The Care and Feeding of Wild-Caught Mutants

David Bingham Brown
Univ. of Wisconsin–Madison, USA
bingham@cs.wisc.edu

Michael Vaughn
Univ. of Wisconsin–Madison, USA
mvaughn@cs.wisc.edu

Ben Liblit
Univ. of Wisconsin–Madison, USA
liblit@cs.wisc.edu

Thomas Reps
Univ. of Wisconsin–Madison and GrammaTech, Inc., USA
reps@cs.wisc.edu

ABSTRACT

Mutation testing of a test suite and a program provides a way to measure the quality of the test suite. In essence, mutation testing is a form of sensitivity testing: by running mutated versions of the program against the test suite, mutation testing measures the suite's sensitivity for detecting bugs that a programmer might introduce into the program. This paper introduces a technique to improve mutation testing that we call *wild-caught mutants*; it provides a method for creating potential faults that are more closely coupled with changes made by actual programmers. This technique allows the mutation tester to have more certainty that the test suite is sensitive to the kind of changes that have been observed to have been made by programmers in real-world cases.

CCS CONCEPTS

• **Software and its engineering** → **Software configuration management and version control systems**; **Software testing and debugging**; Parsers;

KEYWORDS

mutation testing, repository mining, test suites

ACM Reference format:
David Bingham Brown, Michael Vaughn, Ben Liblit, and Thomas Reps. 2017. The Care and Feeding of Wild-Caught Mutants. In *Proceedings of ESEC/FSE'17, Paderborn, Germany, September 4–8, 2017,* 12 pages.
https://doi.org/10.1145/3106237.3106280

1 INTRODUCTION

Good test suites are among the most important tools available to ensure the quality of software. However, bad test suites help nobody, and evaluating test suites themselves is challenging. Mutation testing of a test suite with respect to a program provides one way to measure the test suite's quality. In essence, mutation testing measures the *sensitivity* of the test suite, which is intended to provide an estimate of the ability of the test suite to detect faults inserted into the program in the future [7, 10]. Conventional mutation-testing approaches make random (or effectively random) modifications to the target program's code according to some fixed set of substitution directives, such as replacing ">" with ">=". Just et al. [12] have shown that this strategy is a useful proxy for real faults.

However, the conventional approach to mutation testing has a basic limitation: the *ad hoc* patterns used do not necessarily reflect the types of changes made to source code by human programmers. Consequently, the measured sensitivity does not necessarily reflect how effective the test suite is at identifying the kinds of defects that real programmers might introduce.

The objective of our research has been to reexamine mutation testing by using mutation operators that more closely resemble defects introduced by real programmers. Thus, the high-level goal of our work is as follows:

> Find a method for creating potential faults that are closely coupled with defects created by actual programmers.

We have developed a method for identifying such mutation operators by using the revision histories of software projects. We call such mutants *wild-caught mutants*.

When interpreting a revision history, it may be difficult to determine precisely when a defect was introduced. For this reason, we use instead the *reversal* of what is likely to be a *correction* in the revision history. That is, the orientation of a mutation operator that we recover is *backward* with respect to the direction of the patch from which it was recovered in the revision history. From a patch of the form "before-code → after-code," we create a mutation operator "$\text{pattern}_A \Rightarrow \text{replacement}_B$," where pattern_A is a pattern created from code fragment "after-code," and replacement_B is a rewrite created from code fragment "before-code."[1]

After we present the details of our method for extracting such mutation operators, there are a number of natural research questions that we consider. For starters, we wish to know whether wild-caught mutation operators subsume the manually curated mutation operators widely used until now:

> Research Question 1: Does the the operator-harvesting method of the wild-caught-mutants technique find existing mutation operators?

Conversely, perhaps wild-caught mutants exhibit useful qualities that go beyond past work:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ESEC/FSE'17, September 4–8, 2017, Paderborn, Germany
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5105-8/17/09…$15.00
https://doi.org/10.1145/3106237.3106280

[1] While a patch "before-code → after-code" could introduce a defect—and hence our recovered mutation operator would represent a correction—our system confines itself to small and typically single-line patches, which one might expect to be corrections more often than defect introductions.

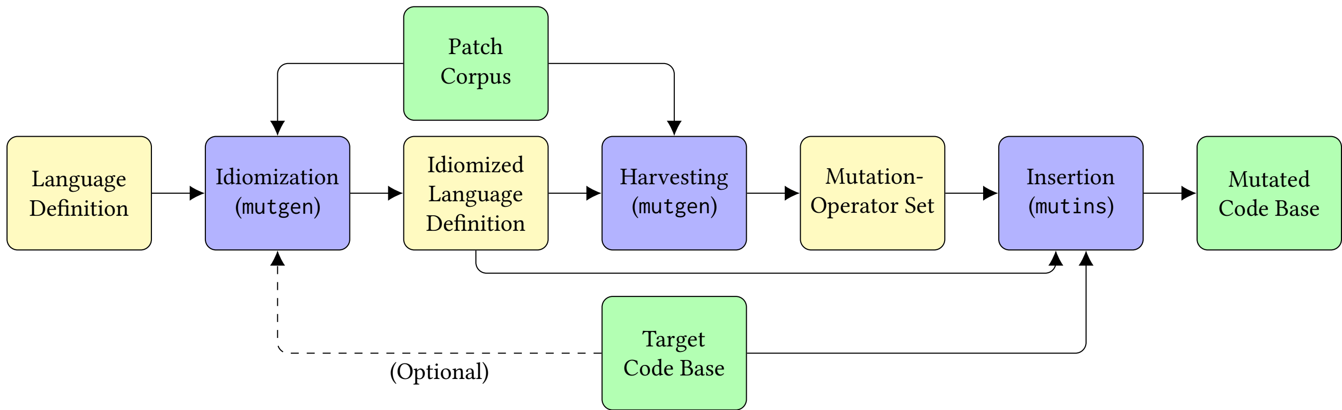

Figure 1: Overview of mutation-operator extraction and insertion through the `mutgen`/`mutins` toolchain

> Research Question 2: Does the operator-harvesting method of the wild-caught-mutants technique find operators that are *not* existing mutation operators?

We also want to know whether our approach leads to improved mutation testing:

> Research Question 3: Do wild-caught mutants exhibit behavior that is quantifiably different than existing mutation operators—and, if so, in what ways?

While backward patches seem more likely to (re)introduce bugs, which is good from the standpoint of mutation testing, forward patches may also describe interesting human-generated changes.

> Research Question 4: Does harvesting from forward patches yield many additional mutation operators, and do the behaviors of these new operators differ significantly from those harvested from backward patches?

The contributions of our work can be summarized as follows:

We describe a technique to automatically create mutation operators that others have identified as being missing from previous mutation approaches (Just et al. [12]).

We created a toolchain for mutation-operator extraction that implements the wild-caught-mutants technique. This toolchain allows the user to harvest mutation operators from most common programming languages (using `mutgen`) and then apply them to a system (using `mutins`) to perform mutation testing of a test suite.

We report on experiments in which we extracted mutation operators from a corpus consisting of the 50 most-forked C-based projects on GitHub. We find that wild-caught mutants can capture faults that traditional mutation operators are unable to reproduce. Compared to existing mutation operators, the mutation operators obtained by the wild-caught-mutants technique lead to mutants that roughly as hard to "kill" as mutants from traditional mutation operators. However, they offer a richer variety of changes, and thereby provide a more extensive way to evaluate the quality of a test suite. Harvesting from forward patches provides a significant number of operators not obtained from backward patches. There is some support for the conjecture that, compared to forward-harvested operators, backward-harvested operators can introduce defects that more closely resemble defects introduced by real programmers.

Organization. The remainder of the paper is organized as follows: Section 2 offers an overview of our approach, then describes mutation-operator extraction and mutant insertion in detail. Section 3 presents our experimental setup, followed by experimental results in Section 4. Section 5 considers threats to the validity of our approach. Section 6 discusses related work. Section 7 describes supporting materials that are intended to help others build on our work. Section 8 concludes.

2 HARVESTING AND INSERTION

As shown in Figure 1, our system consists of two tools: `mutgen`, for extracting reusable mutation operators, and `mutins`, for applying these operators to the code of a system under test. The extraction process—referred to as *harvesting*—generates a *mutation-operator set* from a corpus of `diff`-formatted code patches. Our insertion tool, `mutins`, can then apply these mutation operators to a new code base distinct from that used during harvesting. The main input to `mutgen` is a corpus of patches for harvesting; the main input to `mutins` is a target code base to mutate. Both tools are parameterized by a second input, a *language definition*, which specifies the syntactic elements of the language on which they operate (see Section 2.1). In other words, our implementation of the wild-caught-mutants approach is really a framework that can be retargeted easily to work on other languages.

Our toolchain operates in three phases: idiomization, harvesting, and insertion. *Idiomization* augments the language definition to conform more closely with the patch corpus or system under test. This preprocessing step is performed by `mutgen` and is described in Section 2.2. *Harvesting* extracts novel mutation operators from the patch corpus. This process is also performed by `mutgen` and is described in Sections 2.3, 2.4 and 2.6. The final *insertion* step applies harvested mutation operators to the system under test. It is implemented by `mutins` and described in Section 2.7.

```
K auto break case char const continue default
K do double else enum extern float for goto
K if inline int long register restrict return
K short signed sizeof static struct switch
K typedef union unsigned void volatile while
O = += -= *= /= %= &= |= ^= <<= >>= ++ --
O + - * / % ~ & | ^ << >> ! && ||
O == != < > <= >= [ ] -> . ( ) , ? :
Q ' "
C /* */
c //
```

Figure 2: Language-definition file for C. The first character on each line specifies keywords (K), operators (O), quoted string literals (Q), block comments (C), or single-line comments (c).

2.1 Language Definition

We define a language as a set of operators, keywords, quote delimiters, and comments (both block comments and single-line comments). Our language parser is essentially a lexical analyzer. However, our language-definition files are simpler than those required by a full lexical-analyzer generator (e.g., `flex`), in part because we do not need to feed tokens into a full compiler. Additional simplification is possible by leveraging commonalities seen in the basic syntax of C and other C-influenced languages, such as C++, Java, and C#. These commonalities allow us to recognize tokens with high accuracy using the following strategy:

- Operators are consumed greedily from the input stream.
- Text from one quote delimiter to a matching quote delimiter is parsed as a single string literal.[2]
- Text identified to be not part of an operator is read until whitespace or an operator is encountered; once isolated in this fashion, the text is classified as follows:
  - If the text exists in the keyword list, it is classified as a keyword.
  - If the text begins with a digit, it is classified as a numeric literal.
  - Otherwise, it is classified as an identifier.

While simple, these rules are such that `mutgen` can effectively parse most languages that lack semantic whitespace.[3]

Using such a simple language definition—and not, e.g., a context-free grammar for the language used in the corpus—supports our goal of using patch histories as a source of mutation-operator sets. The patches processed by `mutgen` to harvest mutation-operator sets have varying and unpredictable contexts: one may easily encounter a patch that begins or ends mid-expression or mid-comment. Therefore, we do not parse the input with respect to a context-free grammar for the language of the corpus. Instead, we perform purely lexical analysis, generating a stream of tokens as determined by rules in the language definition. This approach also allows our system to be more flexible with regard to its inputs. The harvester need not be able to compile any part of its input corpus; it can injest incomplete or invalid code fragments as well as complete code.

Comments create a challenge during harvesting. The patches we process can begin and/or end mid-comment. Thus, we cannot guarantee that the tokens generated when analyzing a particular patch correspond to actual code, as opposed to a natural-language comment. We address this problem in both the extraction and insertion phases. During extraction, we use heuristics to identify (and discard) patches that are likely to be comments; Section 2.4 discusses these heuristics in more detail. During insertion, we have the full system under test—and therefore the complete context for any potential insertion—so we can identify comments precisely and exclude them from mutant insertion.

[2]Escaped quote delimiters are not handled, but would not be hard to add.

[3]Python's semantically meaningful whitespace could be handled as well by materializing and later dissolving explicit indent/outdent tokens: a standard Python lexing/parsing technique.

Table 1: Most frequent inferred idioms in our experimental corpus

Idiom	Incidence	Idiom	Incidence	Idiom	Incidence
`0`	1,350,306	`2`	489,152	`s`	294,490
`0x00`	984,949	`u32`	386,045	`line`	227,925
`dev`	695,548	`y`	379,709	`data`	227,376
`1`	578,986	`u8`	311,762	`inode`	216,908
`set`	491,888	`file`	310,500	`o`	215,515

2.2 Idiomization

The language-definition file covers all of a language's keywords and operators. However, some identifiers are used so often, and in such standardized ways, as to effectively be additional keywords. We call these identifiers *idioms* and the process of identifying them *idiomization*. We collect identifiers that occur within the corpus above a user-selected threshold. This threshold can be specified by minimum incidence in the corpus, minimum frequency, or a "top-*k*" limit of accepted idioms. Subsequent extraction passes treat these identifiers as additional keywords.

For example, `NULL` is missing from the C language definition given in Figure 2. This omission is correct: `NULL` is a standard C macro but is not a C keyword *per se*. Adding `NULL` to this definition by hand would be easy, and would expand the pool of admissible candidate mutation operators. Unfortunately, the arbitrary choices required by this method do not necessarily scale. Automated idiomization provides a pragmatic method to augment a base language definition with identifiers that are used idiomatically in practice.

Depending on differences in the subject of the patch corpus and the system under test, idiomization has the potential to identify idiomatic keywords that appear rarely or never in the system under test. Therefore, we make idiomization optional, and also allow the system under test to be used as its own source of idiomization.

Table 1 lists some of the most commonly identified idioms derived from our experimental corpus. The influence of the Linux kernel is apparent in several entries.

```
- if (x)
+ if (x && y)
```

(a) Admissible candidate

```
- :if .( $1 .)
+ :if .( $1 .&& $_ .)
```

(b) Mutation operator extracted from Figure 3a

```
- if (x && y)
+ if (x)
```

(c) Inadmissible candidate: requires synthesizing "y"

```
- if (x > 0)
+ if (x > 1)
```

(d) Candidate made admissible by idiomization of "0"

Figure 3: Example candidate mutation operators.

2.3 Syntactic Mutation

Mutgen identifies candidate mutation operators by isolating small changes (defined as having fewer than a configurable number of lexical tokens) from the revision history it reads as input. For a patch to be considered for extraction, it must contain a contiguous section of removed and replaced code that we divide into a "before" and "after" block. A single patch (that is, a single diff-formatted file) can contain multiple blocks of modified code, and each individual contiguous block is treated as a separate candidate mutation operator.

Corresponding blocks of each identified section are broken into a stream of tokens as described by the language definition (see Section 2.1). Mutgen makes no attempt to understand the underlying semantics or grammar of a processed language.

Mutins does not attempt synthesis of identifiers or literals, so mutgen requires that candidate mutation operators not require the synthesis of new information. In particular, it must be possible to assemble the *before* state solely from identifiers and literals matched in the *after* state, along with any keywords drawn from the idiomization-enhanced language definition. Once the *before* and *after* blocks are tokenized, mutgen then analyzes both to determine whether this requirement is satisfied. A candidate mutant that meets the requirement is called an *admissible candidate*. Figure 3a shows an admissible candidate mutation operator: building the *before* text requires no new identifiers or literals beyond those that appeared in the corresponding *after* text.

Conversely, an *inadmissible* candidate is one that would require synthesis of new information to turn its *after* state back into its *before* state. The candidate mutation operator in Figure 3c would be discarded as inadmissible: its *before* state includes the identifier "y," which is not found anywhere in the *after* state.

The idiomization process discussed above allows for limited synthesis of terms not present in the *after* state. Thus, idiomization turns some otherwise-inadmissible candidates into admissible ones. The candidate mutation operator in Figure 3d would be inadmissible if we had to synthesize the "0" in the *before* state. However, "0" is so common that it is always recognized as an idiomatic keyword in practice. Thus, the *before* state of Figure 3d can be constructed from the *after* state by replacing "1" with the idiomatic keyword "0".

Figure 3b shows the tokenized mutation operator extracted and generalized from Figure 3a. In the mutation-operator language of mutgen and mutins, ":" indicates a keyword, where the text that follows identifies the keyword itself. Likewise, "." indicates an operator, where the text that follows specifies the operator text. For identifiers and literals that appear in both the *before* and *after* states, we number *after* identifiers and literals starting from 1. "$$i$" represents the i^{th} identifier or literal. This notation lets us represent mutation operators that are polymorphic with respect to identifier names and literal values. Thus, the generalized mutation operator in Figure 3b can match a wide variety of "if" statements, not merely those that test the value of "x && y," as in Figure 3a. "$_" marks identifiers and literals that do not appear in the *before* text.

As seen in Figure 3b, mutation operators are stored in a plain-text format that humans can easily read and edit. This feature allows a user to create hand-written mutation operators for use in our mutation-testing system.

```
- }          - while (i < n);    - TMPFILE
+ } else     + while (i < n)     + TMPFILE % 512
```

(a) **(b)** **(c)**

Figure 4: Examples of real candidate mutation operators found within the experimental corpus.

Figure 4 shows some candidate mutation operators identified by mutgen. Figure 4a shows a patch that fixes a missing else keyword. From this patch, we harvest a mutation operator that can remove any else keyword immediately after a right curly bracket. The patch in Figure 4b generalizes into a mutation operator that can add a semi-colon to certain while statements. Figure 4c yields a mutation operator that, when applied to C code, can strip a modulo operation applied to a single identifier.

2.4 Filtering Heuristics

Our initial expectation that admissible candidate mutation operators would be rare proved to be untrue. On the contrary, our initial run of mutgen over the patch corpus used in our experiments yielded over twenty million mutation operators: more than it was reasonably possible to evaluate. Manual inspection revealed that many of these were not worth keeping. For example, some would only mutate comments or were so complex as to be unlikely to match token sequences from other code bases. We therefore extended mutgen with heuristic filters to detect and discard operators with less-promising potential. Mutgen's complete filtering sequence, applied in the order given below, is as follows:

1. **Too many tokens:** Candidates that consist of large amounts of code are so specific that they will probably not match in any other code base. Therefore, we discard candidates that affect eleven or more tokens.
2. **Too few tokens:** Conversely, single-token candidates would match too frequently to be practical. Therefore, we require that either the *before* or the *after* text contain at least two tokens. Note that identifier shifts (Section 2.6) can still apply to single-token *before* and *after* texts.
3. **ASCII art:** Candidates that contain three or more repeated operators are assumed to be comments and excluded. This situation commonly arises in line-spanning ASCII art such as "****," "----," or "////."

(4) **Comment detected:** Using the language definition (Section 2.1), we can recognize candidates that include the start or end of a comment. Mutating comments is not useful in mutation testing, so we exclude such changes.
(5) **Needs synthesis:** Per Section 2.3, we discard candidates for which the *pattern* contains identifiers or literals not present in the *replacement*. Our current version of the toolchain does not support the synthesis of identifiers (outside of the limited identifier conversion in identifier shifts or through the use of identifiers treated as keywords through idiomization) that would be required to apply these mutation operators to a target program.
(6) **Too many identifiers:** As for "Too many tokens," a candidate involving too many identifiers is unlikely to be applicable to new code. We discard candidates that affect more than four identifiers.
(7) **Too many adjacent identifiers:** A candidate containing three "identifiers" in a row is more likely to be natural-language text than programming-language source code; we assume that these candidates involve comments and exclude them from harvesting.
(8) **Identical tokenized strings:** Generalizing a candidate into a reusable mutation operator can make the before and after token streams identical, yielding a "mutation" operator that changes nothing. This situation can arise, for example, when the candidate merely affects whitespace. We discard these candidates.
(9) **Unbalanced brackets:** All mainstream languages include bracketing tokens that must appear in matched pairs, such as round parentheses, square brackets, and curly braces. Mutation operators can introduce mismatches when our harvester splits one commit into multiple separate changes, each of which affects only one side of a matched-bracket pair. We discard candidates that introduce mismatched counts of opening and closing round parentheses, square brackets, or curly braces. Introducing this filter increased the compilation rate of mutants produced by the toolchain from about 8.7% to about 14%.
(10) **Duplicated mutation operator:** Similar candidates can yield identical generalized mutation operators. We discard redundant copies.

Several of the above filters rely on configurable thresholds. Adjusting thresholds for total tokens or potential identifiers can dramatically change the total harvested mutation operators, while also changing the probability that each mutation operator can be matched with (and therefore inserted into) another code base. Some tuning may be needed for specific languages or coding styles. For example, function application can lead to multiple adjacent identifiers in many functional languages: "`sum x y z`" in ML or "`(sum x y z)`" in Lisp instead of "`sum(x, y, z)`" in C. In such languages, the "Too many adjacent identifiers" filter should only be used with a high threshold.

The result of applying these culling heuristics was to reduce the generated set from over twenty million mutation operators, most of them unusable, to roughly forty four thousand, of which a larger

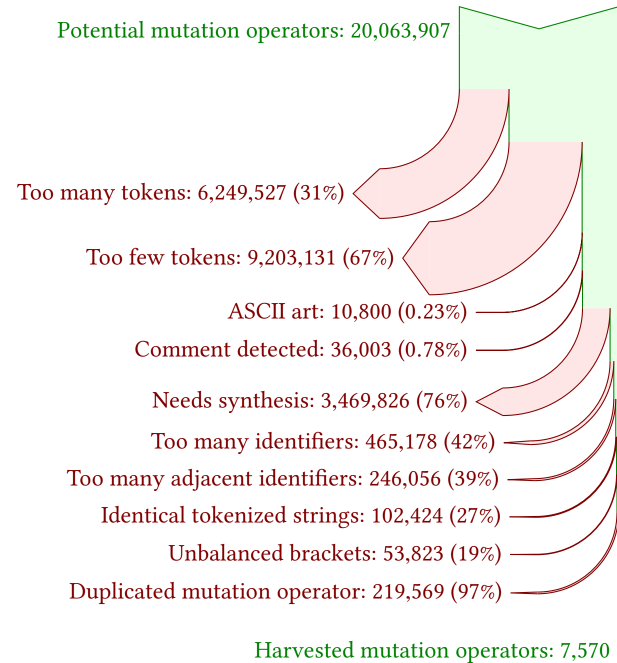

Figure 5: Potential mutation operators discarded and retained at each filtering stage

proportion can be applied to other code bases. Section 2.5 discusses the empirical behavior of these filters in greater detail.

2.5 Effect of Filtering Heuristics

Figure 5 depicts the filtering process as applied to Space. Flow begins at the top with 20,063,907 candidates and proceeds downward. Each filter removes some candidates and allows others to proceed to later stages. The width of each curved arrow represents the absolute number of potential mutation operators discarded at each stage; the actual count is reported immediately after the colon in each stage's description. For example, "Duplicated mutation operator" discards 219,569 candidates. The diminishing width of the straight flow descending along the right edge of the diagram is proportional to the number of candidates retained after all preceding steps. Numbers in parentheses are the fraction of surviving candidates discarded, expressed as percentage of candidates considered at each stage, not as a percentage of the 20,063,907 potential mutation operators gathered at the start. For example, the "Duplicated mutation operator" filter discards 97% of the mutation operators that had not already been eliminated in any preceding stage.

In absolute terms, "Too few tokens" is the major gatekeeper, accounting for nearly half of the initial candidates that do not make it through to the end. "Too many tokens" and "Needs synthesis" also discard large portions of the initial pool. The latter could potentially be relaxed by deeper semantic analysis to allow more ambitious synthesis beyond our idiomization technique. The other filters seem minor relative to the large starting candidate pool, but notice that each of these still discards tens or hundreds of thousands of candidates. "ASCII-art" detection is highly selective and therefore

Table 2: Most frequent identifier shifts in our experimental corpus

Before	After	Incidence
`__init`	`__devinit`	12,967
`module_exit`	`module_platform_driver`	12,897
`DEVICE_PRT`	`DBG_PRT`	10,097
`of_device`	`platform_device`	8,704
`m`	`y`	6,912
`CONFIG_PM`	`CONFIG_PM_SLEEP`	6,617
`CONFIG_EMBEDDED`	`CONFIG_EXPERT`	6,148
`mach`	`plat`	5,963
`A_UINT8`	`u8`	5,658
`device`	`platform_device`	5,610

has the least impact, discarding just 0.23% of the potential mutation operators it considers, but even this filter eliminates 10,800 candidates that would have been pointless to turn into mutation operators. When operating at these large scales, even relatively small contributors can be important.

2.6 Identifier Shifts

We call the second type of wild-caught mutant extracted by `mutgen` an *identifier shift*. During the extraction process, it is common for `mutgen` to identify a patch that consists solely of a change of one identifier to another. While the syntactic-mutation technique explicitly avoids synthesis during the insertion process (with the exception of a limited form permitted through idiomization), we capture these single-identifier changes, calling them *identifier shifts*, to allow an additional, limited form of synthesis.

Any patch that is observed to replace *solely* one identifier with another is marked as a candidate identifier shift. At the end of extraction, all candidate shifts with incidence above a configurable threshold are encoded as identifier shifts within the mutation-operator set. All identifier shifts extracted during the harvesting process are used both "forwards" and "backwards." That is, a single identifier-shift mutation operator can replace either the "before" identifier with the "after" or vice versa.

Table 2 shows example identifier shifts harvested from the corpus used in our experiments.

2.7 Insertion

Once the harvesting process produces a mutation-operator set, our mutant-insertion tool `mutins` can then apply mutation operators to a code base.

`Mutins` works by tokenizing all source-code input files using the same language-definition specification and rules discussed in Section 2.1. It then selects a mutation operator from the mutation-operator set. By default, the selection is done randomly, but the user may specify either a particular mutation operator in the mutation-operator set by index, or specify a seed for the random-number generator.[4]

[4]`Mutins` uses the Mersenne Twister[17] random-number generator, both for the generation of high-quality random numbers, as well as to allow seeds to be used across systems and allow faithful reproduction of random sequences.

Once the mutation operator is selected, `mutins` then attempts to match the mutation operator's pattern to any subset of the token stream generated from parsing the source code. All possible matches are identified, and if any exist, one is chosen randomly if the *insertion index* is not specified by the user. `Mutins` then replaces the tokens in the source file—preserving whitespace—with the tokens from the replacement in the mutation operator.

To avoid inserting mutants into non-executable portions of the source code, `mutins` uses the comment rules defined in the language-definition file—see Figure 2—to identify comments during the insertion process, and does not apply mutation operators to token sequences that lie within comments. In contrast to the harvesting process, the mutation-insertion process has the entire source file available for analysis, and so can more reliably identify comments because the full context is visible.

3 EXPERIMENTS

3.1 Repository Mining

We obtained mutation operators by mining public GitHub repositories that contain C code. We wanted to target the repositories with the largest number of commits; however, the GitHub API does not provide a way to search based on the number of commits. As a proxy for number of commits, we opted instead to select those repositories with the most *forks*, which is accessible via GitHub's API. The number of forks would seem to be a reasonable heuristic for projects with significant activity—and thus a higher rate of development, and commits from more developers. Qualitatively, the assumption appears to be warranted: the top 20 project repositories under this metric include the Linux kernel [27], memcached [18], and Redis [25]. For our experiments, we used the full revision histories of the top 50 project repositories, which consisted of approximately 600 thousand commits containing roughly 20 million individual `diff` blocks spanning 850 million lines of text.

3.2 Target Program

We used the 50 project histories to rerun (part of) an experiment reported by Andrews et al. [1], substituting the wild-caught mutation operators obtained from the 50 GitHub project histories for the set of mutation operators used by Andrews et al.

Andrews et al. experimented on programs from the SIR repository [8]. For each program, Andrews et al. generated a number of test suites by randomly choosing a subset of the tests in the program's full test suite. They then measured the mutation adequacy of each randomly chosen test suite by running each test suite over the set of all mutants of the program created by applying a single mutation operator at a single site in the program. By collecting these measurements, Andrews et al. constructed a model of the statistical distribution of the mutant-detection rate over arbitrary test suites, which they compared to a similarly constructed approximation of the distribution of hand-seeded faults.

To test the effectiveness of mutation testing, Andrews et al. worked with a wide variety of programs from SIR, including the Siemens suite. Among these, Space [28, 29] was the only program that they tested for which *real* faults were available instead of hand-introduced ones. Because we were interested in understanding how wild-caught mutants fare against a test suite's detection rates for

real faults, we worked only with Space. As distributed by SIR, Space has 38 buggy variants and one "gold" version with no known faults. Andrews et al. used the bug-free gold version; we did the same to allow direct comparison with Andrews et al.'s findings.

3.3 Procedure

Following the method of Andrews et al., we generated 5,000 100-case test-suite subsets from Space's set of 13,496 total test cases. Next, we ran `mutins` on Space, to identify each possible point at which a wild-caught mutation operator can be applied. We recorded each possible insertion in a list that could be fed to our test-suite-execution framework at a later time. We then divided the space of mutation insertion points into batches to be run in parallel on a large-scale computing platform capable of serving over 300 million hours of compute time annually.

We inserted each mutation and compiled the result; if compilation succeeded, we ran each of the 5,000 100-case test-suite subsets. The data was gathered in parallel because there are no interdependences among any of the runs of a test-suite subset.

Once all test batches completed, we recorded the number of mutants that successfully compiled. We also computed the *mutation-detection ratio*, $\mathrm{Am}(S)$, for each compiled mutant and each test suite, defined as follows:

Definition 3.1. Let S be a test suite. Then the *mutation-detection ratio* $\mathrm{Am}(S)$ is defined as follows:

$$\mathrm{Am}(S) = \frac{\text{\# of mutants detected by } S}{\text{\# of mutants not equivalent to the original program}}.$$

The denominator of $\mathrm{Am}(S)$ requires determining whether each mutant is equivalent to the original program, which is undecidable in general [3, 7, 22]. Therefore, Andrews et al. [1] adopt, and we reuse here, a decidable approximation:

$$\mathrm{Am}(S) = \frac{\text{\# of mutants detected by } S}{\text{\# of mutants detected by program's complete test suite}}.$$

In other words, any mutant that triggers no failure in Space's extensive 13,496-case complete test suite is assumed to be equivalent.

4 RESULTS

4.1 Research Question 1: Do Wild-Caught Mutation Operators Cover Existing Mutation Operators?

Just et al. [12] describe a set of mutation operators provided by the Major mutation framework [11]:

- **Replace constants.** `Mutgen` can extract mutation operators that replace constants in the system under test both through the idiomization technique (effectively turning literal constants into language keywords, which can then be extracted in the form of a syntactic mutation operator) or identifier shifts. If specific conversions are not found within the corpus from which mutation operators are harvested, a `mutins` user can manually add mutation operators that replace specific constants.
- **Replace operators.** All operators seen in the language-definition file used as an input to `mutgen` are capable of being extracted as syntactic mutation operator. Operator replacements can also be added manually to the mutation-operator set.
- **Modify branch conditions.** Operators to modify branch conditions can be extracted as syntactic mutation operators. (Section 2.3).
- **Delete statements.** `Mutgen` does not yet support the harvesting of statement deletions, but there is no impediment to doing so. In the terminology of Section 1, from a patch of the form

$$\epsilon \longrightarrow \text{after-code},$$

we can create a statement-deletion operator of the form

$$\text{pattern}_A \Rightarrow \epsilon.$$

Our framework allows for the replication of all four classes of mutation operators, although `mutgen` does not currently harvest statement-deletion operators. Future work will support the harvesting of these mutation operators; `mutins` already supports such mutation operators if `mutgen` were capable of producing them.

The PIT Mutation Testing suite[5] supports a set of eleven non-experimental mutation operators, many of which duplicate mutation operators provided by the Major mutation framework:

- **Conditionals Boundary Mutator, Conditionals Mutator, Invert Negatives Mutator, Math Mutator, Negate Increments Mutator.** `Mutins` can replicate these mutation operators in the same manner as Major's **Replace operators** mutation operators, as all of these mutation operators consist of replacing individual operators (or omit a unary minus from a larger expression, in the case of **Invert Negatives**).
- **Return Values Mutator.** `Mutins` can duplicate this mutation operator via idiomization (to harvest common literal numeric values 0 and 1) or via syntactic mutation operators for the language-specific keywords `true`, `false`, and `null`[5].
- **Void Method Calls Mutator** `Mutins` does not currently support statement deletion, of which this mutation operator is an instance.
- **Inline Constant Mutator.** `Mutins` can replicate this mutation operator through idiomization as in Major's **Replace constants** mutation operator.
- **Remove Conditionals Mutator, Constructor Calls Mutator, Non Void Method Calls Mutator.** `Mutins` can utilize harvested mutation operators of these types, so long as an example of a change of the type exists within the input corpus.

Our framework allows for the replication of ten out of the eleven non-experimental mutation operators supplied by PIT, again failing to directly reproduce statement deletion.

[5]The PIT framework operates on Java; while these keywords do not exist in the C language definition used in the experimentation in this paper, a Java language definition for `mutgen` properly identifies them as keywords and treats them as such during the harvesting process without idiomization.

4.2 Research Question 2: Do Wild-Caught Mutation Operators Extend Existing Mutation Operators?

In their study of whether real faults are coupled to mutants, Just et al. [12] found that for 27% of the real faults in their study, none of the triggering tests detected any additional mutants. They manually reviewed those faults, and classified them as follows: (i) cases where a mutation operator should be strengthened; (ii) cases where a new mutation operator should be introduced; and (iii) cases where no obvious mutation operator can generate mutants that are coupled to the real fault

In our experiments, we found that several of the mutation operators identified by Just et al. appeared among the mutation operators harvested by `mutgen`. Specifically, we are able to harvest mutation operators that are consistent with the classifications of Just et al.:

Stronger mutation operators

- *Argument swapping.* `Mutgen` is capable of harvesting patches that rearrange function arguments, which become mutation operators that perform the inverse rearrangement.
- *Argument omission.* `Mutgen` is capable of harvesting patches that contain a function call modified to have additional arguments, which become mutation operators that match a function call and replace it with one that has fewer arguments.
- *Similar library method called.* The identifier-shift technique (Section 2.6) allows `mutgen` to harvest mutation operators of this category by identifying patches in which a single identifier is replaced by another.
  Just et al. specifically mention a Java fault caused by a call to `indexOf`, where a call to `lastIndexOf` should have been performed. Our experiments, which used a C corpus, found multiple occurrences of the analogous C transformation: a `strchr` ⇒ `strrchr` identifier shift.

New mutation operators

- *Omit chaining method call.* `Mutgen` was able to identify mutation operators of this type, where the fault is a missing call to a one-argument function whose return type is equal to (or a subtype of) its argument's type. Specifically, it found patches in which a missing call to an SQL string-sanitization function was inserted.
- *Direct access of field.* While we were unable to find this mutation operator among the harvested operators—most likely because we were using only C patches—this mutation category could be generated by a combination of an identifier shift and a syntactic mutation operator.

Other mutation operators

- *Specific literal replacements.* The idiomization technique (Section 2.2) allows `mutgen` to identify specific literals to be used in mutation operators. To identify literals that are more relevant to the system under test, the implementation allows the system under test to be used as its own source of idioms.

Figure 6 illustrates that these operators are all within the harvesting capabilities of `mutgen`. Just et al. provide `diff`-formatted patches to illustrate faults not coupled to existing mutation operators; `mutgen` is able to harvest mutation operators automatically from the provided patches.[6] The patches provided by Just et al. were in Java; while our experiments exclusively used C, our toolchain is language agnostic and we were able to create a Java language-definition file and extract mutation operators from the provided Java patches. In addition to being able to harvest such mutation operators from `diff`-formatted patches, a user of our system can also manually specify additional mutation operators in all of the above categories.

Table 3: Experimental results when harvesting from backward or forward patches

Aspect	Backward	Forward
Extracted syntactic mutation operators	7,570	8,069
Extracted identifier shifts	5,000	5,000
Total number of syntactic mutants	139,289	183,683
Total number of applied identifier shifts	1,876	1,876
Successfully compiled syntactic mutants	20,803	21,617
Successfully compiled identifier shifts	127	127
Compilation rate	15%	12%
Average Am(S)	0.81	0.81
Median Am(S)	0.81	0.81

4.3 Research Question 3: Do Wild-Caught Mutation Operators Differ From Existing Mutation Operators?

Research Question 3 asks whether wild-caught mutants exhibit behavior that is quantifiably different than existing mutation operators. Table 3 summarizes some basic metrics from the mutation-testing experiment with Space.

Mutation-Detection Ratio. Andrews et al. [1] defined the sample-based mutation-detection ratio Am(S) (see Definition 3.1 in Section 3.3), and measured it as 0.75 when existing mutation-testing techniques were applied to Space [8] and Space's test suite. Using the same sample-based technique, we measured an Am(S) value of 0.81 for the mutation-operator set created via the wild-caught-mutants technique. This indicates that the mutation operators obtained by the wild-caught-mutants technique lead to mutants that roughly as hard to kill as mutants from traditional mutation operators.

Compilability. Using the wild-caught mutation operators, the compilation-success rate of the mutants created for Space was around 14% (see Table 3). Although, this rate is substantially larger than our original guess that the compilation-success rate would be less than 5%, the rate is comparatively low: Andrews et al. [1] reported a compilation-success rate of 92% for Space. However, because of the large number of mutation operators harvested, mutation testing via wild-caught mutants still appears feasible; our set of 34,439 compilable mutants is more than three times larger than Andrews et al.'s 11,379-mutant set (34,439 = 20,802 forward mutants + 21,617 backward mutants - 7,980 duplicate mutants).

[6] For some of these patches, it is necessary to supply command-line arguments to change the values of `mutgen`'s options from their defaults—specifically, those relating to total-identifier count and the commonality threshold for harvesting identifier shifts.

```
- return solve(min, max);
+ return solve(f, min, max);
```

(a) Math-369 fix as found in Just et al. [12]

```
- :return $1 .( $2 ., $3 .) .;
+ :return $1 .( $_ ., $2 ., $3 .) .;
```

(b) Math-369 fix as generalized by mutgen

```
- int indexOfDot = namespace.indexOf('.');
+ int indexOfDot = namespace.lastIndexOf('.');
```

(c) Closure-747 fix as found in Just et al. [12]

```
indexOf ⇒ lastIndexOf
```

(d) Closure-747 fix as generalized to an identifier shift by mutgen

```
- return ... + toolTipText + ...;
+ return ... + ImageMapUtilities.htmlEscape(
    toolTipText) + ...;
```

(e) Chart-591 fix as found in Just et al. [12]

```
- :return ... .+ $1 .+ ... .;
+ :return ... .+ $_ .. $_ .( $1 .) .+ ... .;
```

(f) Chart-591 fix as generalized by mutgen

```
- FastMath.pow(2 * FastMath.PI, -dim / 2)
+ FastMath.pow(2 * FastMath.PI, -0.5 * dim )
```

(g) Math-929 fix as found in Just et al. [12]

```
- $1 .. :pow .( :2 .* $1 .. $3 ., .- $4 ./ :2 .)
+ $1 .. :pow .( :2 .* $1 .. $3 ., .- :0.5 .* $4 .)
```

(h) Math-929 fix as generalized by mutgen, with "pow," "2," and "0.5" keywords added by idomization

```
- return getPct((Comparable<?>) v);
+ return getCumPct((Comparable<?>) v);
```

(i) Math-337 fix as found in Just et al. [12]

```
getPct ⇒ getCumPct
```

(j) Math-337 fix as generalized to an identifier shift by mutgen

```
- lookupMap = new HashMap<CharSequence,
    CharSequence>();
+ lookupMap = new HashMap<String, CharSequence>();
```

(k) Lang-882 fix as found in Just et al. [12]

```
- $1 .= :new $2 .< $3 ., $3 .> .( .) .;
+ $1 .= :new $2 .< $_ ., $3 .> .( .) .;
```

(l) Lang-882 fix as generalized by mutgen

```
- if (u * v == 0)
+ if ((u == 0) || (v == 0))
```

(m) Math-238 fix as found in Just et al. [12]

```
- :if .( $1 .* $2 .== :0 .)
+ :if .( .( $1 .== :0) .|| .( $2 .== :0 .) .)
```

(n) Math-238 fix as generalized by mutgen, with "0" keyword added by idiomization

Figure 6: Examples of mutation operators proposed by Just et al. [12] and identified by mutgen

The majority of failed compilations (64%) arise from simple parsing errors. Another 21% fail because mutation has turned the left operand of an assignment into a non-assignable expression (i.e., not a C *lvalue*). Other frequent compilation errors include 5% due to invalid operands to binary operators (e.g., "+" applied to a pointer and a `double`) and 3% due to using an undeclared identifier. Compilation errors of these kinds are to be expected, given the lexical level at which we operate. Traditional mutation operators limit changes to ones that are unlikely to ever introduce parsing errors. For example, negating an `if` condition or replacing a "<" with a "<=" will not break compilation except under truly exceptional circumstances. Thus, the high compilation rates of traditional mutants arise essentially by construction. The wild-caught-mutants approach offers no such guarantees. That means we waste more time on failed compilations, but it also means that we have the potential to change code in much more interesting ways.

4.4 Research Question 4: Are "Forward" and "Backward" Patches Different?

While considering patches in the backward direction ("backward patches") intuitively seems more likely to (re)introduce bugs, which is good from the standpoint of mutation testing, we also tried harvesting mutation operators by considering the same set of patches in the forward direction ("forward patches").

Overlap. We found that the overlap is considerable between the sets of mutation operators harvested by considering patches in the "forward" and "backward" directions, but there is significant non-overlap: of the 13,929 unique mutation operators found using both techniques, 5,860 were found only from backward patches, 6,359 were found only from forward patches, and 1,710 were found by both techniques.

Mutation-Detection Ratio. The mutants caused by mutation operators harvested by forward and backward patches are ultimately equally difficult to kill: average and median Am(S) scores are 0.81 in each direction, per Table 3. This may seem surprising, if backward patches truly represent bug reintroduction. However, one must keep in mind that the "gold" version of Space used in our experiments passes its entire, extensive test suite. The test suite, then, effectively traps Space into a rather narrow set of allowed behaviors. Any deviation from that, whether to fix a fault or not, is likely to trigger at least one test case failure. Given the constraints of an extensive test suite, any change will look like a new fault, whether derived from backward or forward patches. Ultimately, forward patches may still describe interesting human-generated changes, and therefore harvesting them can be a worthy enhancement to backward-patch harvesting.

Ability to Reproduce Faults in Space. To evaluate the differences between mutation operators harvested from forward and backward patches, we examined the faults in the 38 faulty versions of Space. We classified each fault as to whether the two kinds of harvested mutation operators could reintroduce them, if mutation testing were carried out on the "gold" version.

- Seven faulty versions (3, 4, 5, 6, 20, 21, and 28) had faults that would be reintroduced by some mutation operator harvested from backward patches.
- One faulty version (30) had a fault that would be reintroduced by a mutation operator that was harvested from both the "forward" and "backward" patches.
- Five faulty versions (1, 2, 18, 23, and 33) had faults that could potentially be reintroduced by `mutins`, but with mutation operators that were not harvested—in either direction—from the 50 GitHub projects that we analyzed. Of the five, faulty version 18 had a fault that could potentially be reintroduced via an identifier shift, albeit one that we did not harvest; the faults in the remaining four are expressible as syntactic mutation operators.

The remaining 25 faulty versions required mutations outside of the scope of our current techniques. The majority of these are expressible as syntactic mutation operators, but involve too many lexical tokens to survive our filtering heuristics.

While these results are limited in scope, they provide weak support for the conjecture that, compared to forward-harvested mutation operators, backward-harvested operators can introduce defects that more closely resemble defects introduced by real programmers. Ultimately, while the *ideal* is to insert "bug-like" changes into the target program, a robust test suite must also be able to identify behavior changes introduced by human programmers, which our wild-caught mutants—whether derived from bug fixes or not—simulate.

5 THREATS TO VALIDITY

There are several threats to the validity of our work.

We employ small changes—10 lexical tokens or fewer, and typically single-line—to generate our mutation operators. These size-limited patches represent a distinct subset of all possible changes to code, and as a result we do not derive mutation operators from all valid patches observed. We enact this limitation because the more complex each individual harvested mutation operator is, the less likely it is to be matched in any particular piece of code to which it is applied. Smaller and simpler mutation operators yield a substantially higher proportion of matchable mutation operators; syntactic mutation operators larger than those we harvest clutter the system, but are rarely able to be applied to a system under test.

The idealized goal of mutation testing is to measure a test suite's quality, by measuring its ability to detect faults of the kind that might be inserted into the program in the future. One may question whether *reversals of past changes* are good candidates as predictors of the kinds of future faults that one wants the test suite to detect. Our experiment with the 38 faulty versions of Space provides a small amount of evidence that backward-patch harvesting is a better source of such candidates than forward-patch harvesting.

Even if our specific harvesting approach proves to be sub-optimal, the general idea of supporting mutation testing using information harvested from a revision-control system would still have much potential. A possible improvement, which we plan to investigate in follow-on work, is to extend the harvesting operation to include information from a bug-tracking system, such as Bugzilla [4]. Śliwerski et al. [26] investigated how the combination of a revision-control system and a bug-tracking system provides a way to identify fix-inducing patches in the revision history.[7] Such an approach would provide three sources of input for harvesting mutation operators: (i) the fix-inducing patch; (ii) the corrective patch; and (iii) the commonalities between the fix-inducing patch and the corrective patch.

Rather than experiment shallowly across a large benchmark suite, we chose to focus on evaluating in depth a single application: Space, from the SIR repository [8]. This decision allowed us to make direct comparisons with the empirical findings of Andrews et al. [1]. However, if Space is unlike other real-world code, then this difference would harm the external validity of our findings—i.e., the extent to which our conclusions can be generalized to other situations. In spite of that risk, Space is an appealing subject for an experiment on the effectiveness of mutation testing. It is not a synthetic benchmark, but rather is a mature piece of software that has been subject to years of production use. Among the programs studied by Andrews et al., only Space had variants with real faults instead of hand-introduced ones. Moreover, at 9,124 lines of code, Space is larger than the programs in the Siemens suite. For Space, `mutins` generated 241,517 single-mutation mutants, of which 34,439 were compilable. The set of 241,517 mutants is a nontrivial set, but was still small enough that, for each mutant, we could run 5,000 100-case test-suite subsets.

6 RELATED WORK

To the best of our knowledge, the wild-caught-mutants technique is novel; however, several other projects have used related ideas. Some of the latter techniques could be used to enhance our methods for extracting mutation operators.

Śliwerski et al. [26] describe a technique to identify fix-inducing patches within a revision history, and propose applying similar tactics to identify failure-inducing patches. Many others have used similar strategies, all based on recognizing specific keywords (such

[7] As defined by Śliwerski et al., a *fix-inducing patch* is one that causes a later bug fix.

as "fixed" or "bug") or bug IDs (such as "#42233") in commit messages [2, 6, 9, 13, 20]. Mutgen could be extended to use these techniques to attempt to identify "higher-quality" mutation operators by inferring properties of the changes induced by specific patches in the source corpus.

Le et al. [16] mine revision histories to extract bug-repairing patches, and use these as a basis for program repair. We provide a technique that, effectively, does the opposite—we use mined patches to break code instead of fixing it.

Coccinelle's *semantic patches* are generalized patches that can be applied to code, much like our syntactic mutation technique [23]. However, Coccinelle works with manually authored patches, whereas we harvest new mutation operators automatically. Coccinelle applies semantic changes to multiple blocks of code, and has been applied to bug detection [14, 15], whereas we focus on breaking code for the purpose of mutation testing. The mutation-testing context lets our toolchain utilize simpler patches, as well as harvest them automatically.

As a follow-on to Coccinelle, Palix et al. [24] developed *Herodotos*, a system to track the evolution of patterns in code through analysis of a revision history. We share Palix et al.'s interest in the evolution of code, although we focus on pairwise diffs between adjacent revisions rather than entire revision histories. Herodotos requires more manual intervention than our toolchain, most notably to create and generalize the initial patterns to be tracked across revisions. This approach is sensible for Herodotos, which ultimately drives interactive code-understanding tools. However, our batch-testing usage scenario calls for a fully automated approach.

Nam et al. [21] describe a technique for identifying bug-fixing commits in a source-control repository and calibrating mutation testing to utilize mutation operators that more closely resemble the reverse of changes observed in bug-fixing commits. Nam et al. look for keywords in commit messages, as many others have done, and also manually inspect commits to confirm that they are indeed fixes. Our approach is more automated, as we harvest all patches that fit our purely syntactic filtering heuristics. Likewise, Nam et al. craft several new mutation operators by hand, whereas our approach automates the entire process of harvesting and generalizing new mutation operators. Our automation-focused approach may be less selective, but it allows us to work with a corpus two orders of magnitude larger than that used by Nam et al.

7 EXPERIMENTAL ARTIFACTS

Our core mutation tools, consisting of mutgen and mutins, are available at https://github.com/d-bingham/wildcaughtmutants.

We also provide tools to demonstrate our experiments at https://github.com/d-bingham/fse2017artifact. However, reproducing our complete set of experiments would require months of processor time (and as such was executed on a high-throughput computing platform). Therefore, this experimental artifact recreates scaled-down versions of the experiments described in Section 3.

The artifact allows the user to harvest a set of mutation operators from scraped GitHub repositories (omitting the full Linux kernel source due to space and time concerns). Once a set of mutation operators are harvested from this corpus, the artifact then generates thirty randomly-chosen mutants (chosen as mutation operators and insertion indices into the Space program), attempts to compile them, and evaluates the mutated programs against Space's test suite. With the pass/fail results from each test case, the artifact then generates random "virtual" test suites to calculate Am(S) scores for the generated mutants.

The artifact can be executed via a provided shell script or through the use of a Docker[19] container, allowing demonstration of a small portion of our experiment in a highly portable manner.

8 CONCLUSION

For mutation testing to provide a useful measure of the sensitivity of a test suite, it must produce not only faults within the system under test, but faults that mimic those caused by the actual developers working on a project. Just et al. demonstrated that faults introduced through mutation testing can serve as proxies for real faults introduced by developers and be effectively used to evaluate the sensitivity of a testing suite, although they also described limitations of existing sets of mutation operators. We expand upon that work by automatically harvesting mutation operators—wild-caught mutants—and comparing the capabilities of the harvested mutation operators to those of existing mutation operators.

Andrews et al. [1], discussing threats to validity, caution that "It is also expected that the results of our study would vary depending on the mutation operators selected. . . ." Our findings provide strong empirical support for this expectation. As opposed to existing "synthetic" mutation testing techniques, every mutation we create is based on a (reversed) change that some real programmer made to some real piece of code. Our wild-caught approach produces novel mutation operators, in turn creating defects that are about as difficult to kill as those arising from existing synthetic mutation operators or Space's 38 naturally-arising faults. Whether existing synthetic mutation operators or our wild-caught mutation operators can objectively be characterized as more "realistic" remains an open question.

"Realism" arguments aside, it is clear that developers benefit if their test suites can be challenged by bugs that resemble those they might expect programmers to introduce. Our wild-caught-mutants technique can be a source of such bugs. Instead of crafting mutation operators by hand, we believe that our results demonstrate that wild-caught mutants provide a stronger method for evaluating the sensitivity of test suites.

ACKNOWLEDGMENTS

The authors are grateful to Josh Deaver for insightful discussions regarding visualization of the effects of our filtering heuristics; to Michael Ernst for guidance on experimental evaluation strategies for mutation research; and to the UW–Madison Center For High Throughput Computing (CHTC) for computational support.

This research was supported in part by a gift from Rajiv and Ritu Batra; Defense Advanced Research Projects Agency MUSE award FA8750-14-2-0270 and STAC award FA8750-15-C-0082; and National Science Foundation grants CCF-1217582, CCF-1318489, and CCF-1420866. Opinions, findings, conclusions, or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the sponsoring agencies.

REFERENCES

[1] J. H. Andrews, L. C. Briand, and Y. Labiche. 2005. Is Mutation an Appropriate Tool for Testing Experiments?. In *Proceedings of the 27th International Conference on Software Engineering (ICSE '05)*. ACM, New York, NY, USA, 402–411. https://doi.org/10.1145/1062455.1062530

[2] Cathal Boogerd and Leon Moonen. 2008. Assessing the value of coding standards: An empirical study. In *24th IEEE International Conference on Software Maintenance (ICSM 2008), September 28 - October 4, 2008*. IEEE Computer Society, Beijing, China, 277–286. https://doi.org/10.1109/ICSM.2008.4658076

[3] Timothy A. Budd and Dana Angluin. 1982. Two notions of correctness and their relation to testing. *Acta Informatica* 18, 1 (1982), 31–45. https://doi.org/10.1007/BF00625279

[4] Bugzilla development team. 2016. Home :: Bugzilla :: bugzilla.org. (May 2016). https://www.bugzilla.org/

[5] Henry Coles. 2017. PIT Mutation Testing. (2017). [Online; accessed Jun. 2017].

[6] Davor Čubranić and Gail C. Murphy. 2003. Hipikat: Recommending Pertinent Software Development Artifacts. In *Proceedings of the 25th International Conference on Software Engineering (ICSE '03)*. IEEE Computer Society, Washington, DC, USA, 408–418. http://dl.acm.org/citation.cfm?id=776816.776866

[7] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. 1978. Hints on Test Data Selection: Help for the Practicing Programmer. *IEEE Computer* 11, 4 (1978), 34–41. https://doi.org/10.1109/C-M.1978.218136

[8] Hyunsook Do, Sebastian G. Elbaum, and Gregg Rothermel. 2005. Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact. *Empirical Software Engineering: An International Journal* 10, 4 (2005), 405–435.

[9] Michael Fischer, Martin Pinzger, and Harald Gall. 2003. Populating a Release History Database from Version Control and Bug Tracking Systems. In *Proceedings of the International Conference on Software Maintenance (ICSM '03)*. IEEE Computer Society, Washington, DC, USA, 23–. http://dl.acm.org/citation.cfm?id=942800.943568

[10] Richard G. Hamlet. 1977. Testing Programs with the Aid of a Compiler. *IEEE Trans. Software Eng.* 3, 4 (1977), 279–290. https://doi.org/10.1109/TSE.1977.231145

[11] René Just. 2014. The Major Mutation Framework: Efficient and Scalable Mutation Analysis for Java. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA 2014)*. ACM, New York, NY, USA, 433–436. https://doi.org/10.1145/2610384.2628053

[12] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. 2014. Are Mutants a Valid Substitute for Real Faults in Software Testing?. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, New York, NY, USA, 654–665. https://doi.org/10.1145/2635868.2635929

[13] Sunghun Kim and Michael D. Ernst. 2007. Which Warnings Should I Fix First?. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC-FSE '07)*. ACM, New York, NY, USA, 45–54. https://doi.org/10.1145/1287624.1287633

[14] Julia Lawall, Ben Laurie, René Rydhof Hansen, Nicolas Palix, and Gilles Muller. 2010. Finding Error Handling Bugs in OpenSSL using Coccinelle. In *Proceeding of the 8th European Dependable Computing Conference, EDCC 2010*. IEEE Computer Society, Valencia, Spain, 191–196.

[15] Julia L. Lawall, Julien Brunel, Nicolas Palix, René Rydhof Hansen, Henrik Stuart, and Gilles Muller. 2009. WYSIWIB: A Declarative Approach to Finding Protocols and Bugs in Linux Code. In *The 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE Computer Society, Estoril, Portugal, 43–52.

[16] Xuan-Bach D. Le, David Lo, and Claire Le Goues. 2016. History driven program repair. In *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, Vol. 1. IEEE Computer Society, Suita, Osaka, Japan, 213–224.

[17] Makoto Matsumoto and Takuji Nishimura. 1998. Mersenne Twister: A 623-dimensionally Equidistributed Uniform Pseudo-random Number Generator. *ACM Trans. Model. Comput. Simul.* 8, 1 (Jan. 1998), 3–30. https://doi.org/10.1145/272991.272995

[18] memcached community. 2017. Memcached. (Jan. 2017). https://github.com/memcached/memcached

[19] Dirk Merkel. 2014. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux Journal* 2014, 239, Article 2 (March 2014), 16 pages. http://dl.acm.org/citation.cfm?id=2600239.2600241

[20] Audris Mockus and Lawrence G. Votta. 2000. Identifying Reasons for Software Changes Using Historic Databases. In *Proceedings of the International Conference on Software Maintenance (ICSM'00) (ICSM '00)*. IEEE Computer Society, Washington, DC, USA, 120–. http://dl.acm.org/citation.cfm?id=850948.853410

[21] J. Nam, D. Schuler, and A. Zeller. 2011. Calibrated Mutation Testing. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. IEEE Computer Society, Washington, DC, USA, 376–381. https://doi.org/10.1109/ICSTW.2011.57

[22] A. Jefferson Offutt and Jie Pan. 1996. Detecting Equivalent Mutants and the Feasible Path Problem. In *Proceedings of the 1996 Annual Conference on Computer Assurance*. IEEE Computer Society, Gaithersburg, Maryland, 224–236.

[23] Yoann Padioleau, René Rydhof Hansen, Julia L. Lawall, and Gilles Muller. 2006. Semantic Patches for Documenting and Automating Collateral Evolutions in Linux Device Drivers. In *PLOS 2006: Linguistic Support for Modern Operating Systems*. ACM, San Jose, CA, Article 10, 6 pages.

[24] Nicolas Palix, Julia Lawall, and Gilles Muller. 2010. Tracking Code Patterns over Multiple Software Versions with Herodotos. In *Proceedings of the 9th International Conference on Aspect-Oriented Software Development*. ACM, Rennes and Saint Malo, France, 169–180. https://doi.org/10.1145/1739230.1739250

[25] Salvatore Sanfilippo. 2017. Redis. (Feb. 2017). https://github.com/antirez/redis

[26] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. 2005. When Do Changes Induce Fixes? *SIGSOFT Softw. Eng. Notes* 30, 4 (May 2005), 1–5. https://doi.org/10.1145/1082983.1083147

[27] Linus Torvals. 2017. Linux kernel. (Feb. 2017). https://github.com/torvalds/linux

[28] Filippos I. Vokolos and Phyllis G. Frankl. 1998. Empirical Evaluation of the Textual Differencing Regression Testing Technique. In *1998 International Conference on Software Maintenance, ICSM 1998, Bethesda, Maryland, USA, November 16-19, 1998*. IEEE Computer Society, Washington, DC, USA, 44–53. https://doi.org/10.1109/ICSM.1998.738488

[29] W. Eric Wong, Joseph Robert Horgan, Aditya P. Mathur, and Alberto Pasquini. 1999. Test set size minimization and fault detection effectiveness: A case study in a space application. *Journal of Systems and Software* 48, 2 (1999), 79–89. https://doi.org/10.1016/S0164-1212(99)00048-5