An Analysis of the Search Spaces for Generate and Validate Patch Generation Systems

Fan Long and Martin Rinard
MIT EECS & CSAIL, USA
{fanl, rinard}@csail.mit.edu

ABSTRACT

We present the first systematic analysis of key characteristics of patch search spaces for automatic patch generation systems. We analyze sixteen different configurations of the patch search spaces of SPR and Prophet, two current state-of-the-art patch generation systems. The analysis shows that 1) correct patches are sparse in the search spaces (typically at most one correct patch per search space per defect), 2) incorrect patches that nevertheless pass all of the test cases in the validation test suite are typically orders of magnitude more abundant, and 3) leveraging information other than the test suite is therefore critical for enabling the system to successfully isolate correct patches.

We also characterize a key tradeoff in the structure of the search spaces. Larger and richer search spaces that contain correct patches for more defects can actually cause systems to find fewer, not more, correct patches. We identify two reasons for this phenomenon: 1) increased validation times because of the presence of more candidate patches and 2) more incorrect patches that pass the test suite and block the discovery of correct patches. These fundamental properties, which are all characterized for the first time in this paper, help explain why past systems often fail to generate correct patches and help identify challenges, opportunities, and productive future directions for the field.

Categories and Subject Descriptors

D.2.5 [**SOFTWARE ENGINEERING**]: Testing and Debugging

Keywords

Program repair, Patch generation, Search space

1. INTRODUCTION

Software defects are a prominent problem in software development efforts. Motivated by the prospect of reducing human developer involvement, researchers have developed a range of techniques that are designed to automatically correct defects. In this paper we focus on generate and validate patch generation systems, which work with a test suite of test cases, generate a set of candidate patches, then test the patched programs against the test suite to find a patch that validates [27, 17, 40, 28, 14, 29, 18, 20].

Patch quality is a key issue for generate and validate systems. Because the patches are validated only against the test cases in the test suite, there is no guarantee that the patch will enable the program to produce correct results for other test cases. Indeed, recent research has shown that 1) the majority of patches accepted by many current generate and validate systems fail to generalize to produce correct results for test cases outside the validation test suite [29, 7, 37] and 2) accepted patches can have significant negative effects such as the introduction of new integer and buffer overflow security vulnerabilities, undefined accesses, memory leaks, and the elimination of core application functionality [29]. These negative effects highlight the importance of generating not just *plausible patches* (we define plausible patches to be patches that pass all of the test cases in the patch validation test suite) but *correct patches* that do not have latent defects and do not introduce new defects or vulnerabilities.

A rich search space that contains correct patches for target defects can be critical to the success of any automatic patch generation system. Indeed, recent research indicates that impoverished search spaces that contain very few correct patches is one of the reasons for the poor performance of some prominent previous patch generation systems [17, 40, 29]. While more recent systems work with patch spaces that contain significantly more correct patches [18, 20], continued progress in the field requires even richer patch spaces that contain more successful correct patches. But these richer spaces may also complicate the ability of the system to identify correct patches within the larger sets of plausible but incorrect patches.

1.1 SPR and Prophet

SPR [18] and Prophet [20] are two current state-of-the-art generate and validate patch generation systems. Both systems apply transformations to statements identified by an error localization algorithm to obtain patches that they validate against a test suite. SPR uses a set of hand-coded patch prioritization heuristics. Prophet uses machine learning to characterize features of previously successful human patches and prioritizes candidate patches according to these features. The goal is to prioritize a correct patch as the first patch to validate.

The baseline SPR and Prophet search spaces contain correct patches for 19 out of 69 defects in a benchmark set of

ICSE '16,' May 14-22, 2016, Austin, TX, USA
© 2016 ACM. ISBN 978-1-4503-3900-1/16/05...$15.00
DOI: http://dx.doi.org/10.1145/2884781.2884872

defects from eight open source projects [18, 20].[1] For 11 of these defects, the first SPR patch to validate is a correct patch. For 15 of these defects, the first Prophet patch to validate is a correct patch. The GenProg [17], AE [40], RSRepair [28], and Kali [29] systems, in contrast, produce correct patches for only 1 (GenProg, RSRepair) or 2 (AE, Kali) of the defects in this benchmark suite [29]. Moreover, the correct SPR and Prophet patches for the remaining defects lie outside the GenProg, RSRepair, and AE patch spaces, which suggests that these systems will *never* be able to produce correct patches for these remaining defects. We note that this benchmark set was developed not by us, but by others in an attempt to obtain a large, unbiased, and realistic benchmark set [17].

As these results highlight, the characteristics of the patch search space are central to the success of the patch generation system. But despite the importance of the search space in the overall success of the system, we have been able to find no previous systematic investigation of how the structure of the search space influences critical characteristics such as the density of correct and plausible patches and the ability of the system to identify correct patches within the broader class of plausible (but potentially incorrect) patches. Given the demonstrated negative effects of plausible but incorrect patches generated by previous systems [29], these characteristics play a critical role in the overall success or failure of any patch generation system that may generate plausible patches with such negative effects.

1.2 Patch Search Space Analysis

We present a systematic analysis of the SPR and Prophet patch search spaces. This analysis focuses on the density of correct and plausible patches in the search spaces, on the ability of SPR and Prophet to prioritize correct patches, and on the consequences of two kinds of changes, both of which increase the size of the search space: 1) increases in the number of candidate program statements to which patches are applied and 2) new program transformation operators that can generate additional patches. Starting from the SPR and Prophet baseline search spaces, these changes make it possible to construct a collection of search spaces, with each search space characterized by a combination of the set of transformations and the number of candidate program statements that together generate the search space.

We perform our analysis on a benchmark set of 69 real world defects in eight large open source applications. This benchmark set was used to evaluate many previous patch generation systems [17, 40, 29, 28, 18, 20]. For each defect, we first analyze the full search space to determine whether or not it contains a correct patch. We acknowledge that, in general, determining whether a specific patch corrects a specific defect can be difficult (or in some cases not even well defined). But for the defects in the benchmark set, this never happens — the correct patches that are within the search spaces are all small, all match the corresponding developer patches, and the distinction between correct and incorrect patches is clear.

[1]The paper that presented this benchmark set states that the benchmark set contains 105 defects. An examination of the relevant commit logs and applications indicates that 36 of these defects are actually deliberate functionality changes, not defects. In this paper we focus on the remaining 69 actual defects as within the scope of the paper.

Because the full search spaces can, in general, be too large to exhaustively search within any reasonable time limit, we also consider the subset of the search spaces that can be explored by SPR and Prophet within a reasonable timeout, in this paper 12 hours. Working with these explored subsets of spaces, we analyze the number of plausible and correct patches that each subset contains and the effect of the SPR and Prophet patch prioritization on the ability of these systems to identify correct patches within the much larger sets of plausible but incorrect patches in these search spaces.

1.3 Results Overview

The experimental results indicate that:

- **Sparse Correct Patches:** Correct patches occur only sparsely within the search spaces. For 45 of the 69 defects, the search spaces contain no correct patches. For 15 of the remaining 24 defects, the search spaces contain at most 1 correct patch. The largest number of correct patches for any defect in any search space is 4.
- **Relatively Abundant Plausible Patches:** In comparison with correct patches, plausible (but incorrect) patches are relatively abundant. For all of the benchmarks except php, the explored search spaces typically contain hundreds up to a thousand times more plausible patches than correct patches. These numbers highlight the difficulty of isolating correct patches among the large sets of plausible but incorrect patches.
  The explored search spaces for php, in contrast, typically contain only tens of times more plausible patches than correct patches. And for three of the php defects, all of the (one or two) plausible patches are correct. The density of plausible patches is related to the strength of the validation test suite — weak test suites filter fewer incorrect patches. We attribute the difference in plausible patch density to the strength of the php test suite — the php test suite contains an order of magnitude more test cases than any other benchmark.
- **SPR and Prophet Effectiveness:** The SPR and Prophet patch prioritization mechanisms are both effective at isolating correct patches within the explored plausible patches. Despite the relatively scarcity of correct patches, with the baseline search space, correct patches for 14 defects are within the first ten patches to validate for SPR; correct patches for 16 defects are within the first ten patches to validate for Prophet.
- **Search Space Tradeoffs:** Increasing the search space beyond the SPR and Prophet baseline increases the number of defects that have a correct patch in the search space. But it does not increase the ability of SPR and Prophet to find correct patches for more defects — in fact, these increases often cause SPR and Prophet to find correct patches for *fewer* defects!
  We attribute this phenomenon to the following tradeoff. Increasing the search space also increases the number of candidate patches and may increase the number of plausible patches. The increased number of candidate patches consumes patch evaluation time and reduces the density of the correct patches in the search space. The increased number of plausible but incorrect patches increases the chance that such patches will block the correct patch (i.e., that the system will encounter plausible but incorrect patches as the first patches to validate).

These facts highlight the importance of including information other than the test suite in the patch evaluation process. SPR includes information in the form of hand-coded patch prioritization heuristics. Prophet leverages information available via machine learning from successful human patches. This information is responsible for the ability of these systems to successfully identify correct patches in the baseline search space. The results also highlight that there is still room for improvement, especially with richer search spaces that contain correct patches for more defects.

Previous Systems: These facts also help explain past results from other systems. GenProg, AE, and RSRepair generate very few correct patches [29]. Part of the explanation is that the search space exploration algorithms for these systems are no better than random search [28].[2] Once one appreciates the relative abundance of plausible but incorrect patches and the relative scarcity of correct patches, it is clear that any algorithm that is no better than random has very little chance of consistently delivering correct patches without very strong test suites. And indeed, the majority of the patches from these previous systems simply delete functionality and do not actually fix the defect [29].

ClearView: ClearView, in sharp contrast, does leverage information other than the test suite, specifically learned invariants from previous successful executions [27]. For nine of the ten defects on which ClearView was evaluated, ClearView successfully patched the defect after evaluating at most three candidate patches. These results highlight how targeting a defect class and leveraging fruitful sources of information can dramatically increase the successful patch density.

1.4 Future Directions

Our results highlight the scalability challenges associated with generalizing existing search spaces to include correct patches for more defects. One obvious future direction, deployed successfully in past systems [32, 21, 6], is to address scalability issues by developing smaller, more precisely targeted search spaces for specific classes of defects. An alternative is to infer transformation operations from correct human patches (instead using manually defined transformations). The goal is to obtain a search space that contains correct patches for common classes of commonly occurring defects while still remaining tractable.

More broadly, it is now clear that generate and validate systems must exploit information beyond current validation test suites if they are to successfully correct any but the most trivial classes of defects [27, 20, 29]. One prominent direction is to exploit existing correct code in large code repositories to obtain new correct patches, either via sophisticated machine learning techniques that learn to recognize or even automatically generate correct code, automatically transferring correct code (either within or between applications), or even generalizing and combining multiple blocks of correct code throughout the entire software ecosystem. Encouraging initial progress has been made in all of these directions [20, 35, 38, 13]. The current challenge is to obtain more sophisticated patches for broader classes of defects.

[2]Other parts of the explanation include search spaces that apparently contain very few correct patches and errors in the patch infrastructure that cause the system to accept patches that do not even pass the test cases in the test suites used to validate the patches [29].

An orthogonal direction is to obtain stronger test suites or even explicit specifications that can more effectively filter incorrect patches. One potential approach is to observe correct input/output pairs to learn to recognize or even automatically generate correct outputs for (potentially narrowly targeted) classes of inputs. Another approach leverages the availability of multiple implementations of the same basic functionality (for example, multiple image rendering applications) to recognize correct outputs. Combining either of these two capabilities with automatic input generation could enable the automatic generation of much stronger test suites (with potential applications far beyond automatic patch generation). Specification mining may also deliver (potentially partial) specifications that can help filter incorrect patches.

1.5 Contributions

This paper makes the following contributions:

- **Patch Space Analysis:** It presents an analysis of the patch search spaces of SPR and Prophet, including how these patch spaces respond to the introduction of new transformation operators and increases in candidate program statements. The analysis characterizes:
  - **Correct Patches:** The density at which correct patches occur in the full search spaces and the explored space subsets.
  - **Plausible Patches:** The density at which plausible patches occur in the explored space subsets.
  - **Patch Prioritization:** The effectiveness of the SPR and Prophet patch prioritization mechanisms at isolating the few correct patches within the much larger set of plausible patches (the vast majority of which are not correct).

  This paper presents the first characterization of how correct and plausible patch densities respond to increases in the size and sophistication of the search space. It also presents the first characterization of how these search space changes affect the ability of the patch generation system to identify the few correct patches within the much larger sets of plausible but incorrect patches.
- **Tradeoff:** It identifies and presents results that characterize a tradeoff between the size and sophistication of the search space and the ability of the patch generation system to identify correct patches. To the best of our knowledge, this is the first characterization of this tradeoff.
- **Results:** It presents experimental results from SPR and Prophet with different search spaces. These results are derived from an analysis of 1104 different search spaces for the 69 benchmark defects (we consider 16 search spaces for each defect) and 768 patch generation executions for the 24 defects whose correct patches are inside any of the considered search spaces (we run SPR and Prophet for each of the 16 search spaces for each of these 24 defects). Together, these executions consumed over 9000 hours of CPU time on Amazon EC2 cluster. The results show:
  - **Sparse Correct Patches:** Correct patches occur very sparsely within the patch spaces, with typically no more than one correct patch in the search space for a given defect.

- **Relatively Abundant Plausible Patches:** Depending on the strength of the validation test suite, plausible patches are either two to three orders of magnitude or one order of magnitude more abundant than correct patches.
- **Patch Prioritization Effectiveness:** The SPR and Prophet patch prioritization algorithms exhibit substantial effectiveness at isolating correct patches within the large set of plausible patches (most of which are incorrect).
- **Challenges of Rich Search Spaces:** The challenges associated with successfully searching such spaces include increased testing overhead and increased chance of encountering plausible but incorrect patches that block the subsequent discovery of correct patches.

Progress in automatic patch generation systems requires the development of new, larger, and richer patch search spaces that contain correct patches for larger classes of defects. This paper characterizes, for the first time, how current state-of-the-art patch generation systems respond to changes in the size and sophistication of their search spaces. It therefore identifies future productive directions for the field and provides a preview of the challenges that the field will have to overcome to develop systems that work productively with more sophisticated search spaces to successfully patch broader classes of defects.

2. SPR AND PROPHET

We next present an overview of the two automatic patch generation systems, SPR [18] and Prophet [20], whose search spaces and search algorithms we analyze.

2.1 Design Overview

SPR and Prophet start with a defective program to patch and a validation test suite. The test suite contains 1) a set of negative test cases which the original program does not pass (these test cases expose the defect in the program) and 2) a set of positive test cases which the original program already passes (these test cases prevent regression). The test cases include correct outputs for every input (the negative test cases produce different incorrect outputs). The system generates patches for the program with the following steps:

Error Localization: The system first uses an error localizer to identify a set of candidate program statements to modify. The error localizer recompiles the given application with additional instrumentation. It inserts a call back before each statement in the source code to record a positive counter value as the timestamp of the statement execution. The error localizer then invokes the recompiled application on all test cases and produces, based on the recorded timestamp values, a prioritized list of target statements to modify. The error localizer prioritizes statements that are 1) executed with more negative test cases, 2) executed with fewer positive test cases, and 3) executed later during executions with negative test cases. See the SPR and Prophet papers for more details [18, 20].

Apply Transformations: The system then applies a set of transformations to the identified program statements to generate the search space of candidate patches. SPR and Prophet consider the following transformation schemas [18]:

- **Condition Refinement:** Given a target if statement to patch, the system transforms the condition of the if statement by conjoining or disjoining an additional condition to the original if condition. The following two patterns implement the transformation:

  ```
  if (C) { ... } => if (C && P) { ... }
  if (C) { ... } => if (C || P) { ... }
  ```

  Here `if (C) { ... }` is the target statement to patch in the original program. `C` is the original condition that appears in the program. `P` is a new condition produced by a condition synthesis algorithm [18, 20].
- **Condition Introduction:** Given a target statement, the system transforms the program so that the statement executes only if a guard condition is true. The following pattern implements the transformation:

  ```
  S => if (P) S
  ```

  Here `S` is the target statement to patch in the original program and `P` is a new synthesized condition.
- **Conditional Control Flow Introduction:** Before a target statement, the system inserts a new control flow statement (return, break, or goto an existing label) that executes only if a guard condition is true. The following patterns implement the transformation:

  ```
  S => if (P) break; S
  S => if (P) continue; S
  S => if (P) goto L; S
  ```

  Here `S` is the target statement to patch in the original program, `P` is a new synthesized condition, and `L` is an existing label in the procedure containing `S`.
- **Insert Initialization:** Before a target statement, the system inserts a memory initialization statement.
- **Value Replacement:** Given a target statement, replace an expression in the statement with another expression.
- **Copy and Replace:** Given a target statement, the system copies an existing statement to the program point before the target statement and then applies a Value Replacement transformation to the copied statement.

Condition Synthesis: The baseline versions of SPR and Prophet work with synthesized conditions `P` of the form `E == K` and `E != K`. Here `E` is a *check expression*, which we define as either a local variable, a global variable, or a sequence of structure field accesses. Each check expression `E` must appear in the basic block containing the synthesized condition. `K` is a *check constant*, which we define as a constant drawn from the set of values that the check expression `E` takes on during the instrumented executions of the unpatched program on the negative test cases.

Value Replacement: The baseline versions of SPR and Prophet replace either 1) one variable in the target statement with another variable that appears in the basic block containing the statement, 2) an invoked function in the statement with another function that has a compatible type signature and is invoked or declared in the source code file containing the statement (or in an included header file), or 3) a constant in the statement with another constant that appears in the function containing the statement.

Evaluate Candidate Patches: The system then evaluates candidate patches in the search space against the supplied test cases. To efficiently explore the search space, SPR and Prophet use *staged program repair* [18, 20]. At the first stage, the system operates with parameterized candidate patch templates, which may contain an abstract expression. It instantiates and evaluates concrete patches from a template only if the system determines that there may be a concrete patch from the template that passes the test cases.

For the first three transformation schemas (these schemas manipulate conditions), the system first introduces an abstract condition into the program and determines whether there is a sequence of branch directions for the abstract condition that will enable the patched program to pass the test cases. If so, the system then synthesizes concrete conditions to generate patches.

2.2 Extensions

We implement three extensions to the SPR and Prophet search spaces: considering more candidate program statements to patch, synthesizing more sophisticated conditions, and evaluating more complicated value replacement transformations.

More Program Statements to Patch: The baseline SPR and Prophet configurations consider the first 200 program statements identified by the error localizer. We modify SPR and Prophet to consider the first 100, 200, 300, and 2000 statements.

Condition Synthesis Extension (CExt): We extend the baseline SPR and Prophet condition synthesis algorithm to include the "<" and ">" operators and to also consider comparisons between two check expressions (e.g., `E < K`, $\texttt{E}_1$ `==` $\texttt{E}_2$, and $\texttt{E}_1$ `>` $\texttt{E}_2$, where `E`, $\texttt{E}_1$, and $\texttt{E}_2$ are check expressions and `K` is a check constant). In the rest of this paper, we use "CExt" to denote this search space extension.

Value Replacement Extension (RExt): We extend the baseline SPR and Prophet replacement transformations to also replace a variable or a constant in the target statement with an expression that is composed of either 1) a unary operator and an atomic value (i.e., a variable or a constant) which appears in the basic block containing the statement or 2) a binary operator and two such atomic values. The operators that SPR and Prophet consider are "`+`", "`-`", "`*`", "`==`", "`!=`", and "`&`". In the rest of this paper, we use "RExt" to denote this search space extension.

2.3 SPR Prioritization Order

SPR uses a set of hand-coded heuristics to prioritize its search of the generated patch space. These heuristics prioritize patches in the following order: 1) patches that change only a branch condition (e.g., tighten and loosen a condition), 2) patches that insert an if-statement before the first statement of a compound statement (i.e., C code block), 3) patches that insert an if-guard around a statement, 4) patches that replace a statement, insert an initialization statement, insert an if-statement, or insert a statement before the first statement of a compound statement, and 5) finally all the remaining patches. For each kind of patch, it prioritizes statements to patch in the error localization order.

2.4 Prophet Prioritization Order

Prophet searches the same patch space as SPR, but works with a corpus of correct patches from human developers. It processes this corpus to learn a probabilistic model that assigns a probability to each candidate patch in the search space. This probability indicates the likelihood that the patch is correct. It then uses this model to prioritize its search of the patch space.

App.	LoC	Tests	Defects	SPR	Prophet
libtiff	77k	78	8	1/3	2/3
lighttpd	62k	295	7	0/0	0/0
php	1046k	8471	31	9/13	10/13
gmp	145k	146	2	1/1	1/1
gzip	491k	12	4	0/1	1/1
python	407k	35	9	0/0	0/0
wireshark	2814k	63	6	0/0	0/0
fbc	97k	773	2	0/1	1/1
Total			69	11/19	15/19

Table 1: Benchmark Applications

A key idea behind Prophet is that patch correctness depends on not just the patch itself, but also on how the patch interacts with the surrounding code:

- **Extract Features:** For each patch in the corpus, Prophet analyzes a structural diff of the abstract syntax trees of the original and patched code to extract both 1) features which summarize how the patch modifies the program given characteristics of the surrounding code and 2) features which summarize relationships between roles that values accessed by the patch play in the original unpatched program and in the patch.
- **Learn Model Parameters:** Prophet operates with a parameterized log-linear probabilistic model in which the model parameters can be interpreted as weights that capture the importance of different features. Prophet learns the model parameters via maximum likelihood estimation, i.e., the Prophet learning algorithm attempts to find parameter values that maximize the probability of observing the collected training database in the probabilistic model.

Prophet uses the trained model to rank the patches according to its learned model of patch correctness, then evaluates the patches in that order. Previous results (as well as additional results presented in this paper) show that this learned patch correctness model outperforms SPR's heuristics [20]. This result highlights how leveraging information available in existing large software development projects can significantly improve our ability to automatically manipulate large software systems.

3. METHODOLOGY

Benchmark Application: We use a benchmark set of 69 real world defects to perform our search space study. Those defects are from eight large open source applications, libtiff, lighttpd, the php interpreter, gmp, gzip, python, wireshark, and fbc [17]. Note that the original benchmark set also includes 36 ostensible defects which correspond to deliberate functionality changes, not defects, during the application development [18]. We exclude those functionality changes as outside the scope of our study because they are not actual defects.

Table 1 summarizes our benchmark defects. The first column (App.) presents the name of each application. The second column (LoC) presents the number of lines of code in the application. The third column (Tests) presents the number

Defect	Localization Rank	RExt	CExt
lighttpd-2661-2662	1926	No	No
lighttpd-1913-1914	280	No	Yes
python-70056-70059	214	No	Yes
python-69934-69935	136	Yes	No
gmp-14166-14167	226	Yes	No

Table 2: Search Space Extensions

of the test cases in the supplied test suite of the application. php is the outlier, with an order of magnitude more test cases than any other application. The fourth column (Defects) presents the number of defects in the benchmark set for each application.

The fifth column (SPR) and the sixth column (Prophet) present the patch generation results for the baseline versions of SPR [18] and Prophet [20], respectively. Each entry is of the form "X/Y", where Y is the number of defects whose correct patches are inside the search space, while X is the number of defects for which the system automatically generates a correct patch as the first generated patch.

Configure Systems: We run SPR and Prophet on each of the 16 different search space configurations derived from all possible combinations of 1) working with the first 100, 200, 300, or 2000 program statements identified by the error localizer, 2) whether to enable value replacement extension (RExt), and 3) whether to enable condition synthesis extension (CExt).

We run all of our experiments except those of fbc on Amazon EC2 Intel Xeon 2.6GHz machines running Ubuntu-64bit server 14.04. fbc runs only in 32-bit environments, so we run all fbc experiments on EC2 Intel Xeon 2.4GHz machines running Ubuntu-32bit 14.04.

Generate Search Spaces: For each search space and each of the 69 defects in the benchmark set, we run the configured SPR and Prophet to generate and print the search space for that defect. We then analyze the generated search space and determine whether the space contains a correct patch for the defect.

With all three search space extensions, the generated SPR and Prophet search spaces contain correct patches for five more defects (i.e., 24 defects in total) than the baseline search space. Table 2 summarizes these five defects. The first column (Defect) contains entries of the form X-Y-Z, where X is the name of the application that contains the defect, Y is the defective revision in the application repository, and Z is the reference fixed revision in the repository. The second column (Localization Rank) presents the error localization rank of the modified program statement in the correct patch for the defect. The third column (RExt) presents whether the correct patches for the defect require the RExt extension (value replacement extension) The fourth column (CExt) presents whether the correct patches for the defect require the CExt extension (condition synthesis extension).

Generate Patches: For each search space and each of the 24 defects with correct patches in the search space, we run SPR and Prophet to explore the search space for the defect. For each run, we record all of the plausible patches that the system discovers within the 12 hour timeout.

Analyze Patches: For each defect, we analyze the generated plausible patches for the defect to determine whether the patch is correct or incorrect.

4. EXPERIMENTAL RESULTS

We next present the experimental results. php is an outlier with a test suite that contains an order of magnitude more test cases than the other applications. We therefore separate the php results from the results from other benchmarks. We present the result summary for all of the 24 defects for which any of the search spaces contains a correct patch. See our technical report [19] for the detailed results of each defect in all different search space configurations.

Table 3 presents a summary of the results for all of the benchmarks except php. Table 4 presents a summary of the results for the php benchmark. Each row presents patch generation results for SPR or Prophet with one search space configuration. The first column (System) presents the evaluated system (SPR or Prophet). The second column (Loc. Limit) presents the number of considered candidate program statements to patch under the configuration. The third column (Space Extension) presents the transformation extensions that are enabled in the configuration: No (no extensions, baseline search space), CExt (condition synthesis extension), RExt (value replacement extension), or RExt+CExt (both).

The fourth column (Correct In Space) presents the number of defects with correct patches that lie inside the full search space for the corresponding configuration. The fifth column (Correct First) presents the number of defects for which the system finds the correct patch as the *first* patch that validates against the test suite.

Each entry of the sixth column (Plausible & Blocked) is of the form X(Y). Here X is the number of defects for which the system discovers a plausible but incorrect patch as the first patch that validates. Y is the number of defects for which a plausible but incorrect patch blocked a subsequent correct patch (i.e., Y is the number of defects for which 1) the system discovers a plausible but incorrect patch as the first patch that validates and 2) the full search space contains a correct patch for that defect).

Each entry of the seventh column (Timeout) is also of the form X(Y). Here X is the number of defects for which the system does not discover any plausible patch within the 12 hour timeout. Y is the number of defects for which 1) the system does not discover a plausible patch and 2) the full search space contains a correct patch for that defect.

The eighth column (Space Size) presents the average number of candidate patch templates in the search space over all of the 24 considered defects. Note that SPR and Prophet may instantiate multiple concrete patches with the staged program repair technique from a patch template that contains an abstract expression (See Section 2.1). This column shows how the size of the search space grows as a function of the number of candidate statements to patch and the two extensions. Note that the CExt transformation extension does not increase the number of patch templates. Instead it increases the number of concrete patches which each patch template generates. The ninth column (Correct Rank) presents the average rank of the first patch template that generates a correct patch in the search space over all of those defects for which at least one correct patch is inside the search space. Note that the correct rank increases as the size of the search space increases.

Each entry of the tenth column (Plausible in 12h) is of the form X(Y). Here X is the number of defects for which the system discovers a plausible patch within the 12 hour

System	Loc. Limit	Space Extension	Correct		Plausible & Blocked	Timeout	Space Size	Correct Rank	Plausible in 12h	Correct in 12h
			In Space	First						
SPR	100	No	4	1	7(3)	3(0)	20068.5	4614.0	8(2747)	4(5)
SPR	100	CExt	4	1	7(3)	3(0)	20068.5	4614.0	8(11438)	3(4)
SPR	100	RExt	4	1	7(3)	3(0)	21999.8	6004.8	8(2742)	4(5)
SPR	100	RExt+CExt	4	1	7(3)	3(0)	21999.8	6004.8	8(11192)	3(4)
SPR	200	No	6	2	7(4)	2(0)	46377.6	17889.5	9(2558)	6(8)
SPR	200	CExt	6	2	7(4)	2(0)	46377.6	17889.5	9(10823)	4(6)
SPR	200	RExt	7	2	7(4)	2(1)	52864.3	24759.9	9(3753)	6(8)
SPR	200	RExt+CExt	7	2	7(4)	2(1)	52864.3	24759.9	9(10855)	4(6)
SPR	300	No	6	1	8(5)	2(0)	73559.6	22960.0	9(2818)	6(8)
SPR	300	CExt	8	1	8(6)	2(1)	73559.6	30761.8	9(10237)	4(6)
SPR	300	RExt	8	1	8(6)	2(1)	82187.2	32951.4	9(2069)	7(8)
SPR	300	RExt+CExt	10	1	8(7)	2(2)	82187.2	37427.4	9(10455)	5(6)
SPR	2000	No	7	2	7(5)	2(0)	523753.8	157038.4	9(751)	5(6)
SPR	2000	CExt	9	2	7(6)	2(1)	523753.8	156495.1	9(6123)	4(5)
SPR	2000	RExt	9	2	7(6)	2(1)	574325.1	200996.7	9(657)	5(6)
SPR	2000	RExt+CExt	11	2	7(7)	2(2)	574325.1	192034.0	9(5831)	4(5)
Prophet	100	No	4	4	4(0)	3(0)	20068.5	589.2	8(2481)	4(5)
Prophet	100	CExt	4	3	5(1)	3(0)	20068.5	589.2	8(11901)	3(4)
Prophet	100	RExt	4	4	4(0)	3(0)	21999.8	520.5	8(2183)	4(5)
Prophet	100	RExt+CExt	4	3	5(1)	3(0)	21999.8	520.5	8(11595)	3(4)
Prophet	200	No	6	5	4(1)	2(0)	46377.6	11382.8	9(2564)	5(7)
Prophet	200	CExt	6	4	5(2)	2(0)	46377.6	11382.8	9(10968)	4(6)
Prophet	200	RExt	7	5	4(1)	2(1)	52864.3	19581.0	9(1939)	5(6)
Prophet	200	RExt+CExt	7	4	5(2)	2(1)	52864.3	19581.0	9(10928)	4(5)
Prophet	300	No	6	4	5(2)	2(0)	73559.6	11997.2	9(2555)	5(7)
Prophet	300	CExt	8	3	6(4)	2(1)	73559.6	14466.8	9(10948)	4(6)
Prophet	300	RExt	8	4	5(3)	2(1)	82187.2	25769.1	9(1548)	5(6)
Prophet	300	RExt+CExt	10	3	6(5)	2(2)	82187.2	25455.1	9(10886)	4(5)
Prophet	2000	No	7	4	5(3)	2(0)	523753.8	188588.4	9(1229)	5(7)
Prophet	2000	CExt	9	3	6(5)	2(1)	523753.8	156555.8	9(8208)	4(6)
Prophet	2000	RExt	9	3	6(5)	2(1)	574325.1	170715.4	9(1216)	5(6)
Prophet	2000	RExt+CExt	11	2	7(7)	2(2)	574325.1	148288.4	9(7919)	4(5)

Table 3: Patch Generation Results with Search Space Extensions (excluding php)

timeout. Y is the sum, over the all of the 24 considered defects, of the number of plausible patches that the system discovers within the 12 hour timeout.

Each entry of the eleventh column (Correct in 12h) is of the form X(Y). Here X is the number of defects for which the system discovers a correct patch (blocked or not) within the 12 hour timeout. Y is the number of correct patches that the system discovers within the 12 hour timeout.

4.1 Plausible and Correct Patch Density

An examination of the tenth column (Plausible in 12h) in Tables 3 and 4 highlights the overall plausible patch densities in the search spaces. For the benchmarks without php, the explored search spaces typically contain hundreds up to a thousand plausible patches per defect. For php, in contrast, the explored search spaces typically contain tens of plausible patches per defect. We attribute this significant difference in the plausible patch density to the quality of the php test suite and its resulting ability to successfully filter out otherwise plausible but incorrect patches. Indeed, for three php defects, the php test suite is strong enough to filter out all of the patches in the explored search spaces except the correct patch.

An examination of the eleventh column (Correct in 12h) in Tables 3 and 4 highlights the overall correct patch densities in the explored search spaces. In sharp contrast to the plausible patch densities, the explored search spaces contain, on average, less than two correct patches per defect for all of the benchmarks including php. There are five defects with as many as two correct patches in any search space and one defect with as many as four correct patches in any search space. The remaining defects contain either zero or one correct patch across all of the search spaces.

4.2 Search Space Tradeoffs

An examination of the fourth column (Correct In Space) in Table 3 shows that the number of correct patches in the full search space increases as the size of the search space increases (across all benchmarks except php). But an examination of the fifth column (Correct First) indicates that that this increase does not translate into an increase in the ability of SPR or Prophet to actually find these correct patches as the first patch to validate. In fact, the ability of SPR and Prophet to isolate a correct patch as the first patch to validate reaches a maximum at 200 candidate statements with no extensions, then (in general) decreases from there as the size of the search space increases. For php, Table 4 shows that the number of correct patches in the space does not significantly increase with the size of the search space, but that the drop in the number of correct patches found as the first patch to validate is even more significant. Indeed, the 200+No Prophet configuration finds 10 correct patches as the first patch to validate, while the largest 2000+RExt+CExt configuration finds only four!

We attribute these facts to an inherent tradeoff in the search spaces. Expanding the search spaces to include more correct patches also includes more implausible and plausible but incorrect patches. The implausible patches consume

System	Loc. Limit	Space Extension	Correct		Plausible & Blocked	Timeout	Space Size	Correct Rank	Plausible in 12h	Correct in 12h
			In Space	First						
SPR	100	No	12	8	3(3)	2(1)	13446.5	4157.8	11(237)	9(14)
SPR	100	CExt	12	8	4(4)	1(0)	13446.5	4157.8	12(415)	10(12)
SPR	100	RExt	12	8	4(4)	1(0)	14026.7	4360.8	12(288)	11(15)
SPR	100	RExt+CExt	12	8	4(4)	1(0)	14026.7	4360.8	12(421)	10(12)
SPR	200	No	13	9	3(3)	1(1)	26512.0	7369.8	12(197)	10(15)
SPR	200	CExt	13	9	3(3)	1(1)	26512.0	7369.8	12(330)	10(13)
SPR	200	RExt	13	9	3(3)	1(1)	28158.2	7984.5	12(200)	10(15)
SPR	200	RExt+CExt	13	9	3(3)	1(1)	28158.2	7984.5	12(323)	10(13)
SPR	300	No	13	8	4(4)	1(1)	41859.8	10915.2	12(176)	9(14)
SPR	300	CExt	13	8	4(4)	1(1)	41859.8	10915.2	12(305)	9(12)
SPR	300	RExt	13	8	4(4)	1(1)	44631.1	12440.9	12(179)	9(14)
SPR	300	RExt+CExt	13	8	4(4)	1(1)	44631.1	12440.9	12(313)	9(12)
SPR	2000	No	13	5	2(2)	6(6)	327905.6	81570.5	7(58)	5(6)
SPR	2000	CExt	13	5	2(2)	6(6)	327905.6	81570.5	7(126)	5(6)
SPR	2000	RExt	13	5	3(3)	5(5)	356104.8	83997.9	8(59)	5(6)
SPR	2000	RExt+CExt	13	5	2(2)	6(6)	356104.8	83997.9	7(127)	5(6)
Prophet	100	No	12	8	3(3)	2(1)	13446.5	2599.4	11(279)	11(15)
Prophet	100	CExt	12	6	6(6)	1(0)	13446.5	2599.4	12(466)	11(11)
Prophet	100	RExt	12	9	3(3)	1(0)	14026.7	3433.8	12(327)	11(15)
Prophet	100	RExt+CExt	12	6	6(6)	1(0)	14026.7	3433.8	12(458)	11(11)
Prophet	200	No	13	10	3(3)	0(0)	26512.0	3522.1	13(285)	13(18)
Prophet	200	CExt	13	7	6(6)	0(0)	26512.0	3522.1	13(447)	12(13)
Prophet	200	RExt	13	10	3(3)	0(0)	28158.2	4504.4	13(296)	12(17)
Prophet	200	RExt+CExt	13	7	6(6)	0(0)	28158.2	4504.4	13(434)	12(13)
Prophet	300	No	13	10	3(3)	0(0)	41859.8	4319.6	13(280)	13(18)
Prophet	300	CExt	13	7	6(6)	0(0)	41859.8	4319.6	13(425)	12(13)
Prophet	300	RExt	13	10	3(3)	0(0)	44631.1	5403.1	13(283)	12(17)
Prophet	300	RExt+CExt	13	7	6(6)	0(0)	44631.1	5403.1	13(422)	12(13)
Prophet	2000	No	13	7	2(2)	4(4)	327905.6	21118.6	9(117)	7(10)
Prophet	2000	CExt	13	4	4(4)	5(5)	327905.6	21118.6	8(153)	6(6)
Prophet	2000	RExt	13	6	2(2)	5(5)	356104.8	25168.5	8(104)	6(9)
Prophet	2000	RExt+CExt	13	4	4(4)	5(5)	356104.8	25168.5	8(183)	6(6)

Table 4: Patch Generation Results with Search Space Extensions (php only)

validation time (extending the time required to find the correct patches), while the plausible but incorrect patches block the correct patches. This trend is visible in the Y entries in the sixth column in Table 3 (Plausible & Blocked) (these entries count the number of blocked correct patches), which generally increase as the size of the search space increases.

Tables 3 and 4 show how this tradeoff makes the baseline SPR and Prophet configurations perform best despite working with search spaces that contain fewer correct patches. Increasing the candidate statements beyond 200 never increases the number of correct patches that are first to validate. Applying the CExt and RExt extensions also never increases the number of correct patches that are first to validate.

Our results highlight two challenges that SPR and Prophet (and other generate and validate systems) face when generating correct patches:

- **Weak Test Suites:** The test suite provides incomplete coverage. The most obvious problem of the weak test suite is that it may accept incorrect patches. Our results show that (especially for larger search spaces) plausible but incorrect patches often block correct patches. For example, when we run Prophet with the baseline search space (200+No), there are only 4 defects whose correct patches are blocked; when we run Prophet with the largest search space (2000+RExt+CExt), there are 11 defects whose correct patches are blocked.

  A more subtle problem is that weak test suites may increase the validation cost of plausible but incorrect patches. For such a patch, SPR or Prophet has to run the patched application on all test cases in the test suite. If a stronger test suite is used, SPR and Prophet may invalidate the patch with one test case and skip the remaining test cases.

- **Search Space Explosion:** A large search space contains many candidate patch templates and our results show that it may be intractable to validate all of the candidates. For example, with the baseline search space (200+No), Prophet times out for only two defects (whose correct patches are outside the search space); with the largest evaluated search space (2000+RExt+CExt), Prophet times out for seven defects (whose correct patches are inside the search space).

Note that many previous systems [17, 40, 14, 28] neglect the weak test suite problem and do not evaluate whether the generated patches are correct or not. In contrast, our results show that the weak test suite problem is at least as important as the search space explosion problem. In fact, for all evaluated search space configurations, there are more defects for which SPR or Prophet generates plausible but incorrect patches than for which SPR or Prophet times out.

4.3 SPR and Prophet Effectiveness

We compare the effectiveness of the SPR and Prophet patch prioritization orders by measuring the costs and pay-

Search Space	**SPR**	**Prophet**	**Random (SPR)**	**Random (Prophet)**
100+No	52 / 3	38 / 4	65.0 / 1.4	65.0 / 1.4
100+CExt	65 / 2	53 / 3	73.0 / 1.0	73.0 / 1.0
100+RExt	52 / 3	38 / 4	65.1 / 1.4	65.1 / 1.4
100+RExt+CExt	65 / 2	53 / 3	73.0 / 1.0	73.0 / 1.0
200+No	59 / 4	45 / 5	73.5 / 2.7	76.2 / 2.1
200+CExt	66 / 3	54 / 4	78.1 / 1.7	78.1 / 1.7
200+RExt	59 / 4	45 / 5	76.2 / 2.1	75.9 / 2.1
200+RExt+CExt	66 / 3	54 / 4	77.8 / 1.7	77.8 / 1.7
300+No	60 / 5	50 / 5	80.2 / 2.0	78.2 / 2.1
300+CExt	75 / 2	63 / 3	83.9 / 1.3	83.2 / 1.4
300+RExt	62 / 4	50 / 5	81.4 / 1.4	79.9 / 2.1
300+RExt+CExt	75 / 2	63 / 3	85.2 / 1.1	84.4 / 1.2
2000+No	56 / 4	50 / 5	78.9 / 1.3	77.8 / 2.3
2000+CExt	72 / 2	63 / 3	86.6 / 0.7	83.2 / 1.4
2000+RExt	60 / 3	51 / 5	74.7 / 1.7	78.5 / 2.1
2000+RExt+CExt	72 / 2	64 / 3	82.6 / 1.1	84.4 / 1.3

Table 5: Costs and Payoffs of Reviewing the First 10 Generated Patches (excluding php)

Search Space	**SPR**	**Prophet**	**Random (SPR)**	**Random (Prophet)**
100+No	38 / 9	37 / 9	45.8 / 8.1	44.7 / 8.4
100+CExt	48 / 9	56 / 9	66.7 / 6.8	66.8 / 6.8
100+RExt	39 / 10	39 / 10	50.9 / 8.8	51.4 / 8.9
100+RExt+CExt	46 / 9	57 / 9	66.9 / 6.8	66.7 / 6.8
200+No	39 / 10	39 / 11	47.7 / 9.1	49.3 / 10.4
200+CExt	39 / 10	57 / 11	59.7 / 7.6	68.1 / 7.9
200+RExt	39 / 10	40 / 11	47.8 / 9.1	51.6 / 10.1
200+RExt+CExt	39 / 10	58 / 11	59.6 / 7.6	68.0 / 7.9
300+No	32 / 9	39 / 11	43.8 / 8.1	50.2 / 10.4
300+CExt	32 / 9	57 / 11	59.3 / 6.4	72.8 / 7.9
300+RExt	34 / 9	40 / 11	45.8 / 8.1	51.5 / 10.2
300+RExt+CExt	34 / 9	58 / 11	61.3 / 6.4	69.5 / 8.0
2000+No	7 / 5	25 / 7	16.7 / 4.4	37.4 / 6.3
2000+CExt	7 / 5	34 / 5	29.8 / 2.9	46.4 / 4.0
2000+RExt	9 / 5	17 / 6	18.7 / 4.4	29.3 / 5.3
2000+RExt+CExt	7 / 5	27 / 5	29.8 / 2.9	47.4 / 3.2

Table 6: Costs and Payoffs of Reviewing the First 10 Generated Patches (php only)

offs for a human developer who reviews the generated patches to find a correct patch. We consider a scenario in which the developer reviews the first 10 generated patches one by one for each defect until he finds a correct patch. He gives up if none of the first 10 patches are correct. For each system and each search space configuration, we compute (over the 24 defects that have correct patches in the full SPR and Prophet search space) 1) the total number of patches the developer reviews (this number is the cost) and 2) the total number of defects for which the developer obtains a correct patch (this number is the payoff). We also compute the expected costs and payoffs if the developer examines the generated plausible SPR and Prophet patches in a random order. See our technical report [19] for the raw data used to compute these numbers.

Tables 5 and 6 present these costs and payoffs. The first column presents the search space configuration. The second and third columns present the costs and payoffs for the SPR and Prophet patch prioritization orders; the fourth and fifth columns present the corresponding costs and payoffs for the random orders. Each entry is of the form X/Y, where X is the total number of patches that the developer reviews and Y is the total number of defects for which he obtains a correct patch. These numbers highlight the effectiveness of the SPR and Prophet patch prioritization in identifying the few correct patches within the many plausible but incorrect patches.

Table 6 presents the corresponding results for the php defects. These numbers highlight the difference that a stronger test suite can make in the success of finding correct patches. The correct patch selection probabilities are dramatically higher for php than for the other benchmarks. But note that as the patch search spaces become large, the number of defects for which the developer obtains correct patches become smaller, reflecting 1) the increasing inability of the systems to find any correct patch in the explored space within the 12 hour timeout and 2) the increasing presence of blocking plausible but incorrect patches.

Finally, these numbers highlight the effectiveness of the Prophet learned patch prioritization — following this procedure, the developer always obtains correct patches for at least as many defects with Prophet as with SPR.

5. THREATS TO VALIDITY

This paper presents a systematic study of search space tradeoffs with SPR and Prophet. One threat to validity is that our results will not generalize to other benchmark sets and other patch generation systems. Note that the benchmark set was developed by other researchers, not by us, with the goal of obtaining a large, unbiased, and realistic benchmark set [17]. And this same benchmark set has been used to evaluate many previous patch generation systems [17, 40, 28, 29, 18]. The observations in this paper are consistent with previous results reported for other systems on this benchmark set [29, 17, 40, 28].

Another threat to validity is that stronger test suites will become the norm so that the results for benchmark applications other than php will not generalize to other applications. We note that 1) comprehensive test coverage is widely considered to be beyond reach for realistic applications, and 2) even php, which has by far the strongest test suite in the set of benchmark applications, has multiple defects for which the number of plausible patches exceeds the number of correct patches by one to two orders of magnitude.

6. RELATED WORK

ClearView: ClearView is a generate-and-validate system that observes normal executions to learn invariants that characterize safe behavior [27]. It deploys monitors that detect crashes, illegal control transfers and out of bounds write defects. In response, it selects a nearby invariant that the input that triggered the defect violates, and generates patches that take a repair action to enforce the invariant.

A Red Team evaluation found that ClearView was able to automatically generate patches that eliminate 9 of 10 targeted Firefox vulnerabilities [27], with each defect eliminated after the generation of at most three patches. We attribute the density with which successful patches appear in the ClearView search space, in part, to the fact that ClearView leverages the learned invariant information to focus the search on successful patches and does not rely solely on the validation test suite.

Kali: Kali is a generate-and-validate system that deploys a simple strategy — it simply removes functionality. Although Kali is obviously not intended to correctly repair a reasonable subset of the defects that occur in practice, it is nevertheless at least as effective in practice as previous strategies that aspire to repair a broad class of defects [29].

These results are consistent with the results presented in this paper, highlight the inadequacy of current test suites to successfully filter incorrect patches, and identify one prominent source of the relatively many plausible but incorrect patches that occur in current patch search spaces.

An analysis of Kali remove statement patches, GenProg patches, and NOPOL [4] patches for 224 defects in the Defects4J dataset [11] produced results broadly consistent with the results in this paper: out of 42 manually analyzed plausible patches, the analysis indicates that only 8 patches are undoubtedly correct [7].

GenProg, AE, and RSRepair: GenProg [17], AE [40], and RSRepair [28] were all evaluated on (for RSRepair, a subset of) the same benchmark set that we use in this paper to evaluate the SPR and Prophet search spaces. The evaluations focus on plausible patches with no attempt to determine whether the patches are correct or not. Unfortunately, the presented evaluations of these systems suffer from the fact that the testing infrastructure used to validate the candidate patches contains errors that cause the systems to incorrectly accept implausible patches that do not even pass all of the test cases in the validation test suite [29].

A subsequent study corrects these errors and sheds more light on the subject [29]. This study found that 1) the systems generate correct patches for only 2 (GenProg, RSRepair) or 3 (AE) of the 105 benchmark defects/functionality changes in this benchmark set, 2) the systems generate plausible but incorrect patches for 16 (GenProg), 8 (RSRepair), and 24 (AE) defects/functionality changes, and 3) the majority of the plausible patches, including all correct patches, are equivalent to a single modification that deletes functionality. Moreover, the correct SPR and Prophet patches for these defects lie outside the GenProg, AE, and RSRepair search space (suggesting that these systems will never be able to generate a correct patch for these defects) [18].

Moreover, only 5 of the 110 plausible GenProg patches are correct, only 4 of the analyzed plausible 44 RSRepair patches are correct, and only 3 of the 27 plausible AE patches are correct [29]. These results indicate that the GenProg, AE, and RSRepair search space, while containing fewer correct and plausible patches than the richer SPR and Prophet search spaces [18, 20], still exhibits the basic pattern of sparse correct patches and more abundant plausible but incorrect patches.

A subsequent study of GenProg and RSRepair (under the name TrpAutoRepair) on small student programs provides further support for this hypothesis [37]. The results indicate that patches validated on one test suite typically fail to generalize to produce correct results on other test suites.

RSRepair uses random search; previous research found that the GenProg genetic search algorithm performs no better than random search on a subset of the benchmarks [28]. Systems (such as GenProg) that perform no better than random will need to incorporate additional sources of information other than the validation test suite if they are to successfully generate correct patches in the presence of current relatively weak test suites.[3]

CodePhage: Horizontal code transfer repairs otherwise fatal defects by transferring correct code across applications [35]. It therefore provides another example of how leveraging additional information outside the validation test suite (in this case, correct code from other applications) promotes the automatic generation of successful patches. But of course horizontal code transfer is not limited to patch generation — indeed, it shows enormous potential for leveraging the combined talents and labor of software development efforts worldwide to solve a variety of software engineering problems.

Specifications: Explicit specifications (either provided by a developer or inferred) provide an alternative to validation test suites. Researchers have built repair systems that leverage data structure consistency specifications [5, 6, 8], method contracts [39, 26], access control specifications [38], assertions [33], and pre- and post-conditions [16].

Other Related Work: See the full version of this paper [19] for a more comprehensive discussion of related work including 1) dynamic program repair techniques for memory errors [32], null pointer dereference and divide by zero errors [21], infinite loops [2, 15], and memory leaks [24], 2) additional patch generation systems [4, 7, 14, 23, 12, 9, 39, 26, 3, 16, 25, 10, 34], 3) studies of human patch characteristics [1, 22, 41], and 4) task skipping and loop perforation, which can improve performance and enable programs to survive tasks or loop iterations that trigger otherwise fatal errors [30, 31, 36].

7. CONCLUSION

The scope of any automatic patch generation system is limited by the range of patches in its search space — to repair more defects, future systems will need to work with search spaces that contain more correct patches. This paper characterizes, for the first time, how larger and richer search spaces that contain more correct patches can, counterintuitively, hamper the ability of the system to find correct patches. It therefore identifies a key challenge that designers of future systems must overcome for their systems to successfully generate patches for broader classes of defects.

Experience with previous successful patch generation systems such as ClearView, Prophet, and CodePhage highlights how leveraging information outside the validation test suite enables these systems to successfully identify the few correct patches within the many plausible patches (most of which are incorrect) that the test suite validates. We anticipate that future successful automatic patch generation systems will deploy even more sophisticated techniques that leverage the full range of available information (test suites, previous successful patches, documentation, even formal specifications) to successfully identify the correct patches available in larger, richer patch search spaces.

Acknowledgements

We thank the anonymous reviewers for their insightful comments on early draft of the paper. We thank Yu Lao for logistics support. This research was supported by DARPA (Grant FA8650-11-C-7192 and FA8750-14-2-0242).

[3]Of course, if the patch space does not contain correct patches, stronger test suites will prevent the system from generating any patches at all. This is the case for GenProg — at most two additional test cases per defect completely disable GenProg's ability to produce any patch at all except the five correct patches in its search space [29].

8. REFERENCES

[1] E. T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro. The Plastic Surgery Hypothesis. In *Proceedings of the 22nd ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, pages 306–317, Hong Kong, China, November 2014.

[2] M. Carbin, S. Misailovic, M. Kling, and M. C. Rinard. Detecting and escaping infinite loops with jolt. In *Proceedings of the 25th European conference on Object-oriented programming*, ECOOP'11, pages 609–633. Springer-Verlag, 2011.

[3] V. Debroy and W. E. Wong. Using mutation to automatically suggest fixes for faulty programs. In *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, pages 65–74. IEEE, 2010.

[4] F. DeMarco, J. Xuan, D. Le Berre, and M. Monperrus. Automatic repair of buggy if conditions and missing preconditions with smt. In *Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis*, CSTVA 2014, pages 30–39, New York, NY, USA, 2014. ACM.

[5] B. Demsky and M. C. Rinard. Automatic detection and repair of errors in data structures. In *Proceedings of the 2003 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2003, October 26-30, 2003, Anaheim, CA, USA*, pages 78–95, 2003.

[6] B. Demsky and M. C. Rinard. Goal-directed reasoning for specification-based data structure repair. *IEEE Trans. Software Eng.*, 32(12):931–951, 2006.

[7] T. Durieux, M. Martinez, M. Monperrus, R. Sommerard, and J. Xuan. Automatic repair of real bugs: An experience report on the defects4j dataset. *CoRR*, abs/1505.07002, 2015.

[8] B. Elkarablieh, I. Garcia, Y. L. Suen, and S. Khurshid. Assertion-based repair of complex data structures. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07', pages 64–73, New York, NY, USA, 2007. ACM.

[9] Q. Gao, H. Zhang, J. Wang, Y. Xiong, L. Zhang, and H. Mei. Fixing recurring crash bugs via analyzing q&a sites (T). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, pages 307–318, 2015.

[10] D. Gopinath, S. Khurshid, D. Saha, and S. Chandra. Data-guided repair of selection statements. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 243–253, New York, NY, USA, 2014. ACM.

[11] R. Just, D. Jalali, and M. D. Ernst. Defects4j: a database of existing faults to enable controlled testing studies for java programs. In *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, pages 437–440, 2014.

[12] S. Kaleeswaran, V. Tulsian, A. Kanade, and A. Orso. Minthint: Automated synthesis of repair hints. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 266–276, New York, NY, USA, 2014. ACM.

[13] Y. Ke, K. T. Stolee, C. Le Goues, and Y. Brun. Repairing programs with semantic code search (T). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, pages 295–306, 2015.

[14] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13', pages 802–811. IEEE Press, 2013.

[15] M. Kling, S. Misailovic, M. Carbin, and M. Rinard. Bolt: on-demand infinite loop escape in unmodified binaries. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '12', pages 431–450. ACM, 2012.

[16] E. Kneuss, M. Koukoutos, and V. Kuncak. Deductive program repair. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*, pages 217–233, 2015.

[17] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 3–13. IEEE Press, 2012.

[18] F. Long and M. Rinard. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, pages 166–178, 2015.

[19] F. Long and M. Rinard. An Analysis of the Search Spaces for Generate and Validate Patch Generation Systems. Technical Report MIT-CSAIL-TR-2016-003, 2016.

[20] F. Long and M. Rinard. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2016, pages 298–312, New York, NY, USA, 2016. ACM.

[21] F. Long, S. Sidiroglou-Douskos, and M. Rinard. Automatic runtime error repair and containment via recovery shepherding. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14', pages 227–238, New York, NY, USA, 2014. ACM.

[22] M. Martinez, W. Weimer, and M. 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.

[23] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. Semfix: Program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13', pages 772–781, Piscataway, NJ, USA, 2013. IEEE Press.

[24] H. H. Nguyen and M. C. Rinard. Detecting and eliminating memory leaks using cyclic memory

allocation. In *Proceedings of the 6th International Symposium on Memory Management, ISMM 2007, Montreal, Quebec, Canada, October 21-22, 2007*, pages 15–30, 2007.

[25] F. S. Ocariza, Jr., K. Pattabiraman, and A. Mesbah. Vejovis: Suggesting fixes for javascript faults. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 837–847, New York, NY, USA, 2014. ACM.

[26] Y. Pei, C. A. Furia, M. Nordio, Y. Wei, B. Meyer, and A. Zeller. Automated fixing of programs with contracts. *IEEE Trans. Software Eng.*, 40(5):427–449, 2014.

[27] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard. Automatically patching errors in deployed software. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 87–102. ACM, 2009.

[28] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang. The strength of random search on automated program repair. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 254–265, New York, NY, USA, 2014. ACM.

[29] Z. Qi, F. Long, S. Achour, and M. C. Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*, pages 24–36, 2015.

[30] M. C. Rinard. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. In *Proceedings of the 20th Annual International Conference on Supercomputing, ICS 2006, Cairns, Queensland, Australia, June 28 - July 01, 2006*, pages 324–334, 2006.

[31] M. C. Rinard. Using early phase termination to eliminate load imbalances at barrier synchronization points. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, pages 369–386, 2007.

[32] M. C. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and W. S. Beebee. Enhancing server availability and security through failure-oblivious computing. In *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, pages 303–316, 2004.

[33] R. Samanta, O. Olivo, and E. A. Emerson. Cost-aware automatic program repair. In *Static Analysis - 21st International Symposium, SAS 2014, Munich, Germany, September 11-13, 2014. Proceedings*, pages 268–284, 2014.

[34] H. Samimi, M. Schäfer, S. Artzi, T. Millstein, F. Tip, and L. Hendren. Automated repair of html generation errors in php applications using string constraint solving. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12', pages 277–287, Piscataway, NJ, USA, 2012. IEEE Press.

[35] S. Sidiroglou-Douskos, E. Lahtinen, F. Long, and M. Rinard. Automatic error elimination by horizontal code transfer across multiple applications. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 43–54, 2015.

[36] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. C. Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13rd European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*, pages 124–134, 2011.

[37] E. K. Smith, E. Barr, C. Le Goues, and Y. Brun. Is the Cure Worse than the Disease? Overfitting in Automated Program Repair. In *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 532–543, Bergamo, Italy, September 2015.

[38] S. Son, K. S. McKinley, and V. Shmatikov. Fix me up: Repairing access-control bugs in web applications. In *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013*, 2013.

[39] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller. Automated fixing of programs with contracts. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10', pages 61–72, New York, NY, USA, 2010. ACM.

[40] W. Weimer, Z. P. Fry, and S. Forrest. Leveraging program equivalence for adaptive program repair: Models and first results. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, pages 356–366, 2013.

[41] H. Zhong and Z. Su. An empirical study on real bug fixes. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, pages 913–923, 2015.