The Plastic Surgery Hypothesis

Earl T. Barr[†] Yuriy Brun[☞] Premkumar Devanbu[⋆] Mark Harman[†] Federica Sarro[†]

[†]University College London, London, UK
[☞]University of Massachusetts, Amherst, MA, USA
[⋆]University of California Davis, Davis, CA, USA

{e.barr,mark.harman,f.sarro}@ucl.ac.uk, brun@cs.umass.edu, ptdevanbu@ucdavis.edu

ABSTRACT

Recent work on genetic-programming-based approaches to automatic program patching have relied on the insight that the content of new code can often be assembled out of fragments of code that already exist in the code base. This insight has been dubbed the *plastic surgery hypothesis*; successful, well-known automatic repair tools such as GenProg rest on this hypothesis, but it has never been validated. We formalize and validate the plastic surgery hypothesis and empirically measure the extent to which raw material for changes actually already exists in projects. In this paper, we mount a large-scale study of several large Java projects, and examine a history of 15,723 commits to determine the extent to which these commits are *graftable*, i.e., can be reconstituted from existing code, and find an encouraging degree of graftability, surprisingly independent of commit size and type of commit. For example, we find that changes are 43% graftable from the exact version of the software being changed. With a view to investigating the difficulty of finding these grafts, we study the abundance of such grafts in three possible sources: the immediately previous version, prior history, and other projects. We also examine the *contiguity* or chunking of these grafts, and the degree to which grafts can be found in the same file. Our results are quite promising and suggest an optimistic future for automatic program patching methods that search for raw material in already extant code in the project being patched.

Categories and Subject Descriptors: D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement, D.2.13 [Software Engineering]: Reusable Software

General Terms: Experimentation, Languages, Measurement

Keywords: Software graftability, code reuse, empirical software engineering, mining software repositories, automated program repair

1. INTRODUCTION

Software has successfully relieved humans of many tedious tasks, yet many software engineering tasks remain manual, and require significant developer effort. Developers have long sought to automate development tasks. In 2009, the advent of GenProg [41] and Clearview [31] demonstrated automated bug repair. Automatically fixing bugs requires searching a vast space of possible programs, and a key insight that limits that search space is the assumption that fixes often already exist elsewhere in the codebase [2, 40]. This insight arises from the idea that code is locally repetitive, and that the same bug appears in multiple locations, but, when fixed, is not likely to be fixed everywhere. In fact, program source code changes that occur during development can often be constructed from *grafts*, snippets of code located elsewhere in the same program [41]. The act of grafting existing code to construct changes is known as *plastic surgery* [13]. Reformulated as a hypothesis, the insight follows:

The Plastic Surgery Hypothesis: Changes to a codebase contain snippets that *already exist in the codebase at the time* of the change, and these snippets can be *efficiently found* and exploited.

The early success in automating program repair has triggered a dramatic recent upsurge in research on automated repair [2, 9, 19, 24, 29], refactoring [10, 12, 36], and genetic improvement [14, 22, 23, 30, 42]. These approaches have implicitly assumed the correctness of the plastic surgery hypothesis since they rely, in part, on plastic surgery. Despite the fact that a growing body of work depends on it, the plastic surgery hypothesis has not been validated experimentally. The goal of this paper is to validate this hypothesis empirically, on the large scale, on real-world software. Le Goues *et al.* [24] and Nguyen *et al.* [29] considered repetitiveness of changes abstracted to ASTs, and Martínez *et al.* [25] considered changes that could be entirely constructed from existing snippets. Both restricted their search to changes, neglecting primordial, untouched code that was inherited (unchanged) from the first version to the last. Both report the portion of repetitiveness in their datasets, but do not consider the cost of finding it. In this work, we consider both the changes and the primordial code and also explore aspects of the cost of searching in these spaces. In short, our result provides a solid footing to new and ongoing work on automating software development that depends on the plastic surgery hypothesis.

The plastic surgery hypothesis has two parts: 1) the claim that changes are repetitive relative to their *parent*, the program to which they are applied, and 2) the claim that this repetitiveness is usefully exploitable. To address each claim, we focus our inquiry on two questions: "How much of each change to a codebase can be constructed from existing code snippets?" and "What is the cost of finding these snippets?"

To answer the first question, we measure the graftability of each change. The *graftability* of a change is the number of snippets in it that match a snippet in the search space (we clarify the intuitive term "snippets" below). We study over 15,000 human-implemented changes to a program. If the graftability of these changes is high, then this explains part of the success of automated repair, refactoring,

Author order is alphabetical.

FSE '14, November 16–22, 2014, Hong Kong, China

ACM 978-1-4503-3056-5/14/11…$15.00
http://dx.doi.org/10.1145/2635868.2635898.

and genetic improvement and it is an encouraging news for further research in these areas. We consider only line-granular snippets and search for exact matches, ignoring whitespace. We make this choice because 1) developers tend to think in terms of lines, and, 2) practically, this choice reduces the size of the search space with which any tool seeking to help construct changes must contend. Our choice is informed by our practical experience with GenProg, which allows within-line expression granular changes. When we experimented with this setting on a large — hundreds of fairly small buggy programs — dataset, the genetic programming search algorithm almost always bogged down within a few generations because of the search space explosion [7].

To answer the second question, we consider three spaces in which we search for grafts: 1) the parent of a change, the search space of the plastic surgery hypothesis, 2) a change's non-parental ancestors, and 3) the most recent version of entirely different projects. During search, we consider all the lines in each version, and not merely its changes, as this allows us to search those lines that survive unchanged from the start of a version history to its end. This matters when the start of a version history is not empty, as is often the case, since many projects are bootstrapped from prototypes, adapted from existing projects, migrated from one version control system to another, or undergo extensive development outside of version control. In particular, our dataset covers an interval of each project's history that starts from a nonempty change and, on average, these core lines account for 30% of all lines searched. To quantify our answer to the second question, we simply count the number of grafts found in the different search spaces over their size.

We take a pragmatic, *actionability-oriented* view of the plastic surgery hypothesis. We argue that it is not merely about repetitiveness of changes relative to their parent. This fact alone is not actionable, if the cost of finding grafts were prohibitive. The practical success of work on automated repair has demonstrated both the existence of grafts and the tractability of finding them. Thus, the hypothesis is about richness of the first of these search spaces, the parent of a change. We therefore validate it by comparing cost of finding grafts in this search space against the cost of finding them in the other two.

Over the first search space, we find that, on average, changes are 43% graftable, and that 11% of them can be 100% graftable. This suggests that a fair amount of the lines in code changes could be derived from code that already exists in the project. When we compare this result to the other two search spaces, we see that on average the non-parental ancestors contribute only 5% more grafts than the parents, while other projects only provide 9% on average. Moreover, we found that graftability from parent is significantly higher than graftability from both non-parental ancestors and other projects with a high effect size (0.84 and 0.80, respectively). Thus, we can answer the first question, which captures the claim that many donor sites exist at the time of the change in the plastic surgery hypothesis *does* hold (Section 4.1).

An initial answer to the second question is to count the lines searched for each of the search spaces and report the work done to find each donor as the ratio of number of donor sites found to the number of total lines searched (i.e., density). We found that the density of the parent is significantly higher than those of both non-parental ancestors and other projects with a high effect size. Here, again, we see that the plastic surgery hypothesis holds. We found that the cost to search from parent is significantly lower than the cost to search in the other two search-spaces (Section 4.1).

Having validated the plastic surgery hypothesis, we turn our attention to how to exploit it. The success of automated bug fixing, refactoring, and genetic improvement demonstrates the utility of incorporating the search of existing code into techniques seeking to automate software engineering tasks; that is, the consequences of plastic surgery hypothesis are indeed exploitable.

The grafts we have found are mostly single lines (57%), with the distribution following a power law. Thus, grafts would not be considered clones, because the traditional threshold for minimal clone size is 5–7 lines [38]. These smaller repetitive snippets are *micro-clones*. Syntactic repetitiveness below this threshold has simply been considered uninteresting because it lacks sufficient content. 53% of our grafts fall below this threshold and are, therefore, a micro-clones. The success of automated repair, refactoring, and genetic improvement are evidence that these micro-clones, of which our grafts form a subset, are, to the contrary, useful. We contend that micro-clones are the atoms of code construction and therefore are fundamental to code synthesis. Indeed, Gabel and Su demonstrated that line-length micro-clones from a large corpus can reconstruct entire programs [11]. Regardless of the intrinsic interest (or lack thereof) of a particular graft to a human being, such as a trivial assignment statement over the correct variable names, grafts can usefully bootstrap the automation of software development.

To reduce the cost of searching for grafts, we tried to correlate features of changes with graftability (Section 4.3). If we found such a correlation, we could exploit it to search more or less intensively. To this end, we studied if different categories of human-written changes, *e.g.,* bug fixes, refactorings, or new feature additions, are more graftable than others. We also asked if graftability depends on size (Section 4.2). However, we found no such correlations. Indeed, concerning the category of change, the success of automatic bug fixing, refactoring, and genetic improvement suggest that different kinds of changes exhibit the same graftability, as we found.

As a community, we have learned that several lines of code are required for a change to be unique [11] and that a surprising number of changes are redundant, in the sense that they repeat changes already made [25, 29]. We also know that automated repair can be improved by including elements of human-constructed bug fixes [19]. We know that source code is locally repetitive to a greater degree than natural language [16]. To this growing body of knowledge about the repetitiveness of code and its exploitation, we add the validation of the plastic surgery hypothesis, augmented with insights into the proximity of grafts to each other.

The primary contributions of this paper are:

- A formal statement and validation of the plastic surgery hypothesis;
- A large-scale, empirical study of the extent to which development changes can be constructed from code already available during development, i.e., their graftability; and
- An analysis of the relationship between commit features (i.e., size and type) and commit graftability;
- An analysis of the locality of the distribution of grafts in codebase to which a commit applies.

These findings relating to the plastic surgery hypothesis *bode well* for the likelihood of continuing success of patching-by-grafting approaches (including GenProg); they generally suggest that donor grafts to compose a patch can be feasibly found in temporal and spatial proximity to the patch site.

- Donor grafts can often be found in the *current version* of the program to be patched and it is rarely necessary to search earlier versions (Section 4.1).
- The graftable portions of a patch can usually be *composed out of lines from just one contiguous donor graft site*, and very often from no more than two (Section 4.4).
- A significant portion (30%) of donor graft code can be found *in the same file as the patch site* (Section 4.5).

The rest of this paper is structured as follows. Section 2 formally defines the problem we tackle and Section 3 describes our experimental design and data. Section 4 discusses our findings. Section 5 places our work in the context of related research. Finally, Section 6 summarizes our contributions.

2. PROBLEM FORMULATION

We are interested in the graftablity of changes to a codebase with respect to three search spaces: its parents, its non-parental ancestors, and other projects. In this section, we define graftability, its granularity, these three search spaces, and the cost of finding grafts in each of them.

Figure 1 depicts how we measure the graftability of a change. We are interested in the limits of graftability, so we formulate the problem as a thought experiment in which we take the commit as a given, and ask if we can find its pieces in various search spaces, rather than trying to put pieces from a search space together, then ask if they form a commit. We also assume that we can find where a potential graft applies in a target host commit in constant time; this assumption is reasonable in practice, since commits are small, with median range 11–43 lines (Figure 4). The change, shown on the right of Figure 1, is the target host for the grafts. It is cut up into the snippets S_1–S_n. We search the donor codebase for grafts that *exactly* match these snippets. The shaded snippets in the change are graftable, the unshaded snippets are not. We match snippets that are contiguous in both the host and the donor, when possible, as with S_1–S_2. Contiguity holds the promise of reducing the search space (Section 4.4).

Our interest is redundancy in general, not merely the existence of a snippet shared across donor and host; we want to track the abundance of grafts in a donor search space, as this bears directly on the richness of the search space, which we measure using density as defined in Definition 2.2 below. Recall that a multiset generalizes a set to allow elements to repeat. The number of repetitions of an element is its *multiplicity*. For example, in the multiset $\{a,a,a,b,x,y\}$, the multiplicity of a is 3. We use multisets, therefore, to track the abundance of a particular snippet.

We can view the file f as a concatenation of strings, $f = \alpha\beta\gamma$, over some alphabet Σ. *Snippets* are the strings into which any file can be decomposed. Snippets define our unit of granularity; they are the smallest units of interest to us. The smallest a snippet can be is a symbol in Σ; the largest is the file itself. Snippets allow us to treat a file as an ordered multiset of its snippets. We require ordering to impose coordinates that we use to measure distances.

We define a snipper function s that fragments a file into its snippets, and rewrites the whitespace in each snippet to a well-defined normal form (*e.g.,* a single blank). For f defined above, $s(f) = \{\alpha,\beta,\gamma\}$: in other words, s cuts up its input into substrings from which its input can be reconstructed. The details of how s accomplishes this task are unimportant, so we treat s as a black box.

We are now ready to model version-control systems, including ones that facilitate branching and multi-way merges. A *version* V of a project is an ordered multiset of files. Δ models a change, or *commit* in a distributed version control system like `git` or `hg`. For us, each $\Delta : V^k \to V$ is a function that rewrites a k-tuple of versions to produce a new version. When $k > 1$, Δ merges the k parents to form the new version, as when a branch is merged into the master branch in a distributed version control system like `git` or `hg`. Our data is drawn from subversion repositories in which branching is rare and $k = 1$, so we drop k when writing $\Delta(V)$. In addition to files, our snip function s applies to versions, so $s(V)$ is the ordered multiset of the snippets in all the files in V; s also applies to changes, so $s(\Delta)$ is the set of snippets in Δ. Each Δ is a sequence of snippets added, deleted, and modified, where a modification is a paired addition and deletion, as in Myers [27] and in Unix `diff`. When our snipping function s cuts up a commit, it retains only the snippets, producing a multiset, and does not track whether a snippet was added, deleted, or modified, which produces two snippets.

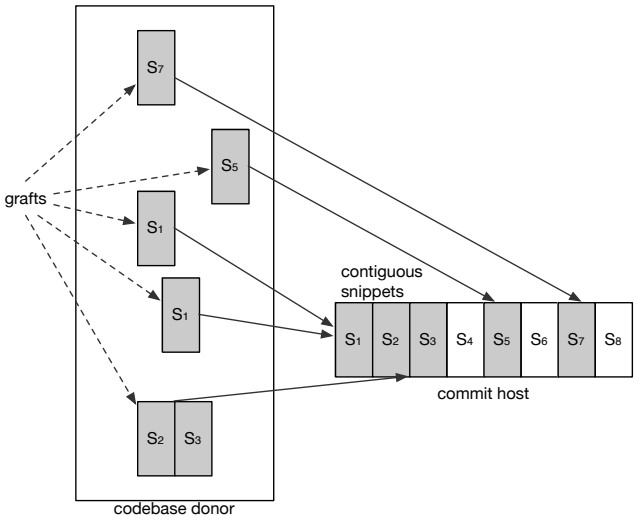


Figure 1: Graftability: We break up a commit into the snippets $S_1,\ldots,S_n$, and search the donor — the codebase — for these snippets. Matches for the snippet in the codebase are grafts (rectangles). A single snippet may have alternate grafts, as with S_1; we try to match snippets that are contiguous in both the donor and the host, as with S_2–S_3. The graftability of the change is the proportion of its snippets we can cover from the donor codebase (shaded grey).

A *version history* is the function composition

$$V_n = \Delta_{n-1}(\Delta_{n-2}(\cdots(\Delta_0(V_0]\,. \tag{1}$$

For clarity, we write this function composition as an alternating sequence of versions V_i and changes Δ_i:

$$V_0\ \Delta_0\ V_1\ \Delta_1\ V_2\ \Delta_2\ \cdots\ \Delta_{n-1}\ V_n. \tag{2}$$

The first version V_0 is special: it can be empty. When $V_0 = \varepsilon$, we have a project's very first commit. Typically, $V_0 = \varepsilon \Rightarrow |\Delta_0| \gg |\Delta_i|,\ i > 0$, because projects often start from a skeleton drawn from another project or inherit code from a prototype. Otherwise, we do not have a project's first commit, but are starting from somewhere within a project's history, as is true in our data set. $V_0 \cap V_n$ is the *core* of a project, those lines that remain unchanged between the first and last commits of a version history, including lines that may have been deleted then re-added in some intermediate versions.

Definition 2.1 (Graftability)**.** The *graftability* of the change Δ against the search space $\mathcal{S}$ is

$$g(\Delta,\mathcal{S}) = \frac{|s(\Delta) \cap s(\mathcal{S})|}{|s(\Delta)|},$$

where $\mathcal{S}$ is an ordered multiset of snippets and $\cap$ is multiset intersection, multiplicity of whose elements is the minimum of its two operands. The graftability of Δ_i against its parent is $g(\Delta_i, V_i)$.

Our notion of graftability measures the partial, not just total, constructibility of changes. Thus, it generalizes previous measures, which focus on 100% graftability. This previous focus was natural,

since such changes are entirely reconstructible. Nonetheless, the existence of changes that are highly, but not completely, graftable, falling into the interval $[0.7..1)$, suggests that the search for grafts is more generally worthwhile than focusing solely on 100% graftable changes, since any nontrivial grafts that are found may considerably constrain the search space, even for novel lines. While it remains to be shown, even lower proportions of graftability may be useful, since a single graft may, in some cases, be highly informative.

Nonparental Ancestral Snippets. The ancestors of the change Δ_j are all the changes Δ_i where $i < j$. Our search spaces consist of snippets, so when searching a version history, our interest is in identifying all the snippets from which a change, in principle, could be constructed. One's parent is, of course, an ancestor, but we already consider this search space; indeed, the plastic surgery hypothesis is couched in terms of a change's parent. Thus, here we are interested only in the snippets that did not survive to a change's parent. This consists of all the snippets in all the ancestors of Δ_j that did not survive to the parent. Thus, we define

$$as(\Delta_j) = \biguplus_{i<j} s(\Delta_i) \setminus s(V_j). \quad (3)$$

Note that a snippet repeatedly added and deleted in a version history has high multiplicity in Equation 3. In practice, $|as(\Delta_j)| \ll |s(V_j)|$ because snippets rarely die, although there are notable exceptions, such Apache's transition in its 2.4 release to handling concurrency via its MultiProcessing Modules, which abandoned many lines.

Search Spaces. Let C be the set of all changes and P be the set of projects. The three search spaces we consider in this paper follow.

$\forall \Delta_i \in C,$

$$\mathcal{S} = s(V_i) \qquad \text{Parent}$$
$$\mathcal{S} = as(\Delta_{i-1}),\ i > 0 \qquad \text{Ancestral lines not in parent}$$
$$\mathcal{S} = \biguplus_{p \in P} s(V^p_{head}) \qquad \text{Other projects}$$

In terms of a version history, the existence component of the plastic surgery hypothesis states $s(\Delta_i) \cap s(V_i) \neq \emptyset$.

Search Cost. To compare the relative richness of these search spaces, we compute their graft density, the number of grafts found in them over their size, averaged over all changes. For the search space $\mathcal{S}$ and the change Δ, let

$$grafts(\mathcal{S}, \Delta) = \{l \in \mathcal{S} \mid \exists k \in s(\Delta) \text{ s.t. } l = k\} \quad (4)$$

be the grafts found in $\mathcal{S}$ for the snippets in Δ. This definition of *grafts* captures the multiplicity in $\mathcal{S}$ of a shared element, with the consequence that $grafts(\mathcal{S}, \Delta) \neq s(\mathcal{S}) \cap s(\Delta)$, since the intersection on the right-hand side computes a set where the multiplicity of each element is the minimum of its two operands.

Definition 2.2 (Search Space Graft Density)**.** The *graft density* of a search space is then

$$gd(\mathcal{S}) = \frac{1}{|C|} \sum_{\Delta \in C} \frac{|grafts(\mathcal{S}, \Delta)|}{|\mathcal{S}|}$$

Graft density is the search space analog of commit graftability. It models the likelihood that a searcher guessing uniformly at random will find a graft for a line in a commit, averaged over all commits. In Section 4.1, we compute and compare the graft density of each of these three search spaces.

Graftability and graft density are the measures we apply to commits and our three search spaces to study the degree to which the plastic-surgery hypothesis applies in a large corpus of versioned repositories of project code.

Project	Description	Commits
`Camel`	Enterprise Integration Framework	1,600
`CXF`	Services Framework	175
`Derby`	Relational Database	820
`Felix`	OSGi R4 Implementation	1,003
`HadoopC`	Common libraries for Hadoop	639
`Hbase`	Distributed Scalable Data Store	3,826
`Hive`	Data Warehouse System for Hadoop	25
`Lucene`	Text Search Engine Library	344
`OpenEJB`	Enterprise Java Beans	534
`OpenJPA`	Java Persistence Framework	84
`Qpid`	Enterprise Messaging system	3,672
`Wicket`	Web Application Framework	3,001

Figure 2: Experimental corpus: 12 Apache projects; `HadoopCommon` is abbreviated as `HadoopC`.

3. EXPERIMENTAL DESIGN

We describe our corpus and present aggregate statistics for its commits, then discuss how we concretely realized our problem formulation for the experiments that follow.

3.1 Corpus

Our corpus contains the 12 software projects listed in Figure 2. All are Java-based, and maintained by Apache Software Foundation. They range in size from 2,712 to 371,186 LOC, from 25 to 3,826 commits, and come from a very diverse range of domains, *e.g.,* service framework, relational database, distributed data storage, messaging system, and web applications.

We mined Apache's git repository[1] to retrieve the change history of the projects from 2004 to 2012. Since Apache uses Subversion and provides only git mirrors, all the changes belong to a single branch. Using git allowed us to access to relevant attributes for each change, such as date, committer identity, source files where the change applies, and so on.

Moreover, since all the projects use the JIRA issue tracking system[2], for each change, we were also able to retrieve the kind of issue (*e.g.,* bug fixing or enhancement commits), its status (*e.g.,* open, closed), and its resolution (*i.e.,* Fixed, Incomplete). Depending on how an organization uses JIRA, a change could represent a software bug, a project task, a help desk ticket, a leave request form, *etc.* By default, JIRA specifies the following five change *types*:

1. *Bug*: A problem which impairs or prevents the functions of the product.
2. *Improvement*: An enhancement to an existing feature.
3. *New Feature*: A new feature of the product.
4. *Task*: A task that needs to be done.
5. *Custom Issue*: A custom issue type, as defined by the organization if required.

The first four types are self-explanatory. The last category groups issues not covered by the other four, but needed by an organization using JIRA. In our dataset, the commits belonging to this set generally concern user wishes, testing code, and sub-tasks.

Each issue has a *status* label that indicates where the issue currently is in its lifecycle, or workflow:

1. `Open`: this issue is ready for the assignee to start work on it.
2. `In Progress`: this issue is being actively worked on at the moment by the assignee.

[1] `http://git.apache.org`.

[2] `https://issues.apache.org/jira/`.

Type	Camel	CXF	Derby	Felix	HadoopC	Hbase	Hive	Lucene	OpenEJB	OpenJPA	Qpid	Wicket
Bug	553	110	626	538	376	2319	15	97	298	28	2102	1855
Improvement	777	57	170	298	160	1053	4	165	82	41	992	839
New Feature	146	0	0	110	31	163	2	54	34	4	252	173
Task	68	3	9	43	4	115	3	11	17	0	115	25
Custom Issue	56	5	15	14	68	176	1	17	103	11	211	109

Figure 3: Count of commit types in our corpus.

Commit Type	Median	Mean	St. Dev.
Bug	11	44.40	156.46
Improvement	43	146.50	289.62
New Feature	16	116.50	359.55
Task	20	76.10	293.69
Custom Issue	37	126.50	197.87

Figure 4: Commit size aggregate statistics.

3. `Resolved`: a resolution has been identified or implemented, and this issue is awaiting verification by the reporter. From here, issues are either Reopened or are Closed.
4. `Reopened`: This issue was once Resolved or Closed, but is now being re-examined.
5. `Closed`: this issue is complete. This means it has been identified, implemented and verified by the reporter.

An issue can be resolved in many ways. The JIRA default resolutions are listed below:

1. `Fixed`: A fix for this issue has been implemented.
2. `Won't Fix`: This issue will not be fixed, *e.g.,* it may no longer be relevant.
3. `Duplicate`: This issue is a duplicate of an existing issue.
4. `Incomplete`: There is not enough information to work on this issue.
5. `Cannot Reproduce`: This issue could not be reproduced at this time, or not enough information was available to reproduce the issue. If more information becomes available, the issue can be reopened.

An issue is initially `Open`, and generally progresses to `Resolved`, then `Closed`. A more complicated life cycle includes an issue whose initial resolution was `Cannot Reproduce` then changed to `Reopened` when the issue becomes reproducible. Such issues can subsequently transition to `In Progress` and, eventually to `Won't Fix`, `Resolved` or `Closed`.

Figure 3 shows the number of changes distinguished per type related to the projects contained in our corpus. We considered only those changes that have been successfully closed, *i.e.,* status=`closed` and resolution=`fixed`. Moreover, we did not consider changes made to non-source code files or containing only comments. As result, we analyzed a total of 15,723 commits. Figure 4 shows the size of the different kinds of commits considered in this study. Note that we did not take into account deleted lines since they are obviously graftable from the parent. We can observe that, on average, *Bug* commits are smaller than all the other types of commits, while *Improvement* commits are the largest.

Figure 5 shows the size of each project's core, the lines that survive untouched from the first version to the last in our version histories for each project. The existence of nonempty first commits is one the reasons for the effectiveness of the plastic surgery hypothesis, which searches these lines in contrast to approaches that focus solely on changes. As Figure 5 shows, the core varies from negligible to dominant at 97% in the case of `Hive`.

3.2 Methodology

We used git to clone and query the histories of the projects in our corpus and extracted the related JIRA information (Section 3.1) into a database. For each project in our corpus, we issued `git reset -hard <commit>` to retrieve a specific version. This command sets the current branch head to `<commit>` modifying index and working tree to match those of `<commit>`. To retrieve a specific change, we issued `git diff` on a commit and its parent and extracted the *commit lines*, *i.e.,* the lines to be grafted. We used the JGit API[3] to automate both tasks.

To realize the snipping function, we wrote a lexer that snips a file into a multiset of code lines, then, from each line, removes the comments and semantically meaningless content, such as whitespace and syntactic delimiters, to normalize the lines. We ran this lexer over each search space, then loaded the resulting multiset into a hash table, whose key is normalized source line (to speed up the search for grafts) and the value is a pair that stores the source line c and its multiplicity. To compute graftability from Definition 2.1 of a commit, we looked up each normalized commit line in the hash table of the commit's parent and divided the number of hits by the number of lines (snippets) in the commit.

4. RESULTS AND DISCUSSION

To open, we validate the plastic surgery hypothesis, the core result of this paper, both its well-known, first claim, the existence of grafts, as well as its heretofore neglected, second claim about that a change's parent is a rich search space for grafts. We then consider features of grafts with the aim of discovering correlations that might help us better decide which software engineering task would benefit most from being augmented with searching an existing codebase. We turn to the question of graft contiguity; that is, for a swath of adjacent lines in the host, can we find an equal sized contiguous donor? If we can, it means we can reconstruct the human-generated patches we are studying more easily, with promising implications for automated construction. We close by considering the distribution of grafts in the donor search space.

4.1 Plastic Surgery Hypothesis

For convenience, we reproduce our central hypothesis:

> **Research Question 1 [The Plastic Surgery Hypothesis]:** Changes to a codebase contain snippets that *already exist in the codebase at the time* of the change, and these snippets can be *efficiently found* and exploited.

Above, by "changes", we mean all commits made, and, by "codebase", we mean the search spaces we defined in Section 2: a com-

[3] `http://www.eclipse.org/jgit/`.

Project	Camel	CXF	Derby	Felix	HadoopC	Hbase	Hive	Lucene	OpenEJB	OpenJPA	Qpid	Wicket	Average
Core	26%	85%	45%	<0.5%	<0.5%	<0.5%	97%	16%	<0.5%	83%	<0.5%	<0.5%	30%

Figure 5: The size of each project's core. The core consists of those lines that are unchanged from the first version to the last in the studied portion of a project's version history.

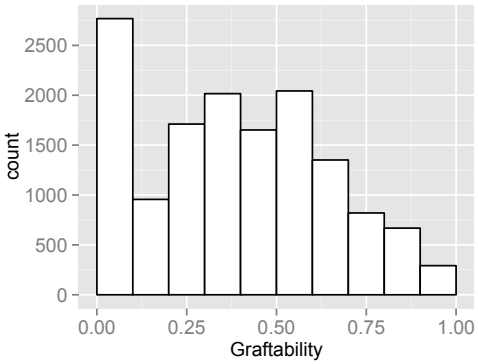


Figure 6: The number of the commits that are x% graftable.

mit's parent V_i, its ancestral lines not in its parent $as(\Delta_i)$ (Equation 3 in Section 2), and the latest versions in our corpus of the other projects. This question explores the limits, or, conversely, the potential, of automatic programming: "How many changes are constituted of snippets that already exist in the code base, or its history, at the time when the commit is applied?"

To answer this question, we analyzed the "graftability" of 15,723 commits coming from a corpus of 12 software projects (Section 3.1). For the commit Δ_i, we model its graftability as shown in Definition 2.1 in Section 2. Nongraftability, or *novelty*, is 1 − graftability. The results immediately prompt us to wonder "How many commits are fully graftable and how many are entirely novel?"

Figure 6 shows the distribution of graftability over the 15,723 commits. We can observe that a large fraction of the commits (42%) are more than 50% graftable. More notably, 10% of the commits can be *completely* reconstructed from grafts. This result aligns with that of Martínez *et al.*, who found that 3–17% of the change in their dataset could be entirely reconstructed [25]. Only 16% of our commits are utterly novel. This data thus clearly suggests confirmation of the first, *"snippets that already exist"* component of the Plastic Surgery Hypothesis.

This finding relates to Gabel and Su, who found very few unique (non-recurring) snippets even of considerable length, in a large (400,000,000 line) corpus of code; however, the mere existence of recurring snippets within this formidably large corpus offers scant hope of feasible graftability [11]. We, however, compute the graftability of commits, not arbitrary snippets of the codebase. Gabel and Su's was a scientific finding, unconcerned with the feasibility of searching for grafts.

The "*efficiently found*" part of the Plastic Surgery Hypothesis is about where to efficiently search; it states the parent's entire codebase (and perhaps just the file where the commit applies, Section 4.5) of the commit, is the best place to search both in terms of richness and cost, in terms of the likelihood of finding a graft in a set of lines in the donor search space. Should we just search the parents and ancestors of the commit to be grafted? Or should we search other projects in the same language? To this end, we address the following research questions:

- RQ1$_a$: How do parents fare as *possible* sources of grafts, when compared to nonparental ancestors and other projects?
- RQ1$_b$: How do parents fare as *efficient* sources of grafts, when compared to nonparental ancestors and other projects?

Figure 7a and Figure 7b show the boxplots of the graftability and density values obtained over the 15,723 commits, when varying the search space from the 3 considered sources: a) commit's parent, b) its ancestral lines not those found in its parent, and c) the latest version of other projects, as defined in Section 2.

Figure 7a bears upon the *existence* of grafts in the three locations. Graftability from parent is much higher than graftability from the nonparental ancestors and than graftability from other projects. This is not that surprising, and at least partially reflects differences in vocabulary (variable names, type names, method names, *etc.*) between projects. Similar inter-project differences were reported in statistical models of code [16]. Code bases tend to monotonically grow in size, so most lines survive from their first introduction to the parent of a given commit. Thus, the nonparental ancestors search space consists of deleted lines. A consequence of the fact that we do not find many changes in the nonparental ancestral lines is that there are not many "regretted deletions": deletions that are later re-added.

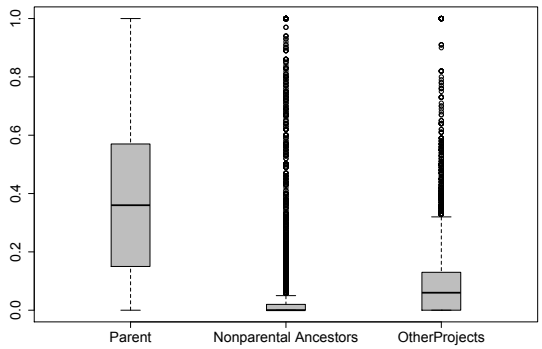
Figure 7b bears upon the *efficiency* of finding grafts in different search spaces, in terms of the the density measure defined in Definition 2.2. We ignore the density figure for non-parental ancestors because (as Figure 7a indicates) they tend to be of low value in graftability. We can observe that density of parent is higher than density from other projects.

Since the boxplots showed no evidence that our samples come from normally distributed populations, we used the Wilcoxon signed-rank test to check for statistical significance. In particular, we tested the following null hypotheses:

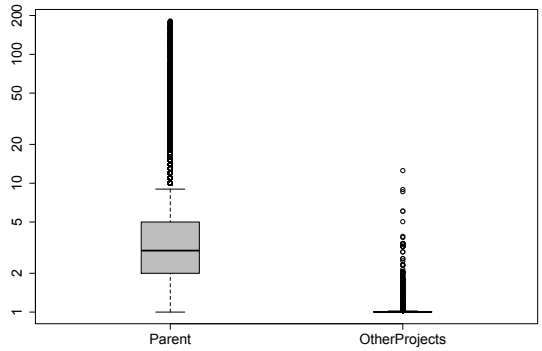
- H0$_a$: There is no significant difference between the graftability from parent and the graftability from nonparental ancestors.
- H0$_b$: There is no significant difference between the graftability from parent and the graftability from other projects.
- H0$_c$: There is no significant difference between the density in parent and the density from other projects.

We set the confidence limit, α, at 0.05 and applied Benjamini-Hochberg [6] correction since multiple hypotheses were tested. To assess whether the effect size is worthy of interest we employed a non-parametric effect size measure, namely the Vargha and Delaney's A_{12} statistic [37]. According to Vargha and Delaney [37] a small, medium, and a large difference between two populations is indicated by A_{12} over 0.56, 0.64, and 0.71, respectively.

The results revealed that there was significant statistical difference ($p < 0.001$) between the graftability from parent and nonparental ancestors in favor of the parent codebase with high effect size ($A_{12} = 0.84$). The Wilcoxon Test also revealed that there was statistical difference between the graftability achieved between parent and other projects codebases in favor of parent with high effect size ($p < 0.001, A_{12} = 0.80$). The Wilcoxon Test between the density of the commit's parent and those of other projects revealed significant statistical difference ($p < 0.001$) in favor of the commit's parent



(a) Graftability of a commit (over 15,723 commits).



(b) Density (log scale) of the search spaces (over 15,723 commits).

Figure 7: Graftability of a commit (a) and cost to search for its grafts (b) as the search space changes from the commit's parent, its ancestors (excluding its parent), and other projects.

with high effect size ($A_{12} = 1$). *We therefore reject the hypotheses that the search spaces are indistinguishable and affirm the Plastic Surgery Hypothesis.*

4.2 Graftability by Commit Size

Next, we consider the fact that commits vary considerably in size. Some are quite small; in fact, about half of all bug fixes are under 10 lines of code. Some commits contain as many as 10,000 lines of code. The question naturally arises, "Is automatic patching only likely to succeed on small patches?" One part of this is the *existence* question: "Do grafts exists only for small patches?". This motivates our second research question:

> **Research Question 2:** How does graftability vary with commit size?

Figure 8 shows the relationship existing between commit graftability and commit size. The plot is a binhex plot, which is essentially a two-dimensional histogram. The x-axis is the size of the commit, and the y-axis is the graftability value for commits of that size. Each hexagon is a "bin" which counts the number of (size, graftability) value pairs that occur within the Euclidean range bounded by that hexagram. The color of the hexagram reflects the count of those pairs that fall within a given bin, lighter colors reflecting a larger count. The figure has some interesting patterns for low values of commit size, which arise from discrete fractions with small denominators and their complements (*e.g.,* $\frac{7}{10}$, $\frac{3}{10}$). But these are just a distraction. The main trend visible in this plot is the absence of one; surprisingly, there appears to be *no relationship* between graftability and commit size — one might rather expect, that as commit size increases, there are more snippets to search for, and thus we might have more difficulty in finding them — thus leading to lower graftability. No such trend is visible.

To confirm this rather strange phenomenon, we estimated a linear regression model, to judge the effect of commit size on graftability. This can be seen as Model 1, in Figure 10. The response variable was graftability, and the predictor variable used in Model 1 was the commit size, log-scaled. The coefficient is very significant ($p < 0.001$), indicating very low probability of observing the data if the true regression coefficient were zero; in other words, we would be very unlikely observe the data if graftability of a commit had no linear dependence on $\log(CommitSize)$. This might seem rather surprising, given that no such dependence is visible in Figure 8. The resolution to this puzzle is clear from the values of R^2 and sample

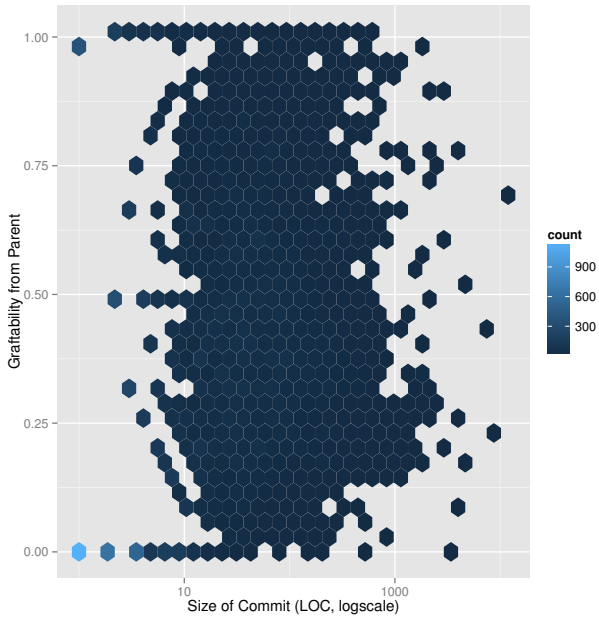


Figure 8: Does commit graftability vary with commits size?

size on the bottom rows of Model 1: just *4%* of the variance in graftability is explained by commit size! In other words, varying the commit size has an extremely weak effect on the variance in graftability; however, even this weak explanatory power is divined as statistically significant by the linear regression, thanks to the large number of samples (15,723). *We conclude that commit size has a negligible effect on the variance in graftability.*

4.3 Graftability by Commit Type

Commits are made for different reasons. As noted in Section 3, the commits in our JIRA data are tagged as one of *Bug*, *Improvement*, *New Feature*, *Task*, or *Custom Issue*. It seems reasonable to wonder whether different types of categories have different graftability. If there were a strong difference, this could tell us which software engineering tasks are most likely to benefit from techniques that rely on the plastic surgery hypothesis. For instance, it seems likely that a *New Feature* commit would be less graftable than one tagged *Bug*, since bugs often appear in multiple locations and may have

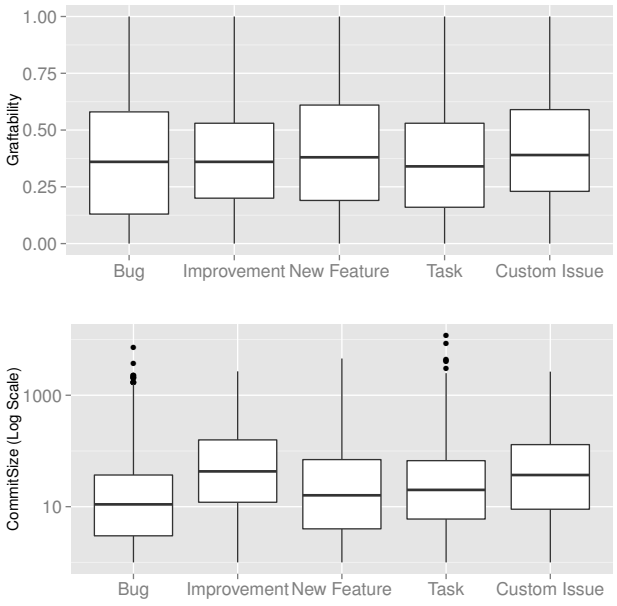


Figure 9: Graftability obtained for five kinds of commits.

already been fixed in some, but not all of those locations. The prior fixes would then be grafts available for fixing the bug of the missed locations. Compare this to *New Feature*, where, especially if *New Feature* is complex, seems more likely to contain novel lines that do not already exist in the system. This leads to the next question.

Research Question 3: Do different kinds of commits exhibit same graftability?

To answer this question, we compared the graftabilty of different types of changes in our dataset (*Bug*, *Improvement*, *New Feature*, *Task*, *Custom Issue*). Figure 9 shows the boxplot (upper plot) of the graftability obtained for the five commit types. The lower plot shows the commit sizes for the different kinds. The lower plot is log-scaled on the y-axis. It is noteworthy that the lower plot shows some differences in the sizes of different types, despite the log-scaling; differences are less visible in the upper plot. To confirm the visual impression that commit types do not affect graftability, we added the type of commit as a factor variable to the regression model discussed earlier, yielding Model 2 in Figure 10.

In this model, the effect of each kind of commit (as a factor) is compared to graftability of the *Bug* commit type, to check if such comparisons have any added explanatory power beyond that provided by commit size, and also to see what that effect might be. In the combined model, variance inflation was well within acceptable limits. This finding echoes that for commit size. While *Improvement* and *Task* commit types are statistically significantly less graftable than the *Bug* commit type, the actual explained effect is very small. The R^2 value essentially remains numerically essentially unchanged. If we consider the commit type by itself as an explanatory factor variable, we can only explain about 0.001 of the variance in graftability (model is omitted). The high significance of this very small effect reported by the regression modeling exercise is simply a result of the large number of samples. Thus, we come to the rather unexpected conclusion that *a commit's type has no significant, practical impact on finding a graft for that commit.*

	Model 1	Model 2
Intercept	0.29***	0.30***
	(0.00)	(0.00)
Commit Size (log scale)	0.07***	0.08***
	(0.00)	(0.00)
Improvement vs. *Bug*		-0.04***
		(0.01)
New Feature vs. *Bug*		0.02
		(0.01)
Task vs. *Bug*		-0.03***
		(0.01)
Custom Issue vs. *Bug*		0.00
		(0.01)
R^2	0.04	0.04
Adj. R^2	0.04	0.04
Number of observations	15,723	15,723

$^{***}p < 0.001$

Figure 10: Size has little effect on graftability, as demonstrated by two regression models with graftability as response: although both models find commit size to be strongly statistically significant with $p < 0.001$, and the standard errors (shown within parentheses) are all small, R^2 shows that these models account for only 4% of the variance in graftability.

4.4 Graft Contiguity

Once a technique has found grafts, it must arrange and order them to transplant them into a host change. Composing grafts, at line granularity, to (re)construct a change even when that change is 100% graftabilty faces a combinatorial explosion of all the permutations of that graft. Novel, nongraftable lines exacerbate the problem. This graft composition search space would be more manageable if grafts were bigger. Intuitively, code decomposes into semantic units that are sometimes bigger than the granularity at which one is searching for grafts. If we could find these units, we could use them to constrain the change (re)construction search space. Thus, we ask how often can we find contiguous grafts of size greater than a single line, in both the host and the donor.

When trying to constitute a commit using snippets that already exist in the code, a natural intuition is that larger chunks will make constituting such a commit easier. At the extremes, searching individual lines in the code that make up a commit would certainly be harder than serendipitously finding all the lines in a commit, altogether in one area of code. We now attempt to formalize this intuition. For this, we refer the reader back to Figure 1. Consider the snippet sequence $S_1 \ldots S_8$. This sequence constitutes the commit. In this, we show how the snippet sequence S_2–S_3 is *contiguous*, in both the donor and in the change host we seek to reconstitute. If this were a common occurrence, the heuristic of search of attempting to constitute commits in groups of lines would be quite effective. When contiguous host snippets are constituted from single or very few donor snippets, the search for donor snippets is simplified and accelerated. This leads the following question:

Research Question 4: To what extent are grafts contiguous?

A commit, in its role as the target host, determines the maximum size of the contiguous region. Contiguous regions of grafts in the donor larger than the largest contiguous region of snippet in the host must necessarily be broken up when transplanted.

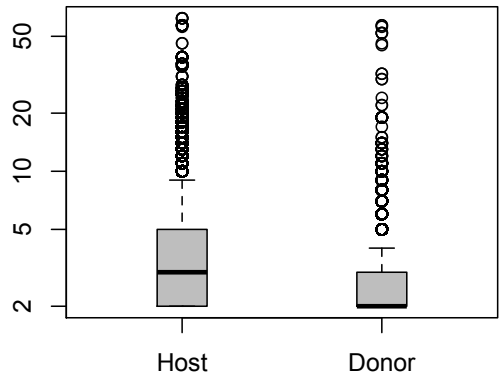


Figure 11: How big are contiguous grafts? The figure reports the size (log scale) of both host and donor snippets.

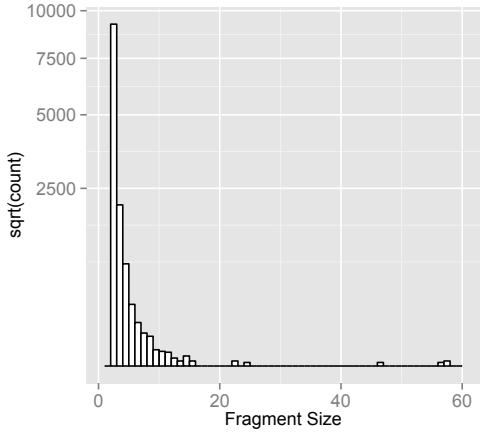


Figure 12: How many host and donor snippets have the same size? The figure shows the number (square root scale) of those host and donor snippets having the same size.

It is very convenient when a contiguous graft in the donor matches a contiguous site in the host: the more often this occurs, the more likely we are to be able to "bite off" larger chunks of code from the donor and shoehorn them to reconstitute substantial pieces of the commit, so we ask

RQ4$_a$: How often do contiguous regions in the donor match contiguous regions in the host? How big are they?

We found 21,726 host snippets and 24,346 donor snippets both consisting of two or more consecutive graftable lines. So we can positively answer the question: continuous graftable regions often appear in both hosts and donors. Figure 11 shows the size of both host and donor snippets. We can observe that the average size of a host snippet (*i.e.,* 4.5 lines) is about twice the one of a donor snippet (*i.e.,* 2.5 lines), this indicates that not all the snippets can be entirely grafted from a single donor and explains the fact that the donor snippets are more than the host snippets. However, in particular, when these continuous regions are *exactly* matched in size, we can simply pluck them out of the donor and paste them into the host: essentially these are little pieces of "micro-clones" that are reproduced in commits.

RQ4$_b$: What is the distribution of host and donor snippets of the same size?

Examining the number of contiguous snippets in both host and donor that have the same size, we found that a host snippet can be grafted from a single donor (*i.e.,* fully matched snippets) in 12,827 cases (53%), while, in the remaining cases more than one donor is needed. Figure 12 shows the number of fully matched snippets grouped by size. We can observe that the majority (72%) of these snippets (9,259) has size 2, while the 16% has size 3, and only 0.6% has size 4, then, as we can observe from the figure, the trend dramatically decreases.

RQ4$_c$: Counting contiguous grafts in the donor as a single site, how many distinct donor sites do we need in order to cover a transplant point in the host?

Since 47% of the donor's snippets does not fully match a host's snippet we are interested in how many donors are needed on average to graft a given host snippet and how difficult is to look for these donors (see next section). We found that 2 donors are needed on average to graft a given host snippet.

The above results revealed that *continuous graftable regions often appear in both hosts and donors. More than a half of host snippets can be grafted from a single donor. In the remaining cases, two donors are needed on average to graft a given host snippet.*

4.5 Graft Clustering

An important factor in the computational feasibility of automatic commit synthesis is the search space size. If one had to search for suitable donors all the time, over the entire possible space of donors (*e.g.,* the entire previous version of the project) it would be much less efficient than if one could just search near the locus of the commit, such as in the same file, the same directory, etc. This motivates the next question

> **Research Question 5:** Are the snippets needed to graft a host snippet in the same file?

Fortunately, we find that 30% of the donor snippets can be found in the same donor file and 9% in the same package. This is an encouraging result, suggesting that *donor snippets are often found in the same file, not requiring more extensive search.*

4.6 Threats to Validity

This section discusses the validity of our study based on three types of threats: construct, internal, and external validity. Construct validity asks whether the measures taken and used in an experiment actually measure phenomenon under study. Internal validity concerns the soundness of the methodology employed to construct the experiment, while external validity concerns the bias of choice of experimental subjects.

Section 3.1 describes how we automatically computed our measures. To mitigate the threat of an implementation error, we applied unit testing and one of the authors manually verified the accuracy of the measurements of 30 commits selected uniformly. To address internal validity, we carefully verified that our data met all required assumptions before applying statistical tests and the regression model. Moreover, our 100% graftability results are low relative to the standard finding of 5–30% redundancy in the clone literature. This is probably due to our choice of considering exact matches over whitespace normalized, but otherwise untouched and notably unobstructed source lines. We adopted this choice because abstraction reduces the semantic content of lines, such as that contained in identifiers, which must then be restored. Thus, we choose exact matching because of our belief that these lines would be strictly more useful for techniques relying on the plastic surgery hypothesis.

Relaxed notions of matching are a distinct and interesting avenue of research that has witnessed positive results [33]. As for the external validity, the projects in our corpus are all open source Apache projects. Although they differ in domain and size (Section 3.1), we cannot claim that our findings generalize to other software systems. However, we have formally stated the problem (Section 2) and described our methodology (Section 3.2) to facilitate the replication of our study.

5. RELATED WORK

It has been known for some time that the production code contains software clones [3, 4, 8, 18]. These can be verbatim cut-and-paste copies of code (so-called Type 1 clones) or might arise from a more elaborate cut-modify-and-paste process (so-called Type 2 and Type 3 clones [8]). The presence of code clones has led to much work on techniques for investigating and evaluating this form of apparent redundancy [5, 38].

More recently, authors have sought to *measure* the degree of redundancy in code and the commits applied to it. Gabel and Su [11] sought to answer the question "How unique (and how repetitive) is source code?" in an analysis of approximately 420M SLoC from 6,000 open-source software projects. They observed significant syntactic redundancy at levels of granularity from 6–40 tokens. For example, at the level of granularity with 6 tokens, the projects were between 50% and 100% redundant. This suggests that code contains a great deal of "redundancy" that could potentially be exploited by code improvement techniques. However, Gabel and Su did not consider code commits, nor the cost of finding redundancy.

Nguyen *et al.* sought to answer the question "How often do developers make the same changes (commits) they have made before?", studying 1.8M revisions to 2,841 open-source projects [28]. They defined a "repeated change" to be an AST extracted from a change that matches an AST from a previous change in some project's history, including the same project. Over ASTs, they found that changes are frequently repeated, with the median 1-line change having a 72% chance of having been performed elsewhere. The repetitiveness dropped exponentially as the granularity (number of lines) increased: For 6-line and greater granularity, it was typically below 10%. This "commit redundancy" meant that future changes could be recommended with over 30% accuracy.

Martínez *et al.* [25] also recently studied commit-redundancy, focusing their attention on commits they term "temporally redundant", or, in our terminology, 100% graftable changes. 100% graftable lines are interesting because they are, in principle, entirely reconstructible. Our measure of graftability in Definition 2.1 additionally measures the degree to which a commit is graftable. Like Nguyen *et al.* [28] and Gabel and Su [11], they find a perhaps surprisingly high degree of redundancy in the code they studied.

Our work has two primary methodological differences to this previous work on commit redundancy [25, 28]: we consider the cost of finding a graft, which the previous work does not, and we are concerned with code-commit graftability rather than just 'commit-commit' redundancy, a special case of graftability. That is, we consider the full code space in assessing graftability, whereas previous work focussed on commits alone. In terms of our formalism, both Nguyen *et al.* and Martínez *et al.* search the set of changes or deltas; in contrast, we focus on versions, and therefore search the lines in $V_0 \cap V_n$; the project's 'core' lines, which can dominate a version history, accounting up to 97% of the lines in the final version, as we show in Section 3.1. If we had a full commit history starting, *ab initio*, with the empty system, then we could assume that 'commit-commit' and 'code-commit' approaches would study largely the same information. However, since version histories typically do not go back this far, that assumption is invalid; the current version of a system is not merely the product of the sequence of commits for which information is available.

Assessing the degree of code-commit graftability has implications for work on program improvement, an area that is witnessing a significant upsurge in interest. Program improvement seeks to automate improving an existing code base with relatively small modifications. Examples include repairing the code base [17, 24, 39, 41], enhancing its properties [22,23,30,34,42,43], or even migrating it to other systems [21, 33]. All these program improvement approaches share the foundational assumption that many software systems contain the seeds of their own improvement. They are united by the way they search for, extract and recombine fragments of code to create desired new functionalities and behaviors. In order to assess the potential of this search space, we need to study not just commit-commit redundancy, but also code-commit graftability. By assessing code-commit graftability, we seek to shed light on the *degree* to which a commit made by a human could have been found by a machine and the *cost* of so-doing. Our results on the degree and cost of graftability or within-system code commits are relevant to program improvement work that searches for modifications in existing system [1, 22, 23, 30, 42], or which finds patches from elsewhere in the system [24, 41]. Our results on graftability between systems are relevant to program improvement work that searches for transplants from one system to another [15, 32]. Overall, our results provide further evidence to support claims that it is promising to mine human commits for patterns, templates, and code fragments that can be reused to improve systems. This is a technique to which other authors have recently turned for automated program improvement [20, 39] and for semi-automated improvement as decision support to software engineers [26, 35], with promising results.

6. CONTRIBUTIONS

In this paper, we validated the plastic surgery hypothesis with a large scale experiment over 15,723 commits from 12 Java projects. Our core finding is that the parent (rather than non-parental ancestors, or other projects) of a commit is by far the most fecund source of grafts. We also find encouraging evidence that many commits are *graftable*: they can be reconstituted. We also find that grafts are often contiguous, which suggests heuristics that attempt to graft commits out of multiple contiguous lines. Finally, we find that fully 30% of the elements of commits can be found within the same file. These are encouraging results for automatic program repair techniques that exploit the plastic surgery hypothesis.

It is also true that there are fragments of commits that are not graftable. The complement of graftability measures the novelty of changes. As future work, we intend to explore if the feature set of novel changes is more predictable than we have found grafts to be, again with the aim of identifying which sorts of changes are most likely to profit from the plastic surgery hypothesis.

7. ACKNOWLEDGEMENTS

The research is part funded by the Engineering and Physical Sciences Research Council CREST Platform Grant (EP/G060525), the Dynamic Adaptive Automated Software Engineering (DAASE) programme grant (EP/J017515), and the National Science Foundation under Grants No. CCF-1247280 and CCF-1446683

8. REFERENCES

[1] Andrea Arcuri, David Robert White, John A. Clark, and Xin Yao. Multi-objective improvement of software using co-evolution and smart seeding. In 7^{th} *International Conference on Simulated Evolution and Learning (SEAL 2008)*, pages 61–70, Melbourne, Australia, December 2008. Springer.

[2] Andrea Arcuri and Xin Yao. A novel co-evolutionary approach to automatic software bug fixing. In *Proceedings of the IEEE Congress on Evolutionary Computation (CEC'08)*, pages 162–168, Hongkong, China, June 2008.

[3] Brenda S Baker. A program for identifying duplicated code. In *Computer Science and Statistics 24: Proceedings of the 24th Symposium on the Interface*, pages 49–49, 1993.

[4] Ira D. Baxter, Andrew Yahin, Leonardo Mendonça de Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *International Conference on Software Maintenance (ICSE'98)*, pages 368–377, 1998.

[5] Stefan Bellon, Rainer Koschke, Giuliano Antoniol, Jens Krinke, and Ettore Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering*, 33(9):577–591, 2007.

[6] Yoav Benjamini and Yosef Hochberg. Controlling the False Discovery Rate: A Practical and Powerful Approach to Multiple Testing. *Journal of the Royal Statistical Society. Series B (Methodological)*, 57(1):289–300, 1995.

[7] Yuriy Brun, Earl Barr, Ming Xiao, Claire Le Goues, and Prem Devanbu. Evolution vs. intelligent design in program patching. Technical Report `https://escholarship.org/uc/item/3z8926ks`, UC Davis: College of Engineering, 2013.

[8] S Carter, R. Frank, and D.S.W. Tansley. Clone detection in telecommunications software systems: A neural net approach. In *Proc. Int. Workshop on Application of Neural Networks to Telecommunications*, pages 273–287, 1993.

[9] Satish Chandra, Emina Torlak, Shaon Barman, and Rastislav Bodik. Angelic debugging. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 121–130, Honolulu, HI, USA, 2011. ACM.

[10] Marios Fokaefs, Nikolaos Tsantalis, Eleni Stroulia, and Alexander Chatzigeorgiou. Identification and application of extract class refactorings in object-oriented systems. *Journal of Systems and Software*, 85(10):2241 – 2260, 2012.

[11] Mark Gabel and Zhendong Su. A study of the uniqueness of source code. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, FSE '10, pages 147–156. ACM, 2010.

[12] Ah-Rim Han and Doo-Hwan Bae. Dynamic profiling-based approach to identifying cost-effective refactorings. *Information and Software Technology*, 55(6):966 – 985, 2013.

[13] Mark Harman. Automated patching techniques: The fix is in: Technical perspective. *Communications of the ACM*, 53(5):108, 2010.

[14] Mark Harman, William B. Langdon, Yue Jia, David Robert White, Andrea Arcuri, and John A. Clark. The GISMOE challenge: Constructing the pareto program surface using genetic programming to find better programs (keynote paper). In 27^{th} *IEEE/ACM International Conference on Automated Software Engineering (ASE 2012)*, pages 1–14, Essen, Germany, September 2012.

[15] Mark Harman, William B. Langdon, and Westley Weimer. Genetic programming for reverse engineering (keynote paper). In Rocco Oliveto and Romain Robbes, editors, 20^{th} *Working Conference on Reverse Engineering (WCRE 2013)*, Koblenz, Germany, 14-17 October 2013. IEEE.

[16] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 837–847. IEEE, 2012.

[17] Guoliang Jin, Wei Zhang, Dongdong Deng, Ben Liblit, and Shan Lu. Automated concurrency-bug fixing. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 221–236, 2012.

[18] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. CCFinder: A multi-linguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(6):654–670, 2002.

[19] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In 35^{th} *International Conference on Software Engineering (ICSE'13)*, pages 802–811. IEEE / ACM, 2013.

[20] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 802–811, 2013.

[21] William B. Langdon and Mark Harman. Evolving a CUDA kernel from an nVidia template. In *IEEE Congress on Evolutionary Computation*, pages 1–8. IEEE, 2010.

[22] William B. Langdon and Mark Harman. Genetically improved CUDA C++ software. In 17^{th} *European Conference on Genetic Programming (EuroGP)*, Granada, Spain, April 2014. To Appear.

[23] William B. Langdon and Mark Harman. Optimising existing software with genetic programming. *IEEE Transactions on Evolutionary Computation*, 2014. To appear.

[24] Claire Le Goues, Stephanie Forrest, and Westley Weimer. Current challenges in automatic software repair. *Software Quality Journal*, 21(3):421–443, 2013.

[25] Matias Martinez, Westley Weimer, and Martin Monperrus. Do the fix ingredients already exist? An empirical inquiry into the redundancy assumptions of program repair approaches. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, pages 492–495, New York, NY, USA, 2014. ACM.

[26] Na Meng, Miryung Kim, and Kathryn S. McKinley. LASE: locating and applying systematic edits by learning from examples. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 502–511, 2013.

[27] Eugene W. Myers. An O(ND) difference algorithm and its variations. *Algorithmica*, 1:251–266, 1986.

[28] Hoan Anh Nguyen, Anh Tuan Nguyen, Tung Thanh Nguyen, T.N. Nguyen, and H. Rajan. A study of repetitiveness of code changes in software evolution. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 180–190, Nov 2013.

[29] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. SemFix: Program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 772–781, San Francisco, CA, USA, 2013. IEEE Press.

[30] Michael Orlov and Moshe Sipper. Flight of the FINCH through the java wilderness. *IEEE Transactions Evolutionary Computation*, 15(2):166–182, 2011.

[31] Jeff H. Perkins, Sunghun Kim, Sam Larsen, Saman Amaras-

inghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin Rinard. Automatically patching errors in deployed software. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, pages 87–102, Big Sky, MT, USA, October 12–14, 2009.

[32] Justyna Petke, Mark Harman, William B. Langdon, and Westley Weimer. Using genetic improvement & code transplants to specialise a C++ program to a problem class. In 17^{th} *European Conference on Genetic Programming (EuroGP)*, Granada, Spain, April 2014. To Appear.

[33] Baishakhi Ray and Miryung Kim. A case study of cross-system porting in forked projects. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 53:1–53:11, New York, NY, USA, 2012. ACM.

[34] Pitchaya Sitthi-amorn, Nicholas Modly, Westley Weimer, and Jason Lawrence. Genetic programming for shader simplification. *ACM Trans. Graph*, 30(6):152:1–152:11, 2011.

[35] Sooel Son, Kathryn S. Mckinley, and Vitaly Shmatikov. Fix me up: Repairing access-control bugs in web applications. In *In Network and Distributed System Security Symposium*, 2013.

[36] Nikolaos Tsantalis and Alexander Chatzigeorgiou. Identification of move method refactoring opportunities. *IEEE Trans. Softw. Eng.*, 35(3):347–367, May 2009.

[37] András Vargha and Harold D. Delaney. A critique and improvement of the CL common language effect size statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000.

[38] Tiantian Wang, Mark Harman, Yue Jia, and Jens Krinke. Searching for better configurations: a rigorous approach to clone evaluation. In *European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13*, pages 455–465, Saint Petersburg, Russian Federation, August 2013. ACM.

[39] Yi Wei, Yu Pei, Carlo A. Furia, Lucas Serpa Silva, Stefan Buchholz, Bertrand Meyer, and Andreas Zeller. Automated fixing of programs with contracts. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, pages 61–72, 2010.

[40] Westley Weimer. Patches as better bug reports. In *Generative Programming and Component Engineering*, pages 181–190, 2006.

[41] Westley Weimer, Thanh Vu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *International Conference on Software Engineering (ICSE)*, pages 364–374, Vancouver, Canada, 2009.

[42] David Robert White, Andrea Arcuri, and John A. Clark. Evolutionary improvement of programs. *IEEE Transactions on Evolutionary Computation*, 15(4):515–538, 2011.

[43] David Robert White, John Clark, Jeremy Jacob, and Simon Poulding. Searching for resource-efficient programs: Low-power pseudorandom number generators. In *2008 Genetic and Evolutionary Computation Conference (GECCO 2008)*, pages 1775–1782, Atlanta, USA, July 2008. ACM Press.