Developing search spaces for automatic repair of vulnerabilities in mobile software*

Alexey Zhikhartsev, Zijin Li
University of Waterloo
200 University Ave W, Waterloo, ON, Canada
{azhikhar, z542li}@waterloo.ca

ABSTRACT

Using automatic repair tools to fix security vulnerabilities and other bugs can potentially save companies great amount of resources. However, current Generate-and-validate automatic repair approaches often produce incorrect patches due to imperfections in their search space; prior work suggests to tackle this problem by creating "targeted" search spaces—the search spaces aimed at fixing specific types of vulnerabilities or aimed at particular domains. In this work, we conducted a manual examination of 430 Android vulnerabilities to discover how many of those vulnerabilities can be potentially fixed by existing tools (GenProg and SPR). Out of total 430 vulnerabilities, 59 are in the search space of GenProg and 24 are in SPR's. To improve the search space of SPR, we proposed a new search space that contains 44 correct patches and has a more promising prioritization of transformations.

1. INTRODUCTION

In the world of countless software bugs and limited resources to fix them, automatic software repair tools are a desirable instrument that can decrease the developers' workload. Current automatic repair tools aim at fixing simpler bugs and thus give developers a chance to concentrate on more sophisticated bugs. In addition, developers' time is highly valuable from the economic point of view; thus, automatic repair tools should significantly decrease the expenses associated with software maintenance. Ideally, the process of bug-fixing should be automated: an automatic repair tool receives a buggy program and outputs a fix for this bug.

One of the most important subsets of all bugs is software vulnerabilities, as they can lead to substantial financial losses and great customers' dissatisfaction; among the most notorious examples are Slammer and Morris worms [14, 17]. In addition, software vulnerabilities on mobile phones are of a particular importance due to the ubiquity of mobile apps and a great amount of private data on mobile phones, which is often undervalued [3].

Several automatic software repair tools were proposed [7, 10, 13] that are able to fix some subsets of bugs (including software vulnerabilities). One of the most common approaches is the so-called Generate-and-validate (G&V) approach, in which the target project has a bug that is exposed in some failing tests; the automatic repair tool goes through the list of potential fixes (*the search space*), applies the fix and runs the tests; if all the tests pass, then the fix is considered to be correct and it is presented to the developer. Otherwise, the tool picks the next fix and runs the tests again.

Along with the test cases that expose the bug, a G&V tool also accepts the test cases that verify the existing functionality to avoid regressions. The imperfection of a test suite can lead to *overfitted* patches—patches that make all the test cases pass but nonetheless are incorrect. Analogically to the concept of overfitness in machine learning, where classifiers may be overfitted to training data and perform poorly on new instances, these incorrect patches overfit existing test suites, but newly added test cases can expose defects in these patches. For example, consider a bug that is exposed through a test case; however, there is no test case that would verify the *functionality*, in which the bug resides. The simplest patch would just remove the functionality altogether and thus would make all the test cases pass. Obviously, such a patch is incorrect and would not be accepted by the developer.

Along with the mentioned problem of poor test suites, there is the problem of poor search spaces: if a search space does not contain a correct patch, then it will never be found. By constructing richer search spaces (that contain more correct patches), automatic repair tool may be able to find more correct patches. However, Long et al. [11] showed that, in the search spaces of G&V automatic repair tools, overfitted patches dominate over correct ones by orders of magnitude. In addition, they discovered that richer search spaces lead to *fewer* correct patches found! Based on that, Long et al. outlined a need in more precise search spaces—the search spaces that would contain as many correct patches and as few overfitted patches as possible.

In our work, we aimed at creating search spaces that are precise in two dimensions: (1) they target *mobile systems* and (2) they target *security vulnerabilities* specifically. The project consisted of the following stages: (1) mining vulnerabilities of mobile software from open-source repositories; (2) manual analysis of the retrieved patches; (3) devising the search spaces that contain correct patches for the analyzed vulnerabilities; and (4) evaluating the search spaces in terms of how many correct patches they contain. The result of our project is a search space for automatic repair tools that aim at repairing mobile vulnerabilities. To the best of our knowledge, there are no prior work of building G&V automatic repair tools specifically for fixing *vulnerabilities* in *mobile* software; this project will be the first step in building such a tool.

*This work is a research project for CS 858—Mobile Privacy and Security (MoPS), Fall 2016

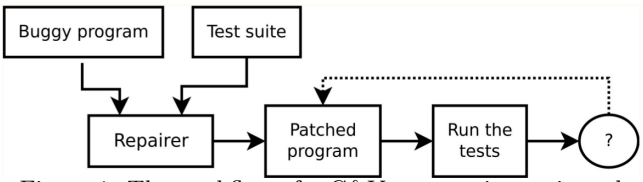


Figure 1: The workflow of a G&V automatic repair tool

The rest of the paper is structured as follows: Section 2 provides the reader with a background on automatic software repair; Section 3 describes the process of obtaining developer patches for security vulnerabilities in Android; the empirical study that assess search spaces of previous automatic repair tools based on data obtained is shown in Section 4; in Section 5, we propose a new search space that targets the examined vulnerabilities; Sections 6 and 7 respectively provide threats to validity and related work; Section 8 concludes.

2. BACKGROUND

In this section, we provide the background information on automatic software repair and present an example of fixing a software vulnerability automatically.

2.1 Automatic software repair

The term "automatic software repair" covers many different approaches which accept a buggy program as an input and produce a fix for the target bug as an output. In this section, we will focus only on G&V automatic repair (e.g., GenProg [7], SPR [10], Prophet [9]); we will be using the terms G&V automatic repair and automatic repair interchangeably. The typical workflow of a G&V automatic repair tool is depicted in Figure 1.

Along with a buggy program, a G&V tool also accepts two types of test cases:

- **Positive** test cases that verify the program's existing functionality and
- **Negative** test cases that expose the *target* bug (i.e., the bug that the tool aims to fix)

The first step is the stage of *fault localization* that produces a ranked list of statements that are suspected to be buggy. Fault localization is a well-established area of Software engineering [18] with many different approaches; the approach adopted by most automatic repair tools is as follows: instrument each statement in the buggy program and run the program with the provided test cases. Then, based on some metric, order the statements from the most likely to be buggy to the least likely. E.g., SPR's fault localization algorithm prefers the statements that are (1) executed with more negative test cases (2) fewer positive test cases and (3) encountered later whilst executing the program with negative test cases.

After the ordered list of buggy statements is produced, the automatic repair tool applies its *transformations* to attempt to fix the target bug. These transformations depend on a search space that is employed by a particular tool. To verify the correctness of the fix applied, the repairer runs all the test cases: negative ones (to verify that the bug is gone) and positive ones (to verify that no regressions are introduced). If the test cases pass, then the patch is considered to be correct and it is presented to the developer; otherwise, a repairer picks another fix from the search space and repeats the previous steps.

2.2 GenProg

GenProg [7] is an automatic repair tool based on genetic programming (GP); it randomly creates candidate fixes and refines them through GP—by exchanging, adding or removing statements—until one of the fixes passes all the test cases. The main assumption that GenProg makes is that the program already has correct fixes for the target bug somewhere in the program; the goal is to find the fixes, order them properly, and apply them in the correct location. Thus, if the program does not contain "ingredients" for the correct fix, then the target bug cannot be repaired. In addition, Qi et al. [16] showed that, for the benchmarks GenProg was evaluated on, most of the fixes correspond to simple functionality deletions; they do not fix the target bug but rather just delete the functionality in which the bug resides.

2.3 SPR

SPR [10] uses a notion of a so-called "schema", i.e., a patch template; putting different conditions into different schemas leads to a large search space with many correct patches. Contrarily to GenProg, SPR's search spaces are deterministic; SPR applies its transformations in a certain order (the order is hard-coded) until either all the test cases pass or the search space is exhausted. Next, we present the transformations in the order that SPR applies them (Figure 2 shows examples from GenProg benchmarks on which SPR was evaluated).

Condition refinement. Transform an existing if-condition by adding "`&& P`" or "`|| P`"; `P` is a new condition of the form "`(v == const)`" or "`(v != const)`", where `v` is an existing variable in the current scope. *Note:* degenerate cases that either remove the whole if-block (`&& 0`) or always execute the if-block (`|| 1`) are possible as well.

Condition introduction. Transform an existing statement by adding an if-condition around it.

Conditional control flow introduction. Put a conditional control-flow statement (e.g., `break`, `continue`, `return`) before a statement.

Insert initialization. Put a call to `memset` before a statement.

Value replacement can either (1) replace a variable with another variable, (2) replace a function with another function that has the same signature or (3) replace a constant with another constant.

Copy and replace. Copy an existing statement before the target statement and apply **value replacement**.

2.4 Example of fixing a vulnerability automatically

Next, we present an example of fixing a simple buffer overflow vulnerability automatically. In Figure 3, there is an incorrect check at line 2 with the common "off-by-one" mistake that leads to a value being written past the end of the buffer; line 3 shows a probable human-written fix.

Let us say that we are given a positive test case that verifies the correct value in the buffer at the index `x` (e.g., `assert(buf[x] == 0)`) and a negative test case that exposes the overflow (e.g., by making the initial value of `x` equal to `sizeof(buf)` and exposing the overflow with address sani-

```
chunk *c;

- if (len == 0)
+ if ((len == 0) || (len == 3))
    return 0;

c = chunkpool_get_unused_chunk();
```

(a) Condition refinement from lighttpd-1913-1914

```
 }
else
 {
+ if (!(crop->img_mode == 0))
     if (...) {
         TIFFError("computeInputPixelOffsets", ...);
         TIFFError("computeInputPixelOffsets", ...);
```

(b) Condition introduction from libtiff-5b02179-3dfb33b

```
if (Z_LVAL_P(dim) < 0 || ... ) {
+   if ((type != 0))
+       return;
    zend_error( ... );
    Z_STRVAL_P(ptr) = STR_EMPTY_ALLOC();
    Z_STRLEN_P(ptr) = 0;
```

(c) Conditional control-flow from php-308262-308315

```
dateobj->time->y = y;
dateobj->time->m = 1;
dateobj->time->d = 1;
+ memset(&dateobj->time->relative, 0, sizeof( ... );
dateobj->time->relative.d = ... ;
dateobj->time->have_relative = 1;
```

(d) "Insert initialization" from php-307846-307853

```
        RETURN_FALSE;
}

- htmlNodeDumpFormatOutput(buf, docp, node, 0, fmt);
+ xmlNodeDump(buf, docp, node, 0, fmt);
mem = (xmlChar*) xmlBufferContent(buf);
if (!mem) {
        RETVAL_FALSE;
```

(e) Value replacement from php-307562-307561

```
int i;

+ (json_globals.error_code) = PHP_JSON_ERROR_UTF8;
if (options & PHP_JSON_PRETTY_PRINT) {
    for (i = 0; i < JSON_G(encoder_depth); ++i) {
        smart_str_appendl(buf, "    ", 4);
```

(f) Copy-and-replace from php-308525-308529

Figure 2: Examples of SPR transformations

tization[1]). The simplest patch that an automatic tool can generate is a mere functionality deletion; in the absence of line 5, the negative test case would stop failing; however, a positive test case would fail and reveal an incorrect fix. Let us say that after a number of failed attempts, the repairer creates the fix shown at line 4; the positive test case passes (since `buf[x] == 0`), as well as the negative one (no overflow). This patch is given to the developer as the correct patch and it is indeed semantically equivalent to the probable developer patch at line 3. Note: under the SPR's transformation classification, this fix belongs to "condition refinement" and indeed can be potentially generated by SPR.

3. MINING ANDROID VULNERABILITIES

In this section, we specify what data is needed for our study, how we obtained the Android vulnerability data and finally describe the data acquired.

[1]https://github.com/google/sanitizers/wiki/AddressSanitizer

```
int buf[3] = {1, 1, 1};
- if (x <= sizeof(buf)) // bug
+ if (x < sizeof(buf)) // developer fix
+ if (x <= sizeof(buf) && x != sizeof(buf)) // SPR fix
    buf[x] = 0;
```

Figure 3: Examples of SPR transformations

3.1 Acquiring the data

For both the empirical study that assess the effectiveness of previous automatic repair tools and the purposes of devising new targeted search spaces, we need the following information for each of Android vulnerabilities:

- Patch: code before and after fixing a vulnerability
- Type: privilege escalation, denial of service, etc.
- The entire project source code before a fix (needed to assess the GenProg search space)
- Miscellaneous: date, severity, etc.

The required data was obtained from the Android Security Bulletins (ASB) website[2], which is the official website that lists security vulnerabilities in Android, their description and links to corresponding software repositories that contain fixes.

We created a program-crawler to parse the ASB website and organize it in a database for further use. The source code of the crawler is released[3]. Due to imperfections of the implementation and format inconsistencies of the website, we manually refined the data by fixing some of the fields in the database (minor changes: the amount of work is approximately 0.5 person-hours).

3.2 Data description

We mined all the data the ASB website contained at the time of running the program-crawler: 636 CVEs (the Common Vulnerability and Exposures identifications) that correspond to 251 Android vulnerabilities (the ASB website is structured in such a way that one "vulnerability" can correspond to multiple CVEs). Developer patches correspond to CVEs rather than "vulnerabilities" (using the ASB terminology); due to that, during the subsequent manual examination, we consider each CVE patch as an atomic unit (rather than a collection of all the CVE patches for a particular vulnerability). Later in the paper, we use the terms "CVE" and "vulnerability" interchangeably.

The most recent vulnerabilities do not have a corresponding link to the developer patch (as we suspect, due to security reasons). After removing such entries, 434 vulnerabilities remained, with publishing date ranging from August 2015 to August 2016. Afterwards, we prune out vulnerabilities, patches for which do not change the source code (e.g., configuration changes); after this step, 430 vulnerabilities remained.

The vulnerabilities obtained span several programming languages: C, C++ and Java. As Figure 4 depicts, most of them either of a critical or a high severity. Figure 5 presents the vulnerability types; privilege escalations are the most

[2]http://source.android.com/security/bulletin/
[3]https://github.com/last5bits/cs858/tree/master/VulnCrawler

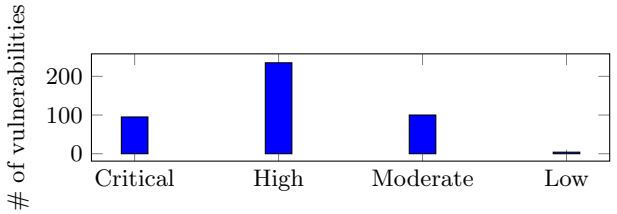


Figure 4: Distribution of vulnerabilities by severity

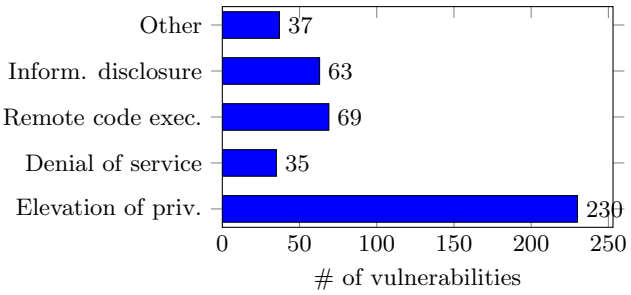


Figure 5: Distribution of vulnerabilities by type

common vulnerability type among all the vulnerabilities obtained.

4. ASSESSING SEARCH SPACES OF GENPROG AND SPR

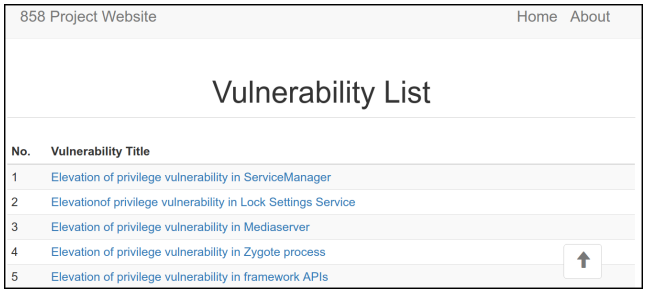
In this section, we describe the empirical study, in which we manually examined 430 Android vulnerabilities to assess the effectiveness of GenProg and SPR search spaces, and present the results.

4.1 Methodology

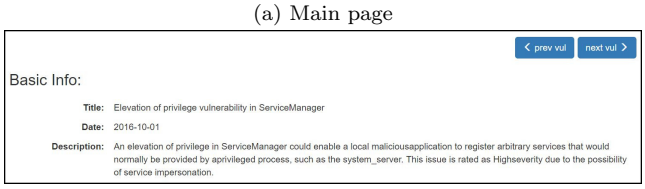
During this empirical study, we aimed to answer the following research question: how effective are the search spaces of GenProg and SPR in fixing Android vulnerabilities? To answer it, we manually examined developer fixes for 430 previously mined vulnerabilities, and for each one we made a decision whether the developer fix is in the search space of a particular tool.

Section 2 provides information on how GenProg and SPR search spaces are structured; to conduct such an empirical study, a precise understanding of the GenProg and SPR search spaces is needed (e.g., what are the ingredients for GenProg? What types of variables can be used by SPR?). Before commencing the study, the authors of this work thoroughly discussed what each tool can and cannot do; during the manual examination, the authors discussed between each other particularly unclear cases.

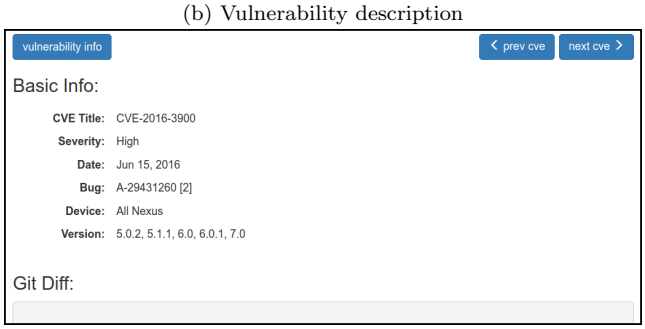
To facilitate the process of manual examination, we created a service in the form of a website that allows to go through all the vulnerabilities and make a decision for each. Figure 6a shows the main page of the website that lists all the vulnerabilities. By clicking on a link, a vulnerability (using the ASB terminology, cf. Section 3.1) description opens (Figure 6b); it also allows to go to the next or the previous vulnerability. A vulnerability page contains a list of associated CVEs; by clicking on a CVE link, a CVE-description page opens (Figure 6c). This page also contains the developer patch that fixes this CVE; the patch is presented by pulling the `git diff` information from a corresponding git-repository. To assess the GenProg search space, an ingredient search can be conducted by putting a particular



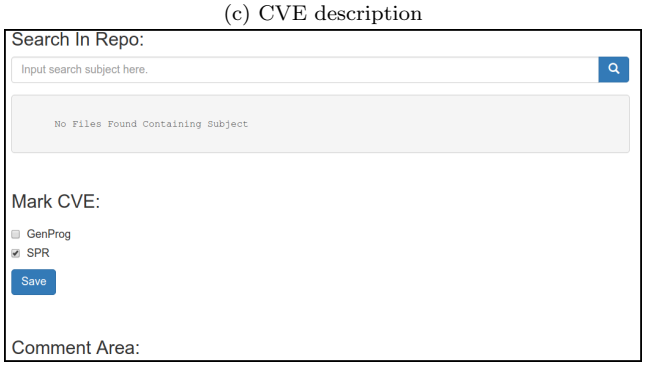
(a) Main page



(b) Vulnerability description



(c) CVE description



(d) Marking section

Figure 6: A service to conduct the empirical study

statement into the search bar (cf. Figure 6d). After the decision is made, corresponding check-marks are manually set and the database is updated.

We release the source code and deployment instructions for the presented service[4]. These materials should be helpful not only for the purposes of reproducing our study, but also for conducting similar studies or extending the current one. Additionally, we provide read-only access to the website that was used by the authors during the assessment[5].

4.2 Results of assessing the search spaces

Figure 7 presents the numbers of correct patches inside GenProg and SPR search space in the form of a Venn diagram: out of total 430 vulnerabilities, 46 can be potentially

[4]https://github.com/last5bits/cs858/tree/master/MarkWebsite
[5]http://lt-pc1.uwaterloo.ca/mark-cs858

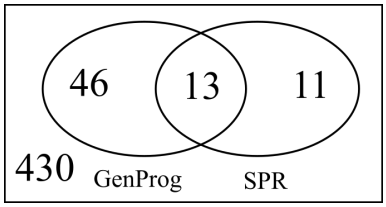
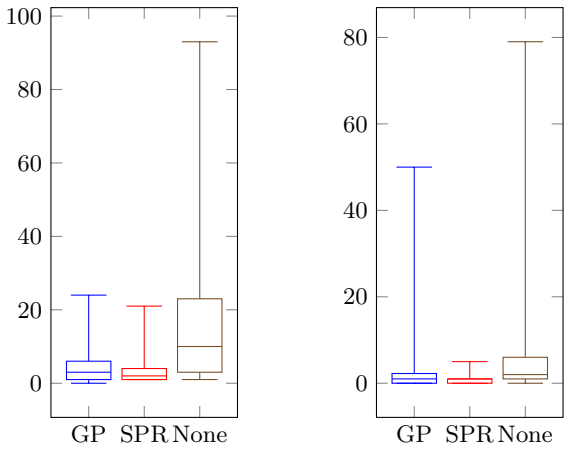


Figure 7: Comparison between GenProg and SPR in terms of numbers of correct patches inside their search spaces



(a) # of added lines (b) # of deleted lines

Figure 8: Patch complexities of correct patches of GenProg (GP), SPR and patches that cannot be fixed automatically

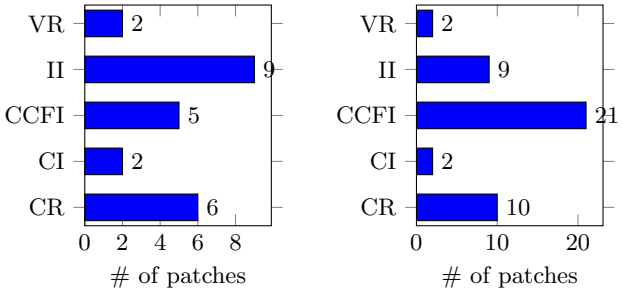
fixed only by GenProg, 11 only by SPR and 13 can be fixed by the both tools. Majority of vulnerabilities (360) cannot be fixed by current automatic repair tools. Note, that a higher number of correct patches in GenProg's search space does not necessarily imply a higher number of correctly produced repairs. Previous work shows that SPR's search space is more targeted and has a higher probability of producing a correct patch [10].

The reason behind such a low number of correct patches in the GenProg and SPR search spaces is a relative complexity of developer patches. Often, they involve more complex code changes than e.g. a simple condition refinement; thus, SPR's search space often lacks a correct patch. In the case of GenProg, programs under examination often lacked "fix ingredients". In Figure 8, we show a comparison between complexities of patches that can be fixed by GenProg, SPR and the ones that can be fixed by neither (we removed several outliers with a number of added / deleted lines greater than 100). Patches that cannot be fixed automatically are more complex both in terms of numbers of added and deleted lines.

Figure 9a shows the distribution of SPR transformations among the correct patches inside SPR search space. Transformations of the type "insert initialization" prevail over all the other types. Note: we assume that deletions of functionality can be fixed by a condition introduction of the form `if (0) { ... }`.

In addition, we provide open access to the SQL database with the results of this empirical study[6].

[6]host: lt-pc1.uwaterloo.ca; DB name: cs858; user/pw: guest/guest



(a) Default SPR transformations (b) Default + new SPR transformations

Figure 9: Distribution of transformations employed in correct SPR patches. CR is Condition Refinement, CI is Condition Introduction, CCFI is Conditional Control Flow Introduction, II is Insert Initialization, VR is Value Replacement

5. CREATING TARGETED SEARCH SPACES

In this section, we propose new targeted search spaces that extend the transformations employed by SPR and suggest a different order in which those transformations should be applied.

5.1 Proposing new transformations

When creating new search spaces for automatic software repair tools, one should be careful not to introduce transformations that make the size of a search space too large; this can lead not only to the increase of a number of correct patches in a search space, but also to the even greater increase of a number of *overfitted* patches. Therefore, we avoided proposing complex transformations; for example, multilevel condition introductions of the form `if (A && B && C)` lead to a large number of different combinations of conditions and can potentially make the search ineffective. With that in mind, we propose more conservative transformations as described below.

Extending conditional operators. In original SPR conditional transformations (i.e., condition introduction/refinement and conditional control flow introduction), only conditions of the form "`(v == const)`" or "`(v != const)`" are allowed. We propose to extend the original SPR transformations by using `<`, `>`, `<=`, `>=`. With these new transformations, SPR can potentially fix four more vulnerabilities with condition refinement and eight more vulnerabilities with conditional control flow introduction.

Boolean functions. In addition, we propose to use calls to Boolean functions (i.e., functions that return a Boolean value) inside conditional control flow introductions. With these new transformations, SPR can potentially fix eight more vulnerabilities. Figure 10 shows an example of such a transformation. This vulnerability (CVE-2014-9790) is in the reading and writing routines of the Qualcomm driver MMC driver; it allows the attacker to escalate privileges via a "specially crafted application" [1]. A call to `access_ok` verifies user privileges and aborts the function if access is denied.

With the aforementioned extensions, SPR can potentially fix 20 more vulnerabilities compared to the vanilla version of SPR; Figure 9b shows a breakdown by each transformation category.

We chose to extend SPR's search space rather than GenProg's due to significantly better results of SPR in terms of the number of bugs fixed [10] and due to many of GenProg

```
size_t mmc_wr_pack_stats_write(...)
        if (!card)
                return cnt;

+       if (!access_ok(VERIFY_READ, ubuf, cnt))
+               return cnt;
```

Figure 10: Example of using a Boolean function in a condition

patches being degenerate functionality deletions in practice [16].

5.2 Prioritizing the transformations

As Long et al. [11] showed in their empirical study, introducing more complex conditions into conditional transformations can lead to a blow-up of overfitted patches and ultimately can hinder the process of bug repair. We mitigate this by using a *targeted* prioritization of transformations; i.e., a different order in which transformations are applied to attempt to fix a vulnerability.

Based on the numbers in Figure 9b, we suggest the following order in which the transformations should be applied to attempt to fix a vulnerability:

1. Conditional control flow introduction
2. Condition refinement
3. Insert initialization
4. Condition introduction
5. Value replacement

6. LIMITATIONS & THREATS TO VALIDITY

The concept of a search space is relevant only to search-based automatic repair tools (e.g., SPR [10], Prophet [9] or GenProg [7]). The research community works on other approaches as well; for example, *semantic-based* automatic repair (DirectFix [12], Angelix [13]), in which a repair produced by employing symbolic execution and constraint solving.

Previous research shows that a portion of developer commits contains changes that are irrelevant to the target bug (e.g., quick refactoring). Ngueyen et al [15] reported that 11 – 39% of bug-fixing commits contain irrelevant changes and we believe that for vulnerability-fixing commits this number should be similar. To mitigate this issue, a more rigorous empirical study should be conducted, in which every vulnerability is precisely examined and relevant changes are manually distinguished from irrelevant ones; we leave this as future work.

Since the empirical study is performed by the authors manually, there is a risk of human bias. Additionally, in the process of manual examination, it does not seem feasible to decide whether the "Copy and Replace" SPR transformation can be used to repair a specific vulnerability; this is due to a great number of possible combinations of statements that can be copied and replaced. This can be solved by creating a tool that generates all the possible combinations and checks a patch automatically. This limitation of our study is mitigated by the fact that the "Copy and Replace" transformation has the lowest priority in vanilla SPR and is not applied often in practice (based on our experience of examining SPR patches for GenProg benchmarks).

The results of our study might not generalize to all the vulnerabilities or all the mobile software systems due to the empirical nature of the study. We mitigate this by examining a large number of vulnerabilities (430).

7. RELATED WORK

Automatic repair of specifically mobile software is its infancy; the most related work is by Azim et al. [2], their approach detects programs' crashes, immediately patches the bytecode of the program (to be certain that it won't crash again) and rolls the program state to the nearest activity. Automatic repair of desktop applications received more attention; the tool Prophet [9] is a state-of-the-art technique that prioritizes potential fixes inside the search space by learning what correct *developer* fixes look like. This direction is promising, however it highly depends on the data to learn from, which is not always present. Our approach of targeted search spaces is orthogonal to the one of Prophet.

BovInspector [4] is a tool for automatic repair of buffer overflow vulnerabilities; its workflow consists of several stages: detecting potential buffer overflows with static analysis, constructing a Control Flow Graph (CFG) and performing a reachability analysis; the reachability information is subsequently used to guide symbolic execution and mitigate its inherent problem of path explosion. After a buffer overflow is confirmed by symbolic execution, BovInspector employs three strategies to fix an overflow: introduce an if-condition, change an API function (i.e., `strcpy` to `strncpy`) or expand the buffer. One of the limitations of BovInspector is its focus on only buffer overflows, while a search-based repair can target a more wide range of vulnerabilities.

Mining bug patterns. There is prior work on manually or semi-automatically inspecting software to create bug patterns [5, 6, 19, 8]. For example, Hanam et al. [5] created an approach to automatically cluster bug-fixing commits of JavaScript code; then, these clusters are manually inspected to construct a set of patterns that represent the most common types of bugs. Some of the found patterns can be used to construct search spaces for automatic repair tools; however, some are not specific enough for this purpose. In another example, to create the PAR automatic repair tool [6], the authors manually inspected a large corpus of human-written patches to devise a set of common bug-fixing patterns, and then used these patterns to implement an evolutionary program-repair algorithm. Although close to our work, the PAR's patterns target a wide range of bugs, rather than only software vulnerabilities. We believe that more narrow search spaces should improve the effectiveness of automatic software repair. Li et al. [8] do consider security vulnerabilities in a separate category when inspecting software bugs; the bugs are classified in terms of their root causes, impacts and software components but no concrete patterns, which could be used by automatic repair tools, are proposed. Ye et al. [19] study false positives and false negatives of static analysis tools in regard to detecting buffer overflows. Additionally, this paper studies fixes that developers employ to fix buffer overflows (e.g., adding boundary checks). While this work should be helpful in devising new search spaces to automatically fix buffer overflows, it does not study other types of vulnerabilities and does not focus on mobile systems.

8. CONCLUSIONS

Automatic repair tools are unable to fix bugs correct patches for which are outside of their search space. However, as the prior work shows [11], blind extensions of search space without looking at the problem domain leads to fewer correct patches produced. In the current work, we conducted an empirical study of 430 Android vulnerabilities to (1) assess the search space of GenProg and SPR in terms of numbers of correct patches, and (2) obtain an insight on how to improve existing search spaces to better target *security vulnerabilities* in *mobile software*. Empirical study showed that majority of Android vulnerabilities cannot be fixed by GenProg or SPR. With the insight obtained during the manual examination of developer patches, we outlined a search space that contains fixes for 44 vulnerabilities (compared to the previous 20) and has a more promising prioritization of transformations. Using this search space should improve effectiveness of automatic repair tools that aim at fixing vulnerabilities in Android.

9. REFERENCES

[1] Common vulnerabilities and exposures, cve-2014-9790, 2016. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-9790.

[2] M. T. Azim, I. Neamtiu, and L. M. Marvel. Towards self-healing smartphone software via automated patching. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 623–628. ACM, 2014.

[3] S. Egelman, S. Jain, R. S. Portnoff, K. Liao, S. Consolvo, and D. Wagner. Are you ready to lock? In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 750–761. ACM, 2014.

[4] F. Gao, L. Wang, and X. Li. Bovinspector: automatic inspection and repair of buffer overflow vulnerabilities. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 786–791. ACM, 2016.

[5] Q. Hanam, F. S. d. M. Brito, and A. Mesbah. Discovering bug patterns in javascript. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 144–156. ACM, 2016.

[6] 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*, pages 802–811. IEEE Press, 2013.

[7] 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 *Software Engineering (ICSE), 2012 34th International Conference on*, pages 3–13. IEEE, 2012.

[8] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai. Have things changed now?: an empirical study of bug characteristics in modern open source software. In *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, pages 25–33. ACM, 2006.

[9] F. Long and M. Rinard. Prophet: Automatic patch generation via learning from successful patches. 2015.

[10] F. Long and M. Rinard. Staged program repair in spr. 2015.

[11] F. Long and M. Rinard. An analysis of the search spaces for generate and validate patch generation systems. In *Proceedings of the 38th International Conference on Software Engineering*, pages 702–713. ACM, 2016.

[12] S. Mechtaev, J. Yi, and A. Roychoudhury. Directfix: Looking for simple program repairs. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 448–458. IEEE, 2015.

[13] S. Mechtaev, J. Yi, and A. Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th International Conference on Software Engineering*, pages 691–701. ACM, 2016.

[14] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. Inside the slammer worm. *IEEE security and privacy*, 1(4):33–39, 2003.

[15] H. A. Nguyen, A. T. Nguyen, and T. N. Nguyen. Filtering noise in mixed-purpose fixing commits to improve defect prediction and localization. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, pages 138–147. IEEE, 2013.

[16] Z. Qi, F. Long, S. Achour, and M. 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*, pages 24–36. ACM, 2015.

[17] P. Streak. The morris worm: A fifteen-year perspective. 2003.

[18] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa. A survey on software fault localization. 2009.

[19] T. Ye, L. Zhang, L. Wang, and X. Li. An empirical study on detecting and fixing buffer overflow bugs. In *Software Testing, Verification and Validation (ICST), 2016 IEEE International Conference on*, pages 91–101. IEEE, 2016.