Towards Understanding and Fixing Upstream Merge Induced Conflicts in Divergent Forks: An Industrial Case Study

Chungha Sung
University of Southern California
Los Angeles, CA, USA

Shuvendu K. Lahiri
Mike Kaufman
Pallavi Choudhury
Microsoft Corporation
Redmond, WA, USA

Chao Wang
University of Southern California
Los Angeles, CA, USA

ABSTRACT

Divergent forks are a common practice in open-source software development to perform long-term, independent and diverging development on top of a popular source repository. However, keeping such divergent *downstream* forks in sync with the *upstream* source evolution poses engineering challenges in terms of frequent merge conflicts. In this paper, we conduct the first industrial case study of the implications of frequent merges from upstream and the resulting merge conflicts, in the context of Microsoft Edge development. The study consists of two parts. First, we describe the nature of merge conflicts that arise due to merges from upstream and classify them into textual conflicts, build breaks, and test failures. Second, we investigate the feasibility of automatically fixing a class of merge conflicts related to *build breaks* that consume a significant amount of developer time to root-cause and fix. Towards this end, we have implemented a tool MrgBldBrkFixer and evaluate it on three months of real Microsoft Edge Beta development data, and report encouraging results.

ACM Reference Format:
Chungha Sung, Shuvendu K. Lahiri, Mike Kaufman, Pallavi Choudhury, and Chao Wang. 2020. Towards Understanding and Fixing Upstream Merge Induced Conflicts in Divergent Forks: An Industrial Case Study. In *Software Engineering in Practice (ICSE-SEIP '20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3377813.3381362

The first author was an intern at Microsoft for the course of the study.

ICSE-SEIP '20, May 23–29, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7123-0/20/05…$15.00
https://doi.org/10.1145/3377813.3381362

1 INTRODUCTION

During software development, a *fork* occurs when software code is copied and used as the starting point of an independent development, thus creating a distinct and separate piece of software. While some forks are created to allow developers to work independently on authoring and testing changes (e.g., new features, refactorings, and bug fixes) to be eventually merged back to the "master" branch, other forks are created to carry out long-term, independent, diverging development on top of the original source code. In the latter case, we call the fork a *divergent fork*.

Unlike a branch that is often short-lived, a divergent fork may live permanently along side the original project. However, flow of information between the original and forked repositories is asymmetric. While most divergent forks need to continuously integrate changes from the original repository, e.g., to keep up with important security patches, changes from the forked repositories seldom flow back into the original repository. To signify this asymmetric nature, we refer to the original repository as the *upstream* and the forked repository as the *downstream*.

Divergent forks are a common practice in open-source development, e.g., to provide customized products by adapting an open-source project. Leveraging an upstream software that defines or adheres to some standards (e.g., Android) allows the downstream software to offer better application compatibility. As an example, web browsers such as Opera, Samsung Internet, and Microsoft Edge build upon the Chromium engine; similarly, customized versions of the Android mobile operating system are offered by various smartphone vendors, together with their own applications.

Although popular and convenient, a divergent fork may incur significant overhead. One challenge is to keep the downstream synchronized with important updates in the upstream. As the upstream software evolves due to API changes and security patches, the downstream needs to be updated accordingly. That is, the downstream needs to perform a *merge* from the upstream. Unfortunately, such a merge may fail due to syntactic conflicts or semantic conflicts that can lead

Table 1: The Chromium version used by Chromium-based web browsers as of October 3, 2019.

Browser Name	Version	Browser Name	Version
Microsoft Edge	78	Avast	77
Brave	77	Vivaldi	77
Colibri	76	Iron	76
Epic	75	Opera	73
Samsung Internet	71	Blisk	70
LG WebOS (TV) web engine	53		

to build breaks and test failures [2, 6, 19, 22, 23]. Manually resolving these conflicts is labor intensive.

Table 1 shows the Chromium versions used in various Chromium-forked web browsers as of October 2019. At the time, the latest branch version of Chromium was 78 but most browsers lagged behind by at least one or two versions. Given that supporting frequent merges from the upstream is expensive, we speculate that most vendors either chose to update less frequently, or budget additional developer time to perform such merges frequently.

While merge conflicts are not unique to divergent forks [2, 6, 19, 22], the complexity and cost of root-causing and fixing the asymmetric *upstream merge induced conflicts* is significantly higher, for three reasons: (1) Changes in the upstream often occur without knowledge of the downstream development. (2) Root-causing the upstream commit responsible for merge conflict in general, and build break in particular, is non-trivial when the commit history of the upstream consists of several thousand commits. (3) A merge induced build break may also be caused by changes in the downstream, often made many commits earlier. This makes it difficult to find the right developer to assign the fix, e.g., if the developer has left the project.

In this work, we study the problem of upstream merge induced conflicts in the context of Microsoft Edge development. Microsoft recently adopted the open-source Chromium project in the development of the Edge browser in order to increase compatibility and reduce fragmentation for web developers [16]. In the remainder of this paper, we may refer to Chromium as the upstream and Edge as the downstream. Our case study has three main contributions.

- We systematically investigate the downstream commits performed by Edge developers to merge from Chromium during a three-month period, and create a taxonomy of the merge conflicts.
- We systematically investigate the repairs that developers have to make manually to resolve these conflicts. We identify a particular sub-class of merge conflicts, named *Structural fixes in C++ files*, which incurs substantial time for developers to root-cause and fix.
- We investigate the feasibility of generating repairs automatically. Toward this end, we develop and evaluate a repair tool, named MrgBldBrkFixer, for *Structural fixes in C++ files*.

In the remainder of this section, we explain *Structural fixes in C++ files* and MrgBldBrkFixer in more detail.

Structural fixes in C++ files. Consider a method *Foo* that was defined and used in the upstream when the downstream was created. Then, the downstream created some new call-sites for *Foo*. At some point during the upstream development, however, *Foo* was renamed to *Bar*, a new parameter was added, and then all the call-sites were properly changed. While merging such a change into the downstream does not cause syntactic conflicts (since the downstream may not change these files), compilation will fail and cause a *build* break. During our case study, we observe many such conflicts. Furthermore, the root-cause is often not obvious to downstream developers. Often times, developers have to manually inspect the upstream commits (which can be a few thousands, as shown in Section 2), analyze the change impact, and then create a suitable patch, e.g., renaming the method and the default value of the additional parameter.

MrgBldBrkFixer. We would like to know how much the repair of merge conflicts can be automated. Toward this end, we develop a prototype tool for repairing the sub-class of *Structural fixes in C++ files* errors. The tool relies on differencing the Abstract Syntax Trees (ASTs) of the two programs [7] to identify the changes in the upstream for a given symbol (say *Foo* in the above example) and then creates a patch that can be applied to the downstream. To improve the scalability and accuracy of the tool, we propose techniques for soundly pruning the irrelevant upstream commits. Using real development data of Microsoft Edge collected in a three-month period, we perform a feasibility study of MrgBldBrkFixer. The result shows that 40% of the build breaks targeted by MrgBldBrkFixer can be repaired automatically.

The remainder of the paper is organized as follows: First, we present our study of the upstream merge induced conflicts in Edge development in Section 2, and our detailed analysis of *Structural fixes in C++ files* in Section 2.4. Next, we present MrgBldBrkFixer in Section 3, followed by our feasibility study results of MrgBldBrkFixer in Section 3.3. We review the related work in Section 5, and then give our conclusions in Section 6.

2 STUDY OF UPSTREAM MERGE-INDUCED CONFLICTS IN EDGE

In this section, we study the upstream merge-induced conflicts in the context of Microsoft Edge development, a recent divergent fork of Chromium. We first describe the Edge branch structure related to such upstream merges in Section 2.1. Next, we present the data for the merges during a

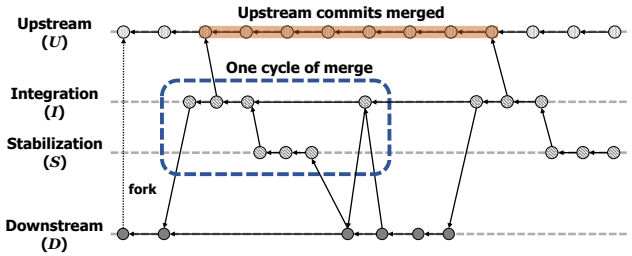


Figure 1: The branch structure of Microsoft Edge.

three-month period from April 2019 to June 2019. We manually investigated the data and classify the nature of conflicts based on the commits that fixed the conflicts.

2.1 Branch Structure of Microsoft Edge

Figure 1 gives an overview of (somewhat simplified) branch structure in Microsoft Edge. Each horizontal line represents one of the four branches: *Upstream* (*U*), *Integration* (*I*), *Stabilization* (*S*) and *Downstream* (*D*). Circles in each branch indicate commits created by developers and an arrow points from a child commit to its parent commit. Here the *D* branch denotes the master branch of Edge, and the *U* denotes the Chromium master. The "fork" indicates the creation of the *D* branch as a divergent fork of *U*, and both branches evolve independently.

At each merge, the downstream pulls the changes from the upstream in a two-phase process through the *I* and *S* branches. First, textual (syntactic) conflicts are resolved in the *I* branch after pulling the changes from the recent versions of the upstream and downstream. After resolving the textual conflicts, any build errors (including compiler errors) or test failures are resolved in the *S* branch. Finally, the source code is merged back to *D* master, where one cycle of merge is completed. We omit the details of finalizing the merge cycle as they are irrelevant to our focus in this paper.

2.2 Commit Data

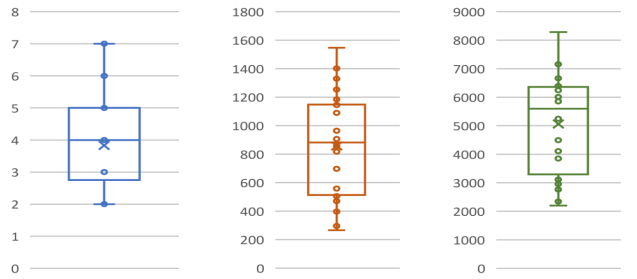
2.2.1 Breakup by month. Table 2 shows the summary statistics of the data by each month. The first row shows the number of merges from *U* each month. Each merge represents a merge process to pull the upstream changes after resolving all the conflicts. It means each merge includes one cycle of resolutions for textual-level conflicts, build breaks and test failures (if any). The second row is the number of commits of *U* that are merged into *D*, which is shown as orange-colored region, as an example, in Figure 1. The third row shows the number of commits in *I*, for fixing any textual-level conflict that prevents the textual merge. The last row is the number of commits in *S*, corresponding to the resolutions of build breaks and test failures.

Table 2: The summary of data.

Contents	Numbers for Each Month		
	April 2019	May 2019	June 2019
The # of merges	11	8	11
The # of upstream commits merged	8,138	9,581	8,031
The # of commits in *integration branch* of downstream	286	560	337
The # of commits in *stabilization branch* of downstream	325	357	353

In total, there are 30 merges over three months, and more than 25,000 upstream commits that merged to downstream over this period. For each month, an average of around 390 (respectively, 345) commits are made to resolve textual-level conflicts (respectively, resolve build breaks and test failures).

2.2.2 Breakup by merge. Figure 2 provides statistics about the upstream payload of each merge in terms of the number of days, commits and files updated. Each merge consumes only a few days (between 2 and 7 days) of upstream changes. Within this short period of time, the number of upstream commits ranges from 266 to more than 1500, updating several thousand files. For example, the five-number summary of the commits is: Min(266), Q1(514), Median(881.5), Q3(1145) and Max(1547). It implies that (1) Chromium evolves rapidly by making many code changes, and (2) the downstream fork can easily lag behind without frequent merges.



(a) # of days **(b) # of commits** **(c) # of files updated**

Figure 2: Statistical information of the upstream merged for each merge process.

2.3 Classification of Fix Commits

We carefully classify the *fix comments*, which are the downstream commits that resolve the upstream merge induced conflicts. Table 3 shows a taxonomy of the resolved conflicts, together with some patterns that we find. In the three months, there are 2,218 commits to resolve all the merge induced conflicts. Recall that we consider three types of conflicts: textual conflict, build break and test failure. The percentage of each category is shown in the last column of the table: The number of fix commits for textual conflict is 1183, for build breaks is 815, and for test failures is 220. We describe each type in more detail in the next few paragraphs.

2.3.1 Textual conflict resolution. Textual conflict occurs when the default textual merge algorithm (e.g., Git merge) for the version control system cannot create a merged file from the upstream and downstream change-sets of a file. These textual conflicts are resolved by developers to obtain a merged file. Recall that such fixes happen in the *I* branch of Edge; the data show that 53% of its fix commits are used to resolve textual conflicts. Although textual conflicts are frequent, we do not further focus on them in this particular study because the nature of these conflicts and their resolution (e.g., using structured merge tools [10]) have been well-studied.

2.3.2 Build break resolution. A build break occurs when the build script fails to generate the executables. Some of the build breaks are due to failures in parsing files, while others are due to the inability to resolve a symbol during compilation. Based on the three-month data, we have studied how build breaks are resolved and identified three categories: (i) *Fixes of ill-formed files*, (ii) *Build script file fixes* and (iii) *Structural fixes in C++ files*. In total, 36% of the downstream fix commits are related to build breaks.

The category *Fixes of ill-formed files* refers to fixes needed to correct various syntax errors. Most commits in this category have messages such as "Fixing a bad merge" or "Reverting changes due to a bad merge". Many of these fixes undo or modify the changes made earlier during the textual conflict resolution. These fixes can be further divided into sub-classes. One sub-class, called *stylelint errors*, results from automatic enforcement of coding conventions using style linters. Another sub-class, called *parse errors*, involves the fixes of broken brackets and parenthesis. In addition to the 50 fixes that all into the above two sub-classes, there are 108 fixes that cannot be accurately characterized. The reason is because many of them happen when the entire code (e.g., class or function) is over-written by some old code during the textual conflict resolution.

The category *Build script file fixes* refers to changes made in the *build script* files. Large source code systems are compiled using various scripts/systems written in Python, JavaScript, NinjaBuild or CSV — we label them as script files. Such files undergo changes when the structure of the directory or flags defined in the source code are changed. In the three months, there are 259 fix commits related to the build scripts.

Finally, the category *Structural fixes in C++ files* refers to changes to the structural elements of the source code files. Since C++ is the main language used for Microsoft Edge development, we focus on errors in these C++ files. Often times, the errors are compiler errors due to failed resolution of symbols. We call them *Structural Build Fixes* because the fixes perform some structural code changes such as changes to function calls, types and namespaces. In the three months, there are 398 fix commits in this category. We also find some patterns. For example, when a directory structure is changed, the header include statement needs to be updated accordingly. Also, some API usages need to be updated due to changes of the API definitions in upstream.

Table 3: Distribution of fixes during merges.

Conflict	Details	# of commits	%
Textual	Textual-level conflict resolution	1183	53.3
Build Break	Fixes of ill-formed files	158	36.7
	- Stylelint fixes	(31)	
	- Parse error fixes	(19)	
	- Uncategorized	(108)	
	Build script file fixes (.js, .gn, .json, etc.)	259	
	Structural fixes in C++ files	398	
Test Failure	Macro fixes in unit test	169	9.9
	Flag file fixes	51	
Total		2218	

```
102   ...
103 - IN_PROC_BROWSER_TEST_F(OmniboxViewViewsTest, PastAndGoAcce){
    + IN_PROC_BROWSER_TEST_F(OmniboxViewViewsTest,
    +     TEST_DISABLED_TRIAGE(PasteAndGoAccelerator, 22305207)) {
104     OmniboxView* view = NULL;
```

Figure 3: Example change to disable macro in unit test.

```
52    ...
53    {"enable-service-worker-long-running-message", true},
    + {"enable-sharing-device-registration", false}
54    {"enable-show-autofill-signatures", true},
```

Figure 4: Example change to adding flag in flag file.

2.3.3 Test failure resolution. Once the project builds successfully, e.g., after the build break resolution, there may still be failures during the execution of unit and integration tests. Fixes in this category fall in two sub-categories. The first sub-category is *Macro fixes in unit test*. For example, as the downstream is customized from the upstream, it may not need all the features. Therefore, some unit tests from the upstream need to be disabled. An example code change found in during our case study is shown in Figure 3. Since the test macro `IN_PROC_BROWSER_TEST_F` with `OmniboxViewViewsTest` is no longer needed, the downstream developers disabled it in line 103. As usual, `-` indicates that the line is removed and `+` indicates that the line is added. In the three months, there are in total 169 commits related to updating/adding/disabling unit tests.

The second sub-category is *Flag file fixes*, which resets certain flags (typically maintained in a special flags file) that cause test failures in the downstream. Figure 4 shows an example flag `enable-sharing-device-registration`, which was introduced by the upstream, and the downstream developers disabled it by resetting the flag due to test failures. In total, there are 51 fix commits related to flags similar to the above example.

2.4 Structural Build Fixes in C++ Files

In this section, we focus on a specific sub-category *Structural fixes in C++ files* of build breaks that are induced when the

merge is performed (see Table 3). Most fixes in this category share a common pattern that we describe informally as follows: Consider a structure element S (e.g., a field in a class, function or a namespace) with a given signature (identified by its name and type) that is defined in the upstream with uses in both the upstream and downstream. At some point in time, an *upstream commit* changes the signature of S and updates all the uses of S in the upstream code. When such a commit is merged into the downstream, the uses of S introduced by the downstream code can no longer be resolved.

Figure 6a illustrates this situation using a fix commit found in Microsoft Edge (the downstream) to fix a build break on May 6th. The build break complains about the `selected_index` function being undefined. In the fix commit, the downstream developers updated the function call `selected_index` into `GetSelectedIndex`. Figure 6b shows the corresponding upstream commit that induces this build break: the upstream commit made on May 2nd changed the function name from `selected_index` to `GetSelectedIndex`, and redirected all its call sites in upstream code.

We focus on this class of merge induced conflicts because, among all classes of fix commits, developers found it to be the most laborious to identify the root-cause and prepare the fix for the downstream code. We attribute this to the following reasons:

- A developer needs to scan the (possibly thousands of) upstream commits merged to identify the relevant changes that induce the build break. This includes not just understanding how the definition is changed, but also how the uses are changed. For example, if the upstream introduces a new parameter to a function and sets it to `null` at all but one of the call sites, the most likely patch downstream is to pass `null` as an additional parameter.
- Unlike other merge conflicts, these build breaks can manifest even in the absence of any downstream changes since the last merge. Consider the case when the compiler cannot resolve the use of a symbol S introduced in the downstream several hundred merges back (which may span across months to years), but updated in the latest upstream commits. The developer who introduced the use in the downstream may not have all the context, or may not even be available. Therefore, even finding the right person to investigate the fix is not easy.

The situation is accurately captured in a quote from a senior manager in the Microsoft Edge development team:

> *"For each upstream induced build break, it takes at least 30 minutes to hours for developers to resolve. The main burden for the developers is they need to look up the history of upstream changes."*

Furthermore, we believe that many of the merge conflicts in the other categories can be addressed by tools available (e.g., structure-aware textual merge conflict resolution tools [1, 10]) and partially automated with custom knowledge of patterns (e.g., in the case of flags involved in test failures). In fact, at the time of this study, the Microsoft Edge team has regular expression based fixes for many commonly-known patterns. In contrast, as we have illustrated in this section, the causes of structural build breaks in C++ files can be quite varied and therefore require a deeper, AST-aware analysis.

In the remainder of this section, we shall identify various common sub-categories of the structural build breaks in C++ files. Table 4 shows a list of resolution cases in the downstream, divided into eight groups by common example causes from the upstream: *Include Statement Update*, *Entire Function Definition/Call Update*, *Function Name Update*, *Function Type/Specifier Update*, *Function Parameter/Argument Update*, *Function Parameter/Argument's Type Update*, *Class/Namespace/Enum Reference Update*, and *Uncategorized*. While the example causes from the upstream are provided to help understand the breaks, they are not meant to be exhaustive. Also, we report the number of commits for each group in the last column. Some commits are counted in multiple groups, as one commit may have several fixes. Since the classification is inherently manual, there exists a set of commits for which we could not find the exact patterns with possible causes; therefore, it is classified as *Uncategorized* (Group 8). In the remainder of this section, we present examples of five cases for some groups in the table due to space limitation.

2.4.1 Include Statement Update. As an example of Group 1, Figure 5 shows a fix commit to fix a build break on May 15th. The include statement for a header in line 26 is updated with a new header path. The reason is the directory `chrome_elf` was moved to outside of `chrome` directory in upstream, so downstream source code referring the directory had to be updated.

```
24    ...
25    #include "base/win/win_util.h"
26  - #include "chrome/chrome_elf/chrome_elf_main.h"
    + #include "chrome_elf/chrome_elf_main.h"
27    #include "chrome/install_static/install_util.h"
```

Figure 5: Header statement resolution in downstream on May 15th (commit: df9e775a).

2.4.2 Function Name Update. We have already described this particular example in Figure 6, which shows the the fix commit in the downstream with a possible cause in the upstream. It is classified as Group 3 in the table.

2.4.3 Function Parameter/Argument's Type Update. As an example of Group 6, Figure 7b shows an upstream cause where the signature of a "virtual" method `SubscribeFromWorker`

Table 4: List of resolution cases in upstream induced build break.

Group	Type of Resolution/Fixes in Downstream	Possible (Example) Causes from Upstream	# of commits
1	Include Statement Update	- File/Directory Name/Structure is updated	62
2	Entire Function Definition/Call Update (e.g., function body move/add/removal)	- Function definition (with body) is added/removed - Function definition (with body) moved to different class/section (e.g., public → private)	35
3	Function Name Update	- Function name is updated - Entire function is removed (e.g., function deprecation)	56
4	Function Type/Specifier Update	- Function definition type is updated - Function definition specifier is added/removed	10
5	Function Parameter/Argument Update (e.g., parameter/argument add/remove/reorder)	- Function definition parameter is added/removed/reordered	44
6	Function Parameter/Argument's Type Update (e.g., parameter/argument type update)	- Function definition parameter's type is updated - Function definition parameter's specifier/modifier is added/removed - Hierarchy/Name of Class/Namespace/Enum definition is updated	56
7	Class/Namespace/Enum Reference Update (e.g., field type update)	- Hierarchy/Name of Class/Namespace/Enum definition is updated	53
8	Uncategorized		120

```
166    ...
167    views::Combobox* combobox = GetLanguageCombobox();
168  - if (model_->GetTargetLanguageIndex() == combobox->selected_index() {
     + if (model_->GetTargetLanguageIndex() == combobox->GetSelectedIndex() {
169      return;
170    }
```

(a) Downstream commit on May 6th (commit: 7abf5c10).

```
102    ...
103    void ModelChanged();
104  - int selected_index() const { return selected_index_; }
     + int GetSelectedIndex() const { return selected_index_; }
105    void SetSelectedIndex(int index);
```

(b) Upstream commit on May 2nd (commit: 0b079bf5).

Figure 6: Function name change in downstream with upstream cause.

```
43     ...
44     void SubscribeFromWorker(const GURL& requesting_origin,
45                              int64_t service_worker_registration_id,
46                              const PushSubscriptionOptions& options,
       -                        const RegisterCallback& callback) override;
47     +                        RegisterCallback callback) override;
```

(a) Downstream commit on May 6th (commit: 582db1e8).

```
75     ...
76     virtual void SubscribeFromWorker(const GURL& requesting_origin,
77                              int64_t service_worker_registration_id,
78                              const PushSubscriptionOptions& options,
79   -                          const RegisterCallback& callback)
     +                          RegisterCallback callback)
```

(b) Upstream commit on May 2nd (commit: b534bf78).

Figure 7: Function parameter's specifier/modifier removal in downstream with upstream cause.

in a base class is updated by removing the specifier and modifier of the 4th parameter. Uses of this method include methods in the inherited classes as well, some of which may have been introduced in the downstream.

Figure 7a shows the corresponding fix commit in the downstream that changes the signature of one such method in a derived class, which was not visible to the upstream. Note that the complete set of fixes should also include changing the argument that is passed at the call sites (not shown in this figure).

```
46     ...
47   - base::TaskScheduler::GetInstance()->FlushForTesting();
     + base::ThreadPool::GetInstance()->FlushForTesting();
```

(a) Downstream commit on April 17th (commit: 9327111c).

```
37     ...
38   - class BASE_EXPORT TaskScheduler: public TaskExecutor {
     + class BASE_EXPORT TaskScheduler: public TaskExecutor {
39       public:
```

(b) Upstream commit on April 15th (commit: 52fa3aed).

Figure 8: Class name change in downstream with upstream cause.

2.4.4 Class Reference Update. As an example of Group 7, Figure 8a shows a fix commit for an unresolved reference of a class name `TaskScheduler`. The fix renames the class from `TaskScheduler` to `ThreadPool`, as a response to an upstream commit 8b that introduced such a change in the first place.

2.4.5 Enum Reference Update. As another example of Group 7, Figure 9a shows the downstream fix commit that changes the name of an enum class, due to the change of the upstream shown in Figure 9b.

3 TOWARDS AUTOMATIC FIXES FOR UPSTREAM-INDUCED BUILD BREAKS

To evaluate the feasibility of automated fixes of merge induced build breaks, we develop a prototype tool named MrgBldBrkFixer. Our focus is on an important sub-class of fixes, called *renaming fixes*, where the build breaks can be resolved by either (i) renaming a function, class, namespace or enum, or (ii) renaming the types of a function's parameters or the return value. While no exhaustive, these fixes already cover a significant number of build breaks in Groups 3, 4, 6 and 7 of Table 4.

```
88    ...
89    base::ThreadPool::GetInstance()->Start(
90      {{kBackgroundMaxThreads, kSuggestedReclaimTime},
91      {kForegroundMaxThreads, kSuggestedReclaimTime},
92  -   base::ThreadPool::InitParams::SharedWorkerPoolEnvironment::COM_MTA});
    -   base::ThreadPool::InitParams::CommonThreadPoolEnvironment::COM_MTA});
```

(a) Downstream commit on May 7th (commit: cc7f9934).

```
73    ...
74    struct BASE_EXPORT InitParams {
75  -   enum class SharedWorkerPoolEnvironment {
    +   enum class CommonThreadPoolEnvironment {
```

(b) Upstream commit on April 30th (commit: 3e2898f0).

Figure 9: Enum name change in downstream with upstream cause.

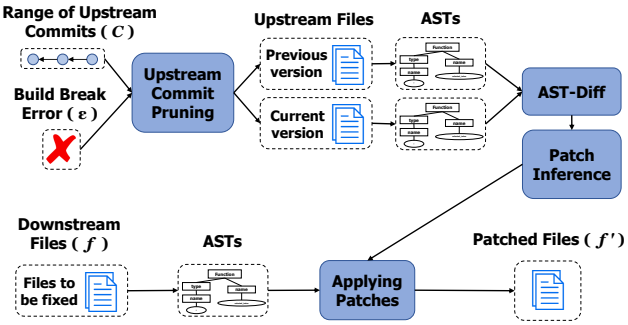


Figure 10: Overview of the automated patching.

Figure 10 presents the overview. The input consists of (i) a set of upstream commits, C, that constitutes the merge, and (ii) a build break error, ϵ, in a downstream C++ file f. The output is the patched downstream file f' aimed to resolve the build break. Internally, there are four steps:

(1) Identify the symbol σ in f that is responsible for the build break error ϵ.
(2) Prune the upstream commits in C to remove the ones not relevant to σ, to obtain $C' \subseteq C$.
(3) Analyze changes to definitions and uses (Defs and Uses) in the files modified in C', to infer a set of possible *renaming patches*, denoted Π.
(4) For each patch $\pi \in \Pi$, apply π to the AST node (in f) that contains σ, to obtain f'.

We explain the patch generation process in more detail in Section 3.1, while deferring the discussion of "pruning upstream commits" to Section 3.2.

3.1 Patch Generation

In a nutshell, our patch generation algorithm searches for changes to the definitions and uses of σ in the set C' of upstream commits. The notion of "use" depends on the nature of the symbol, e.g., whether it is a function, class, namespace, or enum. In addition to considering references to a symbol (e.g., function call for a function, or class reference for a class), we consider implicit usages due to the presence of inheritance. For a virtual function F inside a class J, we consider any override function F in a class V that derives from K as a potential usage of F in J.

Let Δ be a set of diff-regions obtained when comparing the files before and after the commits in C. Let each $\delta \in \Delta$ be a diff region that consists of a pair (δ^-, δ^+) of enclosing AST nodes before and after a change. We associate a region in a file with the smallest AST node that encloses the region; and we associate a dummy AST node *null* for an empty region. Let $\Delta_d \subseteq \Delta$ (respectively, $\Delta_u \subseteq \Delta$) be the subset of diffs that contains changes to the definitions (respectively, uses) of σ. Let $\Delta_\sigma \doteq \Delta_d \cup \Delta_u$.

3.1.1 Function changes. Let us first consider the case when the symbol σ represents a function. Since σ is a function, we look for any diff region $\delta \in \Delta_\sigma$ that contains a change to the function *signature*. This includes regions that change (i) the function name, (ii) the return type, or (iii) the types of function parameters.

For ease of comprehension, we define three predicates. For any diff node δ, let $IsNameChange(\delta)$ return TRUE if δ is a change of a function name, let $IsParTypeChange_i(\delta)$ return TRUE if δ is a change of the i^{th} parameter of a function, and let $IsRetTypeChange(\delta)$ return TRUE if δ is a change of the return type of a function.

Given Δ_d and Δ_u, we generate the set Π of candidate patches as follows. We initialize Π to the empty set. Then, for each $(\delta_d, \delta_u) \in \Delta_d \times \Delta_u$, we update Π as follows:

(1) If $IsNameChange(\delta_d)$ and $IsNameChange(\delta_u)$, then add (RENAMENAME, σ, ${\delta_u}^+$) to Π.
(2) If δ_u changes a function definition that overrides σ defined in ${\delta_d}^-$, then
  (a) add (RENAMERETTYPE, σ, ${\delta_u}^+$) to Π if $IsRetTypeChange(\delta_d)$ and $IsRetTypeChange(\delta_u)$.
  (b) add (RENAMEPARAMTYPE$_i$, σ, ${\delta_u}^+$) to Π if $IsParTypeChange_i(\delta_d)$ and $IsParTypeChange_i(\delta_u)$.

Note that we cannot simply use changes in the definition Δ_d nodes to generate the patches. For example, for the case when an entire function definition is deleted and the function uses are renamed (i.e., due to function deprecation), we need to inspect the changes of uses in upstream ${\delta_u}^+$ to infer the patch.

3.1.2 Enum, Class and Namespace changes. We also consider the cases when the name of an enum, a class, or a namespace changes. Such changes may denote either direct renaming of the type, or changing the hierarchy in which the entity is defined. For this case, our algorithm for generating a patch is similar to the case of renaming the function name; due to the space limit, we omit the details.

3.2 Upstream Commit Pruning

The patch generation algorithm needs to construct the AST for each of the files modified in the upstream commits C

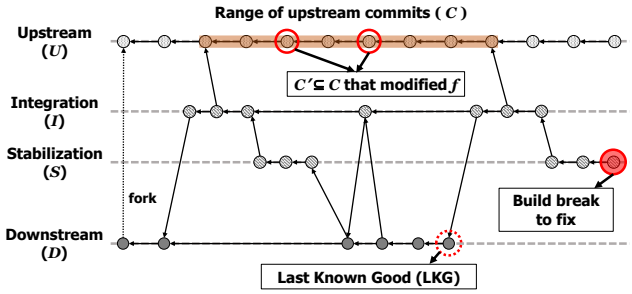


Figure 11: Components of upstream commit pruning

and then inspect the diff-regions. As shown in Figure 2, the number of modified files in the upstream for each merge process can be thousands (2,000~8,000). The presence of such a large number of files can reduces both the scalability and the accuracy of the algorithm. While their scalability impact is obvious, the accuracy impact is due to the introduction of spurious patches, e.g., when we use techniques that do not yield the most qualified type for each symbol due to the best-effort construction of AST using srcML [3]. While future prototypes can leverage high-fidelity ASTs, e.g., the ones constructed by the CLANG compiler, it will be at the cost of a longer build time.

To alleviate this problem, we developed an *Upstream Commit Pruning* procedure, shown in Figure 10. Given a build break, and a set of upstream commits C (orange-colored region in Figure 11, the procedure does the following:

(1) Find the symbol σ responsible for the build break (such as `selected_index` in Figure 6a).
(2) Find the file f that contains the definition of σ, using the *last known good* (LKG) build (dashed-red circle in the figure) in the downstream by navigating the def-use chain of the symbol. The LKG commit can be obtained by searching for the first ancestor in the downstream starting from the build break commit in the stabilization branch.
(3) Find the subset of commits $C' \subseteq C$ that modify f.

It is easy to see that any change to the definition of σ has to be included in one of the commits C'. We additionally assume that the upstream also updates the uses of σ in the same commit; this is a reasonable assumption because developers would at least compile the code before submitting a commit. This assumption has been confirmed empirically using the real data set.

3.3 Evaluation Setup

In this section, we present the feasibility study by using MrgBldBrkFixer on the historical fix commits in Microsoft Edge, as discussed in Section 2.4. The prototype tool is written in Java while using srcML [3] to create the AST and GumTreeDiff [7] to perform AST-diff. All the steps outlined in previous subsections are implemented, except for the extraction of the symbol σ from the build error message ϵ, and the search of the file containing the definition of σ. (Currently, we do it by manually loading the project in Visual Studio and navigating to the definition).

Out of the 398 *Structural fixes in C++ files* in our three months of fix commits (Table 3), we use only the commits from Group 3, 4, 6 and 7 in Table 4, because our current prototype only produces fixes related to *renaming*. In other words, it does not yet handle fixes categorized as *Include statement* (Group 1), or fixes that require adding/removing/reordering parameters (Group 5). Moreover, for Group 6 and 7, our prototype obtains partial fixes, since some of these fixes require more complex analysis (e.g., relating the type of actual parameters at a call site with the type of formal parameter, or hierarchy change for enum/class/namespace).

Thus, we obtained a total of 164 candidate commits for our experimental evaluation. Our experiments were conducted in Microsoft Windows 10 Enterprise edition on a computer with an Intel i7 2.6 GHz CPU and 32 GB of RAM.

We considered the following research questions:

- How many of these manually resolved commits can actually be automated?
- How effective is the upstream commit pruning technique that we propose?

We answer these two questions in the remainder of this section.

3.4 Evaluation Results

Table 5: Auto-patch rate by groups in Table 4

Group Number	Fixed Ratio
Group 3	38/56 (67.8%)
Group 4	4/10 (40%)
Group 6	12/56 (21.4%)
Group 7	13/53 (24.5%)

3.4.1 Resolution Generation. MrgBldBrkFixer successfully generated the patches for 64 out of 164 resolution commits *without false positives*, almost 40% of the resolution commits for the categories that we target. Some of these patches are *partial fixes* because a commit may contain multiple resolutions, some of which may not be covered by our prototype (e.g., groups in Table 4 that we do not handle). Out of the 64 patches, our fixes fully cover the developer resolution for 41 commits, and partially cover for 23 commits. On average, each fix updates 2.5 downstream files since the same symbol σ may be renamed in multiple files. Our patch generation algorithm consumed, on average, 1.72 commits after applying the upstream commit pruning technique, and 48.25 files (including non C++ files).

MrgBldBrkFixer took 70.90 seconds on average to generate and apply the patches to downstream files. Given that developers often spend 30 minutes or more to fix each build break, this means our prototype tool can substantially improve the

productivity of developers who are trying to root-cause and fix the merge-related build breaks.

Table 5 shows the ratio of automatically fixed commits by the groups in Table 4. We can conclude that many cases of *Function Name Update* (Group 3) can be automated, the cases where our prototype is most effective, fixing almost 68% of cases. There are various reasons why our method could not generate a fix for all instances in a group. Many of them are due to limitations of our current prototype implementation, as opposed to any fundamental challenges. For example, we current do not support complex fixes such as hierarchy changes of functions, classes, enums and namespaces.

We also witnessed srcML-based GumTreeDiff sometimes produces diff regions that do not accurately captures the changes [5]. As an example, consider two functions whose names are changed as follows: `Foo1` → `Foo2`, and `Bar1` → `Bar2`. However, diff results may show the following changes: `Foo1` → `Bar2`, `Bar1` → `Foo2`. In addition, since srcML constructs a *best-effort* AST from the source code, it misses fully qualified semantic information such as class hierarchy. We plan to overcome some of the problems by integrating our tool with CLANG and making the diffing algorithm more semantics-aware.

Finally, we found one interesting class of fixes where focusing on the set of upstream commits since the previous merge (orange-colored region in Figure 11) does not suffice for patch generation; one needs to look at additional upstream commits prior to the last merge. We found cases of function deprecation fixes where upstream developers first changed only the function call sites of f with a different function g in a prior commit, and it did not cause any downstream conflict. However, the current merge removed the function definition (possibly as part of a cleanup) that caused a downstream build break at a call site of f. In this case, one needs to search for the upstream commit that removed the last reference of f in the upstream in order to synthesize a patch, which our current algorithm misses.

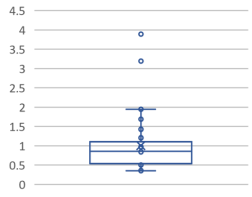


Figure 12: # of commits selected out of 1000 commits by upstream commit pruning

3.4.2 Impact of Upstream Commit Pruning. Figure 12 shows the reduction in the number of commits obtained by our upstream commit pruning technique. The graph shows the number of commits selected as relevant commits out of a normalized 1,000 upstream commits in a merge. Recall that the number of upstream commits typically range from 500 to more than 2,500 commits per merge (Figure 2).

This result illustrates that the pruning technique is able to achieve substantial reduction (almost 1000 fold on average) in the number of commits considered for our patch generation algorithm, which improves the scalability as well as the accuracy. We also manually inspected the reduced set of commits and fixes performed by the developers, to confirm that pruning did not unsoundly remove commits that contain the root-cause of a build break.

4 THREATS TO VALIDITY

In this work we studied the nature of merge conflicts that arise in a divergent fork such as Microsoft Edge. Our findings may not be representative of other divergent forks, as it may depend on how frequently merges are performed and the nature of changes that may be cherry-picked by developers (e.g., security patches only). However, we believe that most of the patterns of conflicts identified in this work arise in the setting of other divergent forks as well. Since we only studied three months of merge data, there is a small chance that the pattern of conflicts may evolve over time for Edge. However, as per the third author, who has expert knowledge of the Edge development, these conflicts are representative of conflicts in production since June 2019.

5 RELATED WORK

The effort to understand and resolve conflicts in cooperative merges within a project has been done in various contexts such as mining merge conflicts [2, 15, 26] or early detection of conflicts [2, 8, 17]. For more semantic conflicts (i.e. build breaks, test failures), several detection [23] and resolution approaches were proposed for preserving semantic relation of a program [6, 9, 21, 25]. In contrast to the nature of asymmetric merge relations in a divergent fork, all of the works focus on symmetric merges where the payload of each merge is relatively small. Our focus is to study the conflicts in asymmetric merges of divergent forks and investigate the feasibility of automatic fixes by utilizing the asymmetric flow. Also, API migration [4, 11, 13, 24] and applying same code change patterns [18] are not applicable to divergent forks. Constructing change patterns from a large number of commit history is not practical since divergent forks evolve independently, especially when upstream code evolves rapidly without providing documentations for detailed changes. Our approach combines upstream commit pruning and semantic rule generation to overcome the challenges.

The work closest to ours is by Mahmoudi et al. [12], who perform an empirical study of code changes of LineageOS as a response to upgrading the version of Android. They analyzed textual-level changes, and postulate some changes could be applied automatically; however, the work does not provide any algorithms or implementations for such fixes. In contrast, our work shows the actual fix patterns across the history of several merges, and provides the first implementation that is capable of generating patches in a real-world

production setting. Similarly, our work shares the underlying motivation of prior works on generalizing from a program edit to apply to other similar locations [14, 20]; however such approaches do not understand semantics of asymmetric merge and do not scale to the changes in our setting.

6 CONCLUSIONS

We have presented the first industrial case study of upstream merge induced conflicts in a divergent fork, namely the Microsoft Edge. We identified a class of conflicts, namely those requiring structural fixes in source files, that require substantial manual effort to root-cause and fix due to the scale of upstream commits that have to be considered. We provided a simple analysis based on constructing a patch for such conflicts by analyzing the changes upstream through an AST-aware diff. Our preliminary results are encouraging in that we are able to generate patches for almost 40% of the cases we consider as candidates.

We are currently working on extending the prototype tool to target more fixes. First, we plan to deal with the addition and removal of function parameters, by inspecting the arguments used at the upstream call sites. We also see examples where a lightweight data-flow analysis may help infer new arguments to a call when the type of a parameter has changed. Furthermore, we can extend the current algorithm to deal with updates of header file paths in include statements that appear frequently. Finally, we plan to eventually integrate our tool into the production merge resolution system of Edge development, and perform a user study of the effectiveness of the patch generation.

Acknowledgments. We thank Mark Marron, Hitesh Kanwathirtha, Nachi Nagappan, Jessica Wolk, and Madan Musuvathi for several insightful discussions around the problem.

REFERENCES

[1] S. Apel, J. Liebig, B. Brandl, C. Lengauer, and C. Kästner. 2011. Semistructured Merge: Rethinking Merge in Revision Control Systems. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*. 190–200.

[2] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin. 2011. Proactive Detection of Collaboration Conflicts. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*. 168–178.

[3] M. L. Collard and J. I. Maletic. 2016. srcML 1.0: Explore, Analyze, and Manipulate Source Code. In *IEEE International Conference on Software Maintenance and Evolution*. 649–649.

[4] B. Dagenais and M. P. Robillard. 2008. Recommending Adaptive Changes for Framework Evolution. In *International Conference on Software Engineering*. 481–490.

[5] G. de la Torre, R. Robbes, and A. Bergel. 2018. Imprecisions Diagnostic in Source Code Deltas. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. 492–502.

[6] C. R. B. de Souza, D. Redmiles, and P. Dourish. 2003. "Breaking the Code", Moving Between Private and Public Work in Collaborative Software Development. In *International ACM SIGGROUP Conference on Supporting Group Work*. 105–114.

[7] J. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus. 2014. Fine-grained and accurate source code differencing. In *IEEE/ACM International Conference On Automated Software Engineering*. 313–324.

[8] M. L. Guimarães and A. R. Silva. 2012. Improving Early Detection of Software Merge Conflicts. In *International Conference on Software Engineering*. 342–352.

[9] S. Horwitz, J. Prins, and T. Reps. 1989. Integrating Noninterfering Versions of Programs. *ACM Trans. Program. Lang. Syst.* 11, 3 (1989), 345–387.

[10] O. Leßenich, S. Apel, and C. Lengauer. 2015. Balancing precision and performance in structured merge. *Automated Software Engineering* 22, 3 (2015), 367–397.

[11] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk. 2013. API Change and Fault Proneness: A Threat to the Success of Android Apps. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*. 477–487.

[12] M. Mahmoudi and S. Nadi. 2018. The Android Update Problem: An Empirical Study. In *International Conference on Mining Software Repositories*. 220–230.

[13] T. McDonnell, B. Ray, and M. Kim. 2013. An Empirical Study of API Stability and Adoption in the Android Ecosystem. In *IEEE International Conference on Software Maintenance*. 70–79.

[14] N. Meng, M. Kim, and K. S. McKinley. 2011. Systematic editing: generating program transformations from an example. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. 329–342.

[15] T. Mens. 2002. A State-of-the-Art Survey on Software Merging. *IEEE Trans. Softw. Eng.* 28, 5 (May 2002), 449–462.

[16] Microsoft. 2018. Microsoft Edge: Making the web better through more open source collaboration. https://blogs.windows.com/windowsexperience/2018/12/06/.

[17] H. V. Nguyen, M. H. Nguyen, S. C. Dang, C. Kästner, and T. N Nguyen. 2015. Detecting semantic merge conflicts with variability-aware execution. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*. 926–929.

[18] Y. Padioleau, R. R. Hansen, J. L Lawall, and G. Muller. 2006. Semantic patches for documenting and automating collateral evolutions in Linux device drivers. In *Proceedings of the 3rd workshop on Programming languages and operating systems: linguistic support for modern operating systems*. 10–es.

[19] D. E. Perry, H. P. Siy, and L. G. Votta. 2001. Parallel Changes in Large-scale Software Development: An Observational Case Study. *ACM Trans. Softw. Eng. Methodol.* 10, 3 (July 2001).

[20] R. Rolim, G. Soares, L. D'Antoni, O. Polozov, S. Gulwani, R. Gheyi, R. Suzuki, and B. Hartmann. 2017. Learning Syntactic Program Transformations from Examples. In *International Conference on Software Engineering*. 404–415.

[21] D. Shao, S. Khurshid, and D. E Perry. 2009. SCA: a semantic conflict analyzer for parallel changes. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, 291–292.

[22] L. Silva. 2019. Detecting, Understanding and Resolving Build and Test Conflicts. In *International Conference on Software Engineering: Companion Proceedings*. 192–193.

[23] M. Sousa, I. Dillig, and S. K. Lahiri. 2018. Verified Three-way Program Merge. *Proc. ACM Program. Lang.* 2 (2018), 165:1–165:29.

[24] Z. Xing and E. Stroulia. 2007. API-Evolution Support with Diff-CatchUp. *IEEE Trans. Softw. Eng.* 33, 12 (Dec. 2007).

[25] W. Yang, S. Horwitz, and T. Reps. 1992. A Program Integration Algorithm That Accommodates Semantics-preserving Transformations. *ACM Trans. Softw. Eng. Methodol.* 1, 3 (1992), 310–354.

[26] T. Zimmermann. 2007. Mining Workspace Updates in CVS. In *International Workshop on Mining Software Repositories*. 11–11.