mutants are detected using GRAFTER's test-level comparison. For example, 10/28 indicates that 10 out of 28 mutants are detected using GRAFTER's test-level comparison. Similarly, State shows how many mutants are detected using GRAFTER's state-level comparison while Jiang shows how many mutants are detected by Jiang et al.

| Operator | Description | Example |
|---|---|---|
| AOR | Arithmetic operator replacement | $a+b \rightarrow a-b$ |
| LOR | Logical operator replacement | $a \wedge b \rightarrow a\|b$ |
| COR | Conditional operator replacement | $a \vee b \rightarrow a\&\&b$ |
| ROR | Relational operator replacement | $a == b \rightarrow a >= b$ |
| SOR | Shift operator replacement | $a >> b \rightarrow a << b$ |
| ORU | Operator replacement unary | $\neg a \rightarrow \sim a$ |
| STD | Statement deletion operator: delete (omit) a single statement | foo(a, b) → // foo (a, b) |
| LVR | Literal value replacement: replace by a positive value, a negative value or zero | 0 → 1<br>0 → -1 |

TABLE III: 8 kinds of mutants injected by MAJOR

Overall, GRAFTER detects 167 mutants (46%) using the test-level comparison and 231 mutants (64%) using the state-level comparison. This finding that the state-level comparison is more sensitive to seeded mutants than the test-level comparison is consistent with the literature of strong and weak mutation testing [26]–[28]. Jiang et al. detect 127 mutants (35%) only, as shown in Figure 8. GRAFTER outperforms Jiang et al. by detecting 31% more mutants at the test level and almost twice more at the state level. Its mutant detection ability is similar for both Type II and III clones.

Figure 9 shows GRAFTER is less biased than Jiang et al. when detecting different kinds of mutants. Jiang et al. detect 60% of COR mutants and 44% of STD mutants but less than 20% in other mutant types. This is because Jiang et al. only detect three pre-defined types of cloning bugs—removed statements (STD mutants) often flag renaming mistakes, and COR mutants flag inconsistent conditional predicate updates. Because many removed statements do not affect program states examined by test oracles, GRAFTER's test-level comparison detects fewer STD mutants than Jiang et al. GRAFTER does not detect mutants in eight AOR mutants, because they are all injected in an untested branch in pair#22. Jiang et al. do not detect these AOR mutants because they ignore inconsistencies in arithmetic operators. In summary, our experiment shows

```
File: /org/apache/tools/ant/types/TarFileSet.java
protected AbstractFileSet getRef(Project p){
  dieOnCircularReference();
  Object o = getRefid().getReferencedObject(p);
  if(o instanceof TarFileSet){
    return (AbstractFileSet)o;
  }else if (o instanceof FileSet){
    TarFileSet zfs = new TarFileSet((FileSet)o);
    configureFileSet(zfs);
    return zfs;
  }else{
    throw new Exception(..);
  }
}
```

```
File: /org/apache/tools/ant/types/ZipFileSet.java
protected AbstractFileSet getRef(Project p){
  dieOnCircularReference();
  Object o = getRefid().getReferencedObject(p);
  if(o instanceof ZipFileSet){
    return (AbstractFileSet)o;
  }else if (o instanceof FileSet){
    ZipFileSet zfs = new ZipFileSet((FileSet)o);
    configureFileSet(zfs);
    return zfs;
  }else{
    throw new Exception(..);
  }
}
```

Fig. 7: Type II clones (Pair#9) where GRAFTER detects behavioral differences and Jiang et al do not.

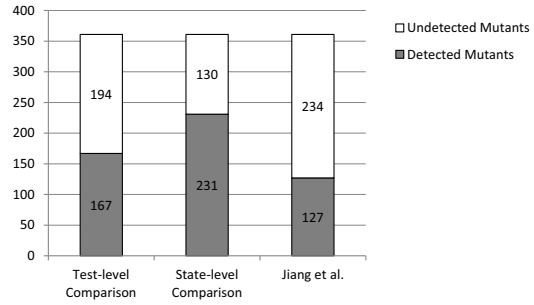



Fig. 8: Mutant detection

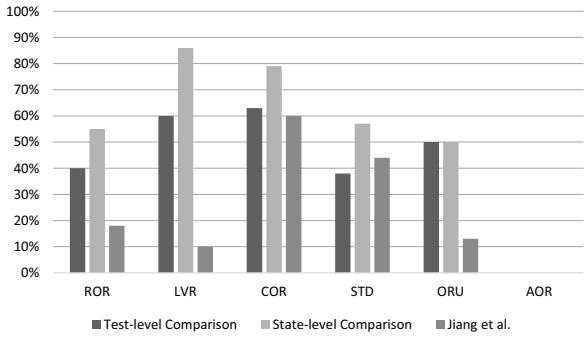



Fig. 9: Mutant killing ratio for different mutant kinds

that GRAFTER can complement a static cloning bug finder via test reuse and differential testing.

## V. DISCUSSION

In terms of *external validity*, since clones in our study are found using an AST-based clone detector [21], our dataset does not include Type IV clones—functionally similar code without any syntactic resemblance. For Type IV clones, a programmer may need to provide the correspondence between variables to enable differential testing. In Section IV-C, we use mutants as a replacement for real faults to assess the robustness of GRAFTER. Recent studies [29, 30] find a strong correlation between mutants and real faults, so our results should generalize to real faults.

In terms of *internal validity*, like other dynamic approaches, GRAFTER's capability to expose behavioral differences is affected by test coverage and quality. If an existing test covers only some branches within a clone, GRAFTER may not expose behavioral differences in uncovered code. However, GRAFTER is still useful for boosting test coverage through code transplantation. GRAFTER's transplantation is guided by the syntactic resemblance of input and output variables. GRAFTER matches variables based on name and type similarity. This heuristic works well for real-world clones in our evaluation, which can be attributed to the fact that these clones are intra-project clones and developers in the same project may follow similar naming conventions. However, manual adjustments may be needed when variables have significantly different names. Experimentation with cross-project clones and alternative matching heuristics [31, 32] remain as future work.

In terms of *construct validity*, GRAFTER conservatively chooses not to graft clones referencing unrelated types—not castable nor structurally equivalent. This limit can be overcome by allowing programmers to provide user-defined type transformation functions. GRAFTER grafts clones rather than tests. Transplanting tests could have the advantage of minimizing impact on the functionality under test. Extending GRAFTER to transplant tests remains as future work.

In Section IV-B, we do not assume that all clones should behave similarly at runtime nor we argue that all behavioral differences indicate cloning bugs. Rather, GRAFTER helps detect behavioral differences concretely and automatically. Therefore, it is necessary for the authors of clones to confirm whether detected behavioral differences are intended or represent potential bugs. Assessing if the generated tests are valuable to the authors remains as future work.

## VI. RELATED WORK

***Test Reuse for Clones.*** Skipper copies associated portions of a test suite when developers reuse code [33]. It determines relevant test cases and transforms them to fit the target system. Skipper is built on Gilligan [10, 34] and requires users to provide a *pragmatic reuse plan* to guide the transformation process. Skipper assumes that clones are full-feature clones at the level of methods and classes. It is infeasible to empirically compare GRAFTER with Skipper due to the requirement of having a reuse plan. GRAFTER differs from Skipper in three perspectives. First, GRAFTER supports sub-method level clones without explicit interfaces. Second, GRAFTER does not require a reuse plan but rather leverages the syntactic resemblance between clones to guide the grafting process. Third, Skipper may require manual adjustments to fix compilation errors when the reuse plan is incomplete. In contrast, GRAFTER is

fully automated using transplantation rules (Section III-B) and data propagation heuristics (Section III-C).

By inferring code correspondences, Jigsaw [35] decides which code to be copied, renamed, or deleted. In contrast, GRAFTER aims to enable differential testing for clones.

***Clone Inconsistency Detection.*** Several static techniques detect copy and paste bugs. CP-Miner finds renaming mistakes between clones [3]. Jiang et al. detect three types of cloning bugs (mentioned in Section IV) [2]. SPA also detects *redundant operations* along with similar cloning bug types [13]. Lo et al. actively incorporate incremental user feedback to selectively filter cloning bugs [36]. Critics leverages interactive user feedback to identify inconsistent edits to clones [11]. Unlike these techniques, GRAFTER grafts clones to facilitate test reuse and enable runtime behavior comparison.

***Dynamic Clone Detection.*** Dynamic clone detectors can find semantic clones that static clone detectors do not find [37]–[40]. Juergens et al. manually inspect code with similar I/O behavior and find that behaviorally similar code can be syntactically different [41]. Jiang et al. detect functionally equivalent code fragments in C based on input and output behavior using random testing [37]. Elva et al. also use random testing but work at the method level [38]. GRAFTER differs from these tools by automatically enabling test reuse via code grafting.

***Differential Testing.*** It is difficult to define the oracle of automatically generated tests without prior knowledge of expected outputs. Differential testing addresses this problem by examining test outputs of comparable systems [42]–[45]. For example, Groce et al. randomly simulate system failures and compare their fault tolerance capability under different systems [44]. Daniel et al. automatically generate Java programs to test refactoring engines via differential testing [45]. All these techniques require target code to have the same interface of input and output arguments, whereas GRAFTER is applicable to clones without a clear interface.

Several differential testing techniques use a *record-and-replay* approach to create the same test environment. For example, Saff et al. generate unit tests from system tests by recording interactions such as method calls and referenced objects and by introducing wrapper classes to load serialized interactions [46]. Instead of serializing actual objects, Orso and Kennedy record unique object ids in a reference map and inserts stub code to look up the reference map [47]. Elbaum et al. [48] detect heap objects reachable from a given method using k-bound analysis [49] and serialize reachable objects before and after the execution as the pre-state and post-state. Diffut allows simultaneous execution and comparison of corresponding methods between different program revisions [50]. These techniques target regression testing scenarios and assume that identifier names stay unchanged between revisions. Unlike these techniques, GRAFTER handles name and type variations between clones to enable differential testing.

***Software Transplantation.*** The way GRAFTER grafts clones for test reuse resembles software transplantation techniques. Petke et al. use code grafting and genetic programming to specialize miniSAT for high performance combinatorial interaction testing [19]. Harman et al. introduce a *grow-and-graft* approach that transplants new functionality into an existing system [20]. This approach requires developers to provide hints regarding where to capture functionality from and how to constrain search space. $\mu$SCALPEL transplants arbitrary functionality from a donor system to a target system [18]. It requires organ entry and implantation points, similar to how GRAFTER requires donor and recipient clones. $\mu$SCALPEL first over-approximates graftable code through slicing and reduces and adapts it through genetic programming. To guide a generic search algorithm, $\mu$SCALPEL requires the existence of test suites at both the origin program and the target program. In contrast, GRAFTER does not require both clones to be already tested by existing tests. In GRAFTER, organ extraction and adaptation is not a search problem, rather a deterministic process guided by syntactic resemblance and its goal is to reveal behavior differences between clones at runtime. The Java type-safe grafting technology presented in our paper may have the potential to be used for automated code reuse.

Genprog transplants code from one location to another for automated repair [17]. Such technique relies on existing tests as oracles, and thus may not find repair solutions effectively, when a test suite is inadequate. A recent study shows that using human-written tests to guide automated repair leads to higher quality patches than using automatically generated tests [51]. By reusing human-written tests and boosting test coverage for similar code, GRAFTER may help improve test oracle guided repairs. Sidiroglou-Douskos et al. present Code Phage, a system for automatically transferring input validation checks [52]. They require an error generating input and a normal input. Through instrumented execution, they obtain a symbolic expression tree encoding a portable check. GRAFTER is not limited to porting input validation checks.

## VII. CONCLUSION

Up to a quarter of software systems consist of code clones from somewhere else. However, the cost of developing test cases is high, which makes test reuse among similar code attractive. This paper introduces GRAFTER, the first test transplantation and reuse approach for enabling runtime behavior comparison between clones. To handle the use of different types, methods, and variables in clones, GRAFTER automatically inserts stub code to propagate data between corresponding variables while ensuring type safety. GRAFTER's code transplantation succeeds in 94% of the cases, and its fine-grained differential testing can detect more seeded faults than a baseline static cloning bug finder. This result shows GRAFTER's potential to assist developers in catching subtle bugs during copy and paste and to aid developers in comprehending the runtime behavior of nonidentical clones.

## ACKNOWLEDGMENT

Thanks to Todd Millstein and anonymous reviewers for helpful feedback on this paper and research. This work is supported by AFRL grant FA8750-15-2-0075, and NSF grants CCF-1527923 and CCF-1460325.

## REFERENCES


[1] M. Kim, L. Bergman, T. Lau, and D. Notkin, "An ethnographic study of copy and paste programming practices in oopl," in *ISESE '04: Proceedings of the 2004 International Symposium on Empirical Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 83–92.

[2] L. Jiang, Z. Su, and E. Chiu, "Context-based detection of clone-related bugs," in *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT symposium on The foundations of software engineering*. New York, NY, USA: ACM, 2007, pp. 55–64.

[3] 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*, 2004, pp. 289–302.

[4] B. S. Baker, "On finding duplication and near-duplication in large software systems," in *Reverse Engineering, 1995., Proceedings of 2nd Working Conference on*. IEEE, 1995, pp. 86–95.

[5] R. Al-Ekram, C. Kapser, R. Holt, and M. Godfrey, "Cloning by accident: an empirical study of source code cloning across software systems," in *Empirical Software Engineering, 2005. 2005 International Symposium on*, nov. 2005, p. 10 pp.

[6] C. K. Roy and J. R. Cordy, "An empirical study of function clones in open source software," in *Reverse Engineering, 2008. WCRE'08. 15th Working Conference on*. IEEE, 2008, pp. 81–90.

[7] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. R. Engler, "An empirical study of operating system errors." in *SOSP*, 2001, pp. 73–88.

[8] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do code clones matter?" in *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 2009, pp. 485–495.

[9] G. Fischer, "Cognitive view of reuse and redesign," *IEEE Software*, vol. 4, no. 4, p. 60, 1987.

[10] R. Holmes and R. J. Walker, "Supporting the investigation and planning of pragmatic reuse tasks," in *Software Engineering, 2007. ICSE 2007. 29th International Conference on*. IEEE, 2007, pp. 447–457.

[11] T. Zhang, M. Song, J. Pinedo, and M. Kim, "Interactive code review for systematic changes," in *Proceedings of 37th IEEE/ACM International Conference on Software Engineering. IEEE*, 2015.

[12] N. H. Pham, T. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Detection of recurring software vulnerabilities," in *Proceedings of the IEEE/ACM International Conference on Automated software engineering*, ser. ASE '10. New York, NY, USA: ACM, 2010, pp. 447–456. [Online]. Available: http://doi.acm.org/10.1145/1858996.1859089

[13] B. Ray, M. Kim, S. Person, and N. Rungta, "Detecting and characterizing semantic inconsistencies in ported code," in *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, Nov 2013, pp. 367–377.

[14] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A multilinguistic token-based code clone detection system for large scale source code." *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654–670, 2002.

[15] M. Kim, V. Sazawal, D. Notkin, and G. Murphy, "An empirical study of code clone genealogies," in *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*. New York, NY, USA: ACM, 2005, pp. 187–196.

[16] R. Just, "The major mutation framework: Efficient and scalable mutation analysis for java," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM, 2014, pp. 433–436.

[17] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 2009, pp. 364–374.

[18] E. T. Barr, M. Harman, Y. Jia, A. Marginean, and J. Petke, "Automated software transplantation," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015. New York, NY, USA: ACM, 2015, pp. 257–269. [Online]. Available: http://doi.acm.org/10.1145/2771783.2771796

[19] J. Petke, M. Harman, W. B. Langdon, and W. Weimer, "Using genetic improvement and code transplants to specialise a c++ program to a problem class," in *European Conference on Genetic Programming*. Springer, 2014, pp. 137–149.

[20] M. Harman, Y. Jia, and W. B. Langdon, "Babel pidgin: Sbse can grow and graft entirely new functionality into a real world system," in *International Symposium on Search Based Software Engineering*. Springer, 2014, pp. 247–252.

[21] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 96–105.

[22] "Levenshtein distance wikipedia," https://en.wikipedia.org/wiki/Levenshtein_distance.

[23] "Stable marriage problem (smp) wikipedia," https://en.wikipedia.org/wiki/Stable_marriage_problem.

[24] N. Davey, P. Barson, S. Field, R. Frank, and D. Tansley, "The development of a software clone detector," *International Journal of Applied Software Technology*, 1995.

[25] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Science of Computer Programming*, vol. 74, no. 7, pp. 470–495, 2009.

[26] W. E. Howden, "Weak mutation testing and completeness of test sets," *IEEE Transactions on Software Engineering*, no. 4, pp. 371–379, 1982.

[27] M. Woodward and K. Halewood, "From weak to strong, dead or alive? an analysis of some mutation testing issues," in *Software Testing, Verification, and Analysis, 1988., Proceedings of the Second Workshop on*. IEEE, 1988, pp. 152–158.

[28] R. Just, M. D. Ernst, and G. Fraser, "Efficient mutation analysis by propagating and partitioning infected execution states," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM, 2014, pp. 315–326.

[29] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?[software testing]," in *Proc. of ICSE*, 2005, pp. 402–411.

[30] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing," in *Proc. of FSE*, 2014, pp. 654–665.

[31] N. Meng, M. Kim, and K. S. McKinley, "Systematic editing: generating program transformations from an example," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '11. New York, NY, USA: ACM, 2011, pp. 329–342. [Online]. Available: http://doi.acm.org/10.1145/1993498.1993537

[32] R. Cottrell, R. J. Walker, and J. Denzinger, "Semi-automating small-scale source code reuse via structural correspondence," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. ACM, 2008, pp. 214–225.

[33] S. Makady and R. J. Walker, "Validating pragmatic reuse tasks by leveraging existing test suites," *Software: Practice & Experience*, vol. 43, no. 9, pp. 1039–1070, Sep. 2013.

[34] R. Holmes and R. J. Walker, "Systematizing pragmatic software reuse," *ACM Transactions on Software Engineering and Methodology*, vol. 21, no. 4, pp. 20:1–20:44, Nov. 2012.

[35] R. Cottrell, R. J. Walker, and J. Denzinger, "Semi-automating small-scale source code reuse via structural correspondence," in *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2008, pp. 214–225.

[36] D. Lo, L. Jiang, A. Budi *et al.*, "Active refinement of clone anomaly reports," in *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 2012, pp. 397–407.

[37] L. Jiang and Z. Su, "Automatic mining of functionally equivalent code fragments via random testing," in *Proceedings of the eighteenth international symposium on Software testing and analysis*. ACM, 2009, pp. 81–92.

[38] R. Elva and G. T. Leavens, "Semantic clone detection using method ioe-behavior," in *Proceedings of the 6th International Workshop on Software Clones*. IEEE Press, 2012, pp. 80–81.

[39] T. Kamiya, "An execution-semantic and content-and-context-based code-clone detection and analysis," in *Software Clones (IWSC), 2015 IEEE 9th International Workshop on*. IEEE, 2015, pp. 1–7.

[40] M. A. A. Khan, K. A. Schneider, and C. K. Roy, "Active clones: Source code clones at runtime," *Electronic Communications of the EASST*, vol. 63, 2014.

[41] E. Juergens, F. Deissenboeck, and B. Hummel, "Code similarities beyond copy & paste," in *Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on*. IEEE, 2010, pp. 78–87.

[42] W. M. McKeeman, "Differential testing for software," *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107, 1998.
[43] R. Lämmel and W. Schulte, "Controllable combinatorial coverage in grammar-based testing," in *Testing of Communicating Systems*. Springer, 2006, pp. 19–38.
[44] A. Groce, G. Holzmann, and R. Joshi, "Randomized differential testing as a prelude to formal verification," in *Software Engineering, 2007. ICSE 2007. 29th International Conference on*. IEEE, 2007, pp. 621–631.
[45] B. Daniel, D. Dig, K. Garcia, and D. Marinov, "Automated testing of refactoring engines," in *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. New York, NY, USA: ACM, 2007, pp. 185–194.
[46] D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst, "Automatic test factoring for java," in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. ACM, 2005, pp. 114–123.
[47] A. Orso and B. Kennedy, "Selective capture and replay of program executions," in *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 4. ACM, 2005, pp. 1–7.
[48] S. Elbaum, H. N. Chin, M. B. Dwyer, and J. Dokulil, "Carving differential unit test cases from system test cases," in *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*. ACM, 2006, pp. 253–264.
[49] V. P. Ranganath and J. Hatcliff, "Pruning interference and ready dependence for slicing concurrent java programs," in *International Conference on Compiler Construction*. Springer, 2004, pp. 39–56.
[50] T. Xie, K. Taneja, S. Kale, and D. Marinov, "Towards a framework for differential unit testing of object-oriented programs," in *Proceedings of the Second International Workshop on Automation of Software Test*. IEEE Computer Society, 2007, p. 5.
[51] E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun, "Is the cure worse than the disease? overfitting in automated program repair," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 532–543.
[52] S. Sidiroglou-Douskos, E. Lahtinen, F. Long, and M. Rinard, "Automatic error elimination by horizontal code transfer across multiple applications," in *PLDI '15: Proceedings of the 36th ACM SIGPLAN Conference on Programming language design and implementation*, vol. 50, no. 6. ACM, 2015, pp. 43–54.