Apply+ : A tool to intelligently apply security patches

Vikram N. Subramanian*, Shayon Banerjee* , Yinuo Wang*, Yuvika Khardenavis*, Meiyappan Nagappan
SWAG Lab
David R. Cheriton School of Computer Science
University of Waterloo
{vnsubram, s23banerjee, yn6wang, ykharden, mei.nagappan}@uwaterloo.ca
* *Equal contribution in developing tool*

Glenn Wurster, Scott Cosentino
BlackBerry Limited
{gwurster, scosentino}@blackberry.com

***Abstract*—Open source projects are sometimes forked by other organizations who continue development on the fork. One of the tasks that is common for such organizations is to take the patches that fixes vulnerabilities in the original open source project and apply them to the fork. In the Git version control system, they would use a tool called git-apply. However, because of independent development on the fork, git-apply can fail to apply patches. In this paper, we build a tool on top of git-apply that can resolve issues and apply more of the patches than just git-apply. We first carry out a study to see when git-apply fails and then build a tool called Apply+ that will resolve patch application in some of those cases. We carry out a case study on two open source projects to see how many of the failed patches we can apply. We validate the correctness of the tool in an industrial setting as well. Overall, we find that our tool can help developers apply patches when git-apply fails.**

***Index Terms*—Patches, Git-apply, Security Vulnerabilities**

I. Introduction and Motivation

Open source is commonly used as part of commercial software development in order to reduce development costs. While some open source components are used as-is, other components are forked and modified during development. These modifications might add new features or change functionality based on requirements, resulting in large changes to the source code. The resulting fork, often maintained in-house, diverges in ways that make applying (or detecting the application of) patches laborious. Open source projects release patches (or patch files) as vulnerabilities are fixed. A vulnerability is fixed if the patch that fixes that vulnerability is *applied* to the source code. For any fork, including those maintained in-house, it is important to identify if these vulnerabilities still exist. If they do, it is necessary to apply the patch in order to remove the vulnerability.

The `apply` command in GIT (Git-apply) is used to apply patches to software repositories. To apply a patch, it searches for an exact match of the state of the source file as found in the patch containing the changes. Modified forks often contain changes to files that a patch is also trying to modify. When Git-apply cannot find an exact match, it will return an error. A developer must manually search for the location to apply the patch, understand how the patch is different, and apply the necessary changes manually. By empirically studying a large number of patches, we discovered that there is often a pattern to the way the source file changes. Many patches can be automatically applied by a *smarter* tool. Our goal is to develop an algorithm and associated tool that can automatically apply patches to modified forks of projects.

II. Background

A. Terminology Used in this paper

1) **Hunk**: A hunk is a textual representation of a modification to be made to a adjacent lines from a single file. A hunk has information on what adjacent lines should contain before and after the hunk is applied. A hunk is made up of context lines, lines to be removed, and lines to be added. A hunk begins with a line containing @@, the starting line number of the adjacent lines, and optionally a function name. A hunk may optionally be prefixed by the name of the file to which the hunk should be applied. If a hunk is not prefixed by the name of the file, the same file name as the previous hunk is assumed.
2) **Patch**: A patch is a file that contains one or more hunks. The first hunk in a patch will always be prefixed with the name of the file to be modified. Git-apply, when given a patch, will apply each hunk listed in the patch to the given repository. In this paper, we assume patches which apply to C, C++, C#, or Java source code.
3) **Context lines**: Lines of code that start with a space are context lines. In Figure 1, they are black bold lines. Context lines are unchanged lines used to find the position of lines added or removed by the hunk. Context lines might be empty (except for the beginning space).
4) **Removed lines**: Lines starting with a – are removed when applying a hunk. In Figure 1, they are also

```
diff --git a/net/ipv4/netfilter/arp_tables.c b/net/ipv4/netfilter/arp_tables.c
index f95b6f9..3d45ce5 100644
--- a/net/ipv4/netfilter/arp_tables.c
+++ b/net/ipv4/netfilter/arp_tables.c
@@ -470,14 +470,12 @@ static int mark_source_chains(const struct xt_table_info *newinfo,
        return 1;
 }

-static inline int check_entry(const struct arpt_entry *e, const char *name)
+static inline int check_entry(const struct arpt_entry *e)
 {
        const struct xt_entry_target *t;

-       if (!arp_checkentry(&e->arp)) {
-               duprintf("arp_tables: arp check failed %p %s.\n", e, name);
+       if (!arp_checkentry(&e->arp))
                return -EINVAL;
-       }

        if (e->target_offset + sizeof(struct xt_entry_target) > e->next_offset)
        return -EINVAL;
```

Fig. 1. Example of a hunk

highlighted in red. Running Git-apply would remove this line from the file.

5) **Added lines**: Lines starting with a + are added when applying a hunk. In Figure 1, they are also highlighted in green. Running Git-apply would add this line to the file.

B. How Git-apply works

Git-apply works through each hunk in a patch, searching for an exact match (character-by-character) of the context lines and to-be-removed lines in the hunk. It searches only the file associated with the hunk, starting from the line mentioned in the hunk expanding outwards. When it finds an exact match, it adds and removes the lines listed in the hunk. If it does not find an exact match, Git-apply will return an error.

A patch contains one or more hunks, changing one or more files. Git-apply individually processes each hunk in the same order as listed in the patch. All hunks are applied to a file if and only if all hunks changing that file in the patch can be applied successfully by Git-apply. If one hunk fails for a file, no changes are made to that file. This means Git-apply can partially apply a patch by applying only a few hunks from it (to the files where all the hunks changing it produced no error) and raising errors for the hunks that didn't. If all the hunks in a patch produce no error, the patch runs successfully and raises no errors

Git-apply will fail to apply a hunk if an exact match is not found. This includes changes in white space, new lines, extra comments, and variable name changes. These changes, however, do not cause functional differences in the program. The conservative behaviour of Git-apply can be seen as a safety feature - applying some hunks without an exact match might have bad consequences. When we apply these hunks to modified versions of projects, however, Git-apply fails more often. A developer would manually apply failed hunks, even those for which no functional change existed. Having a tool that can identify non-functional differences in hunks and apply them without requiring an exact match will save developer time and effort.

C. Related work

The Unix program, `patch` (1) is a tool similar to Git-apply. However, unlike Git-apply, it supports fuzzy matching. Fuzzy matching in the patch utility works by ignoring one or more context lines when searching for the location to apply a hunk. Our approach is more fine-grained. It does not ignore the entire context line when performing a fuzzy match, making it less likely to apply a patch incorrectly. While our approach provides better assurance that functionality is not affected by applying the patch, our approach does require an understanding of source language to which the patch is being applied.

Binary Patch Management Systems-

While source code patches apply to source code, binary patches also exist. Binary patches are applied to compiled applications to fix vulnerabilities. Unlike source code patches, binary patches are specific to individual binaries. Binary patches are shipped to customers, allowing them to patch vulnerabilities in their applications. Our work focuses only on patches which are applied to source code, not binaries.

Zhao et al. (2) developed a binary patch management system to be a precaution against security vulnerabilities. It maintains a database and has tools to apply patches from this database. Jung-Taek et al. (3) proposed a binary patch management system for multi-platform environments with a patch profiling

mechanism and patch dependency solving mechanism. Jung-Taek et al. (4) proposed a patch management system, after researching existing patch management systems and provided solutions for managing and distributing critical patches that resolved known security vulnerabilities and other stability issues with various platforms. Chang et al. (5) presented a five-layer patch-management-system application architecture supporting heterogeneous environments hoping to make enterprise patch management more efficient and reducing the risks of improper patch management. Higby et al. (6) proposed a wireless security patch management/antivirus update system to remedy the problems presented by wireless clients that are not updated with current security patches, antivirus software, and weak or no firewall settings.

Apply+ focuses on applying source code patches instead of binary patches. Additionally, unlike a lot of the work described above, it is not a patch management system. It's scope is to only deal with errors in individual patches. A developer can't keep track of patches they have applied to projects or investigate vulnerabilities that have been fixed/not fixed through Apply+

Economics of Patch Management-

A game-theory model was developed by Cavusoglu et al. (7) to study the strategic interaction between a vendor and a firm in balancing the costs and benefits of patch management. Another game-theory model was developed by Cavusoglu et al. (8) to derive the optimal frequency of patch updates to balance the operational costs and damage costs associated with security vulnerabilities. Okhravi et al. (9) discussed an analytical model for the trade-off between pre-deployment testing and the total number of open vulnerabilities in a system. They also developed a stochastic model a the patch management system and solved it using a simulation tool.

These studies discuss the costs, benefits and trade-offs of using a patch management system. Once again, Apply+ is not a patch management system but a tool that can apply an individual patch. Apply+ could however improve the effectiveness of patch management systems by reducing the number of times patches fail to apply.

Patch Management Practices-

Brykczynski et al. (10) provided guidance to organizations on the aspects of well-engineered patch management. They examined eight key patch management practices to reduce or eliminate vulnerability management. Zhang (11) provided managerial insights for effective patch management in organizations, for efficient learning from security vulnerabilities, and for appropriate product strategies in a security software market. Gerace et al. (12) discussed the results of a survey of IT professionals which sought to determine critical elements in the patch management process.

These studies discuss best practices while managing patches in order to avoid, amongst others, the situation that creates a demand for Apply+. Apply+ tries to fix issues that, in-part, these best practices can avoid.

Furthermore, none of the related work address the problem of applying source-code patches to modified versions of projects when git-apply fails- the goal of this paper.

III. Preliminary Study

The first step in achieving higher patch application rate for forked source repositories is to understand how and why Git-apply fails to apply patches. By examining the reasons for failure, we can better handle each type of failure.

We collected 720 patches that fix vulnerabilities from the Linux kernel open source project for our initial analysis (13). The CVE Details website (14) was manually browsed for vulnerabilities affecting the Linux kernel from 2012 to 2016 inclusive. For each vulnerability, the references section was checked for commits to the Linux kernel, identified by a GIT commit hash. The referenced patch was retrieved from the Linux kernel mainline GIT repository(15). In cases where more than one patch was identified, all patches were retrieved.

We attempted to apply these patches to the msm-3.10 Linux kernel from Codeaurora (16) to study the individual reasons for failure. We describe our data collection and the subsequent analysis below.

A. Data Classification

A patch is a collection of hunks changing one or more files. Git-apply applies a hunk to a file only when all the hunks changing that file produce no errors. A patch failing does not mean all hunks in that patch failed. We choose to classify and study only the hunks that failed. Apply+ also treats each hunk individually, processing them one at a time.

In order to produce the categorization, we investigated the reason hunks failed when run with Git-apply on kernel/msm-3.10. Based on our analysis, we created a classification for why hunks can fail to apply.

1) **Already applied** - The hunk has already been applied. Reapplying it causes Git-apply to fail. All of the removed lines are missing and all of the added lines are already present. These hunks might have already been applied by developers.
2) **Code Change**
   a) **Context Changed** - The context lines were missing or different from the ones in file. In the example below, the hunk would fail to apply because the `x = 15;` line has changed.

Hunk

```
 int x = 15;
-x = 4;
+x = 5;
+x += 1;
 std::cout << x << std::endl;
```

Source File

```
int x = 3;
x = 4;
std::cout << x << std::endl;
```

b) **Removed lines missing or changed** - Lines removed by the hunk were missing or different from those in the hunk. Lines added by the hunk were not identified. In the example below, the hunk would fail to apply because the `x = 4;` line is missing.

Hunk

```
 int x = 3;
-x = 4;
+x = 5;
+x += 1;
 std::cout << x << std::endl;
```

Source File

```
int x = 3;
std::cout << x << std::endl;
```

c) **Partially applied** - Some of the added line in the hunk are present, but not all. In the example below, the hunk would fail to apply because the line `x += 1;` already exists.

Hunk

```
 int x = 3;
-x = 4;
+x = 5;
+x += 1;
 std::cout << x << std::endl;
```

Source File

```
int x = 3;
x = 4;
x += 1;
std::cout << x << std::endl;
```

If all the lines added by the hunk are already present and all lines removed by the hunk are absent, we categorize that hunk as 'Already applied'. Our tool simply reports this and takes no further action. For 'Partially applied' hunks, on the other hand, Apply+ performs the analysis documented in Section 5

Note: A hunk can be categorized under more than one of 'Partially applied', 'Removed lines missing or changed' and 'Context Changed', depending on the differences between the hunk and the source.

3) **File Changed**- The file where the hunk should be applied was deleted, moved, renamed, or not created yet. The patch failed as the file where the change had to be applied was not found.

IV. Approach behind our tool

Apply+ starts by taking in a patch and the path to the root of the repository the patch is being applied to. It first checks if Git-apply can apply the entire patch. If Git-apply can, the tool informs the user of this and applies the patch. If not, it processes each hunk in the patch, one at a time through the algorithm described in Figure 2.

Due to Apply+ using the srcSclice tool for program slicing, Apply+ supports only C, C++, C#, and Java. Apply+ is written in Python 3.

Some definitions-

1) **Levenshtein Ratio**: The Levenshtein Ratio(17) is used to determine the level of similarity between two strings a and b. The ratio takes on values in the interval $[0, 1]$, with a Levenshtein ratio of 1 between strings a and b meaning that $a = b$. It is calculated as $1 - \frac{e(a,b)}{|a|+|b|}$, where $e(a, b)$ is the number of edit operations required to convert a to b, with the only valid edit operations being adding a character and removing a character. $|a|$ and $|b|$ are the length of the strings a and b respectively.
2) **R-Value**: R-value refers to an absolute data value that is not stored at some address in memory. An R-value is an expression that cannot have a value assigned to it. An R-value can appear on the right but not on left hand side of an assignment operator(=).
3) **L-Value**: L-value represents an object that occupies a location in memory (i.e. has an address) to which values can be assigned.

A. Parsing Patch files- Step 1-2, 8 of Algorithm.

Patches and individual hunks have a very specific syntax. We built a parser that takes in the location of a patch and returns a list of hunk objects. Each hunk object corresponds to one hunk in the patch. It contains all the lines mentioned in the hunk, along with whether it is a context, added, or removed line. Each object also contains the line numbers indicated in the hunk and the path to the file modified by the hunk. We also wrote scripts to emulate the behaviour of Git-apply with individual hunks.

B. Fuzzy Searching for a Patch- Step 4 of Algorithm.

The first step in determining whether a hunk has been applied is to find the lines of code that the hunk is attempting to change. Git-apply's methodology of searching for an exact match of the context lines and to-be-removed lines (as described in Section II-B) cannot be applied in the case where the context lines differ even slightly. To counter this problem, the tool runs fuzzy string searching using Google's Diff-Match-Patch library(18).

The initial search area is the location of lines in the original file, which we get from the hunk. The search iteratively expands outwards from there until a fuzzy match is found or the whole file is covered. If a fuzzy matched block of code is found, each line in the hunk is assigned a corresponding line in the block of code (the line in the code that is most similar to the line in the hunk). Code similarity is then measured

via calculating the Levenshtein ratio (as defined in 1). If any context line has a match ratio below the Levenshtein ratio threshold of 0.8, that block of code is dropped and the search continues.

C. Determining whether to run a patch- Step 5-7 of Algorithm.

The context lines surrounding the lines added and removed by the hunk may be different in the hunk and code. This difference would prevent Git-apply from finding an exact match and applying the hunk. Apply+ makes up for this shortcoming by first quantifying the similarities using the Levenshtein ratio discussed above by finding a similar line in the file for each line in the hunk. Then, Apply+ compares the two lines, and determines if there is a semantic difference to make a decision on whether to apply the patch or not. This section explores that decision-making process.

First, the algorithm runs srcML and srcSlice [13] on the files the hunk is attempting to apply to. The srcML tool converts C, C++, C#, or Java source code into an XML representation. The output of srcML is fed to srcSlice, a forward static slicing tool. The srcSlice tool returns a list of line numbers, dependent variables, aliases, and function calls for each variable in the target file.

When exploring the semantics of each context line, the Apply+ algorithm will conduct a series of checks on each line. It stops processing the whole hunk even if any check fails. The tool considers modifications to context lines of the following types: function declarations, function calls, function assignments, L-Value and R-Value variable changes.

1) **Function declarations-** If a function declaration in the context lines of the hunk is different from the code, the algorithm will not apply the hunk. An example would be:
```
Patch File: void example_function();
Source File: void example_fn();
```
Apply+ does not apply the hunk because there may be other dependencies affected by making this change. For function declarations, there may be dependencies outside the scope of the file such as other functions and variables. Changing the name of the function will result in code that either does not compile or does not run properly. For these reasons, the algorithm will decide to not run the hunk, while providing slicing information for line numbers, dependent variables, aliases, and functions calls and the option to override this decision.
2) **Function calls-** If the change reflects a function call, such as:
```
Patch File: example_function(...);
Source Code File: example_fn(...);
```
For this change, Apply+ will try to continue to apply the hunk. The algorithm will go on to process the next context line, but gives a warning to the user before proceeding. The reason behind this decision is that it assumes that the function in the hunk has previously been defined or will be defined in the patch file. With that being said, it is possible that this function call could introduce potential unwanted side effects. For this reason, the algorithm will provide a warning message to the user with the context lines in question. This message will give the user the ability to make a decision on whether this change should be applied or not.
3) **Function assignments-** For function calls that are assigned to a variable, such as:
```
Patch File: result = example_function();
Source File: result = example_fn();
```
This change would result in Apply+ not applying the hunk. The reason behind this decision is that the variable, the function is assigned to, might have dependencies within or outside the scope of the sliced file. Once again, the algorithm provides slicing information, with the option to override the decision made.
4) **L-Value Change-** The algorithm decides to not run the hunk when a L-Value variable (as defined in 3) has changed between the hunk and code. Two examples of L-Value variable differences between a hunk file and source code are provided below:
```
Patch File: X = 5;
Source File: Y = 5;
```
or
```
Patch File: Z = X;
Source File: Z = Y;
```
These variables may have dependencies, such as functions or variables, outside the scope of the file that is sliced. Allowing the change to update the code could result in undesired side effects. For this reason, the algorithm will decide to not run the hunk, while providing slicing information to the user.
5) **R-Value Change-** If the context line difference represents an R-Value change (as defined in 2), the algorithm will aim to continue to apply the hunk by moving to the next context line, but first provide a warning message to the user.
An example of R-Value variable change:
```
Patch File: X = 3;
Source File: X = 4;
```
In other words, if the value stored at the address of a variable is changed by an entity that does not occupy some identifiable location in memory, the algorithm will not raise an issue with that specific line. Although, this R-Value assignment could introduce a potential side effect when the variable being assigned to is used elsewhere. We take this dependency into account and provide a warning message to the user along with slicer information. Hence, the user will be able to make a decision on whether this R-Value change should be applied or not, before continuing to other context lines.
6) **Other Changes-** Finally, if a context line is neither a function declaration, function assignment, variable declaration, variable assignment, or an unstructured control flow, such as go to statements or exceptions, no error is raised and the program continues. Exceptions and go-to statements can not be sliced by srcSlice; hence, these statements are not handled by our tool either.

1) Parse Patch file for individual hunks and run step 2 to 7 for each of them, one by one.
2) Check if the Git-apply emulator for hunks (See section IV-A) can apply the hunk.
   a) If yes, inform the developer and apply the individual hunk (if desired). Stop processing the hunk.
   b) Else, continue to step 3.
3) Search for an exact match for all context lines and all lines added by the hunk
   a) If an exact match is found, check if all the lines removed by the hunk are not present. If so, inform the developer that this individual hunk has already been applied. Stop processing the hunk.
   b) Else, continue to step 4.
4) Search for the patch using the fuzzy search algorithm described in section 4b.
   a) If no fuzzy match is found, inform the developer that we do not know where the hunk should be applied. Stop processing the hunk.
   b) If a fuzzy match is found that links context lines in the hunk to the code in file, continue to step 5.

   At this point, we know that the individual hunk has not been applied already and that there are at least a few matching context lines between the hunk and file. The algorithm now aims to compare the semantics of each of the context lines in the hunk file to the corresponding context lines in the source file.
5) Create a XML representation of the source file that is being changed by the hunk using srcML [13].
6) Feed the XML representation to srcSlice [13]. Save the result.
7) Iterate through each context line in the hunk
   a) Compare, character by character, each context line in the hunk to the corresponding line in file (as identified in Step 4). If they are different in any way proceed to step b. Else continue to the next context line.
   b) Compare the functionally as described in section IV-C. If any of the context lines differ semantically, inform the developer that we might have found where to apply but can't automatically apply, provide information such as the location where the hunk should be (file and line number) and what percentage of context lines, lined to removed and lines to be added were found. Stop processing the hunk.
8) None of the context lines differ functionally. Therefore, apply the hunk using the Git-apply emulator and inform the user. Stop processing the hunk.

Fig. 2. Apply+ Algorithm

V. Results

In Section III, we categorized a set of hunks in order to understand the reasons why these hunks fail to be applied. We then discussed and experimented with different algorithms and techniques to solve each reason, combined them and produced the algorithm presented in Section 5. In Section III, we categorized the reasons hunks failing. Below we explain which categories we were able to solve and how.

As explained before, a patch is a collection of hunks. Git-apply throws an error even if one hunk does not work as developers would either have the entire patch applied or not apply any of it applied. Apply+, however, treats each hunk individually. It will apply what it can, get as much information as possible for the others and let the developer take care of the hunks it can't in order to apply the entire patch.

Apply+ first runs Git-apply to check if a patch runs. If it does, it means all the hunks work and the patch is okay. We inform the developer of this and return. As mentioned before, if all the hunks changing a particular file have no errors, all the hunks changing that file are applied by Git-apply. We explicitly list the patches that were applied(if any) and process the failed hunks.

Our algorithm processes the failed hunks one by one. The hunks end up in 1 of 5 different states:

1) **Hunk already applied/partially applied** Some hunks (or even the entire patch) might have already been applied or partially applied. Reapplying them produces an error with Git-apply. Our tool detects hunks that have already been applied using fuzzy search as explained in Section 5. We are able to account for Category 1- 'Already applied' from section III-A through this.
2) **Files could not be found** and **File moved and could apply**- If the file mentioned in the hunk does not exist, Git-apply fails. Our tool searches all directories recursively starting from the root of the repository (given as input to Apply+). If a file with the same name is found in a different directory, we ask the developer if we should investigate that file. If no file with the name referenced in the patch is found, we report that to the developer. If the developer wants to search a file with the same name that we have found, we search for the context in the hunk like normal. If we find the context lines, we apply the hunk with the user's permission. This would partially account for Category 3- 'File Changed' from

Category	Codeaurora		Vivaldi	
	Count	Percentage	Count	Percentage
Hunk already applied	518	30.15%	1182	17.37%
File could not be found	82	4.77%	655	9.83%
File moved and could apply	3	0.17%	59	0.89%
Our code can apply	434	25.26%	1613	24.20%
Could not automatically apply, but we have possibly found where to apply	380	22.12%	1644	26.67%
Could not find where the hunk should be applied	301	17.52%	1511	22.67%

TABLE I
RESULTS OF RUNNING APPLY+ ON THE CODEAURORA AND VIVALDI PATCH SETS

section III-A. It does not fully account for Category 3 as some files could have been deleted or renamed.

The next three states collectively account for all hunks under Category 2- 'Code Change' from section III-A:

4) **Git-apply fails, but our code can apply** Git-apply applies a hunk only when it finds an exact match. It does not cater for syntactic changes, non-code changes (e.g., comments), empty lines, etc. Our tool accounts for these changes and applies a hunk if it's clear that there would be no unexpected consequence by applying the hunk.
5) **Can't automatically apply, but we have possibly found where the hunk should be applied** Apply+ often identifies a significant functional change in the context lines while applying the hunk (as described in section). It therefore does not apply the hunk and instead informs the developer of the location where the hunk could probably be applied (file and line number). Apply+ also outputs what percentage of context lines, removed lines and added lines were present from the hunk so that the developer can investigate.
   If there are close to 0% removed lines and close to 100% added lines from the hunk in the source file, a developer can confidently assume that the hunk has already been applied. They would still have to check that the semantic changes Apply+ found do not 'undo' the hunk and that the tool did not produce a false positive. On the other hand, if removed lines from the hunk are close to 100% and added lines from the hunk are close to 0%, the developer can predict that the hunk has not been applied.
6) **Could not find where the hunk should be applied** Similar to Git-apply, Apply+ searches for context lines in order to know where to apply hunks. If they are no context lines to be found, it is highly likely that the context lines have been removed or heavily modified. Apply+ returns with an error indicating it failed to identify the location.

Apply+ is therefore able to process all the patches and hunk categories we identified. It is able to safely apply some, recognize where others could be applied or at the least, identify that it can not apply the patch.
We now discuss the effectiveness of our tool:

A. Case Study - Linux Kernel

As described in Section III, the set of patches used to produce our classification and algorithm has 720 individual patch files from the Linux kernel. We attempted to apply the 720 patches to the Codeaurora msm-3.10 kernel using Apply+. Out of the 720 patches, only 333(46.2%) patches applied successfully with Git-apply (no-errors). 99(13.78%) patches had partial application (as no errors were produced by all hunks changing one file but errors were found elsewhere), resulting in 119 hunks being applied. 288(40%) patches failed completely. This left, in total, 1718 hunks that produced an error with Git-apply. Our tool processed these hunks and the results are presented in Table I.

Of the 1718 hunks, Apply+ was able to automatically apply 25% (518) of the hunks and identify that 30% (952) of the hunks had already been applied. For 191 of the 387 patches that produced errors with Git-apply (49.35%), we were able to rectify all issues in all hunks and the patch was fully applied using Apply+.

We were able to gather information in order to predict where the hunk should be applied for 22.12% of the hunks, even if we couldn't actually apply the hunk. A developer could use this information to quickly pinpoint where and how to apply the hunk. When we studied 40 hunks from this category, we found that Apply+ had correctly predicted the location to apply 33 of the hunks, resulting in an accuracy of 82.5%.

We could not find the file for 4.77% of the hunks. We could not find where to apply the hunk for 17.52% of the hunks. We leave it to the developer to manually investigate these hunks, but we provide data such as what percentage of context lines, to-be-removed lines and to-be-added lines were found to understand if the hunk has already been applied or not as described in Section 5.

B. Case Study - Vivaldi Browser

We collected another set of patches from the Chromium open source project to benchmark our tool against another project. We queried Chromium's bug tracker system (19) for a list of security vulnerabilities. A query string of "-status:WontFix,Duplicate Type=Bug-Security" was used. The first 5999 issues were downloaded in a CSV file, including their Bug ID. The Chromium GIT repository was then searched for any commit with the string "Bug ID:" and a referenced bug ID listed in the CSV file, resulting in 1232 patches. We attempted to use Apply+ to apply these patches to the Vivaldi Browser version 2.10.1745 (20), an open source fork of the Chromium project.

Only 295 (23.95%) of the 1232 patches applied successfully using Git-apply. 212 (17.20%) patches had partial application

(as no errors were produced by all hunks changing one file but errors were found elsewhere), resulting in 812 hunks being applied. 725 (58.85%) patches failed completely. This left, in total, 6664 hunks that produced an error with Git-apply. Our tool processed these hunks and the results are presented in Table I.

C. Industrial validation

Apply+ was also run with 123 patches against an internal BlackBerry repository. On the old branch, 117 patches applied using Git-apply. Apply+ was able to correctly apply the remaining 6 patches. Manual verification of the Apply+ patch confirmed that the solution was functionally equivalent to what was manually applied by developers in later versions of the branch. In a separate test, the tool was able to correctly apply hunks from 40 out of 48 patches and reported it couldn't apply hunks from 8 patches.

VI. Threats to validity and Areas for improvement

1) Apply+ tries to be more productive than git-apply by being not as conservative. The cost of that is the occasional false-positive but a developer would save much more time verifying that a patch has been correctly applied than manually applying it themselves.
   When we verified Apply+'s accuracy in V-C, we found no false positives.
2) The srcSlice tool has a few limitations which influence Apply+:
   a) It is designed to be run on srcML output for only C, C++, C#, and Java projects. This means Apply+ supports only C, C++, C#, and Java.
   b) Since srcSlice does not require a compiled version of the source code, it does not provide slicing information on dynamic binding through virtual functions or function pointers.
   c) It does not support full-type resolution. This means some pointer variables will be missed in the slicer output, i.e., the tool is able to accurately represent simple pointer aliasing but struggles to handle more complex scenarios. Not being able to support full-type resolution also means that only one method may be included in the slicer information (instead of all virtual methods).
   d) It isn't very successful at identifying overloaded methods or instances of polymorphism.
   e) Unstructured control flow, such as goto statements, are not represented accurately by srcSlice.
   f) Exceptions are also not supported by srcML.
   g) Apply+ runs srcSlice on only the file mentioned in the patch. This implies that dependencies that are shared between files will not be represented in the slicer output.
3) We use Google's Diff Match Patch library (18) to implement the fuzzy search algorithm to search for the context lines within the source file. It returns the first match it finds above some threshold. This means code similar to the code targeted by a hunk could be found in multiple locations in the file resulting in false positives.
4) The algorithm applies each hunk one at a time from a patch file and is not aware of dependencies that exist between hunks. For example, if our solution is applying a hunk with variables or function calls that are different from the source code, it doesn't track if these changes are defined in other hunks.
5) Code application is also an area for improvement. Currently, if there are additional lines in the matched chunk of code that do not correspond to a line in the patch, the tool is unable to determine whether this additional line will cause a semantic difference in the program or not, resulting in the tool being unable to apply the patch.
6) One area for future development could be a more robust search for missing files. Right now, we investigate only files that have the same name. We could use other smarter methods such as searching for functionally similar files.
7) The tool is rather slow in finding where to apply a hunk. The further a hunk's context lines are from its supposed location (as mentioned in the hunk), the longer it takes to find it. Several optimization techniques could be used to speed this up.

VII. Conclusion

Using Git-apply on modified forks of a open source repository produces mixed results. In our initial study, only 45% of the patches applied without issues. For the other 55%, developers would have to manually check patches and the repository and apply the patch themselves. Usually, most of the patch runs without error. Errors occur only for a few hunks, but result in the developer going through the entire patch. Apply+ tries to address this issue by doing all the *mundane* and repetitive tasks. Apply+ fixes relatively straightforward issues and involves the developer only at the point of failure. Apply+ also gathers information on hunks it fails to apply, minimizing the effort of the developer.

Apply+ and the data from the studies conducted are available at: https://github.com/meido/ApplyPlus

References

[1] I. Free Software Foundation, "Gnu patch - summary," 2018. [Online]. Available: http://savannah.gnu.org/projects/patch/

[2] D. Zhao, S. M. Furnell, and A. Al-Ayed, "The research on a patch management system for enterprise vulnerability update," in *2009 WASE International Conference on Information Engineering*, vol. 2, 2009, pp. 250–253.

[3] J.-T. Seo, D.-S. Choi, E.-K. Park, T.-S. Shon, and J. Moon, "Patch management system for multi-platform environment," in *Parallel and Distributed Computing: Applications and Technologies*, K.-M. Liew, H. Shen, S. See, W. Cai, P. Fan, and S. Horiguchi, Eds. Berlin,

Heidelberg: Springer Berlin Heidelberg, 2005, pp. 654–661.

[4] J.-T. Seo, Y.-j. Kim, E.-K. Park, S.-w. Lee, T. Shon, and J. Moon, "Design and implementation of a patch management system to remove security vulnerability in multi-platforms," in *Fuzzy Systems and Knowledge Discovery*, L. Wang, L. Jiao, G. Shi, X. Li, and J. Liu, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 716–724.

[5] Chuan-Wen Chang, Dwen-Ren Tsai, and Jui-Mi Tsai, "A cross-site patch management model and architecture design for large scale heterogeneous environment," in *Proceedings 39th Annual 2005 International Carnahan Conference on Security Technology*, 2005, pp. 41–46.

[6] C. Higby and M. Bailey, "Wireless security patch management system," in *Proceedings of the 5th Conference on Information Technology Education*, ser. CITC5 '04. New York, NY, USA: Association for Computing Machinery, 2004, p. 165–168. [Online]. Available: https://doi.org/10.1145/1029533.1029575

[7] H. Cavusoglu, H. Cavusoglu, and J. Zhang, "Security patch management: Share the burden or share the damage?" *Management Science*, vol. 54, no. 4, pp. 657–670, 2008. [Online]. Available: http://www.jstor.org/stable/20122418

[8] H. Cavusoglu and J. Zhang, "Economics of security patch management," in *WEIS*, 2006.

[9] H. Okhravi and D. Nicol, "Evaluation of patch management strategies."

[10] B. Brykczynski and R. A. Small, "Reducing internet-based intrusions: Effective security patch management," *IEEE Software*, vol. 20, no. 1, pp. 50–57, 2003.

[11] G. Y. Zhang, "Three essays on managing information systems security: Patch management, learning dynamics, and security software market," Ph.D. dissertation, USA, 2007.

[12] T. Gerace and H. Cavusoglu, "The critical elements of the patch management process," vol. 52, no. 8, 2009. [Online]. Available: https://doi.org/10.1145/1536616.1536646

[13] [Online]. Available: https://git.kernel.org/

[14] [Online]. Available: https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html

[15] [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git

[16] [Online]. Available: https://source.codeaurora.org/quic/la/kernel/msm-3.10/

[17] V. Levenshtein, "Binary Codes Capable of Correcting Deletions, Insertions and Reversals," *Soviet Physics Doklady*, vol. 10, p. 707, 1966.

[18] [Online]. Available: https://github.com/google/diff-match-patch

[19] [Online]. Available: https://bugs.chromium.org/p/chromium/issues/list

[20] [Online]. Available: https://vivaldi.com/source/vivaldi-source_2.10.1745