Social Coding Platforms Facilitate Variant Forks

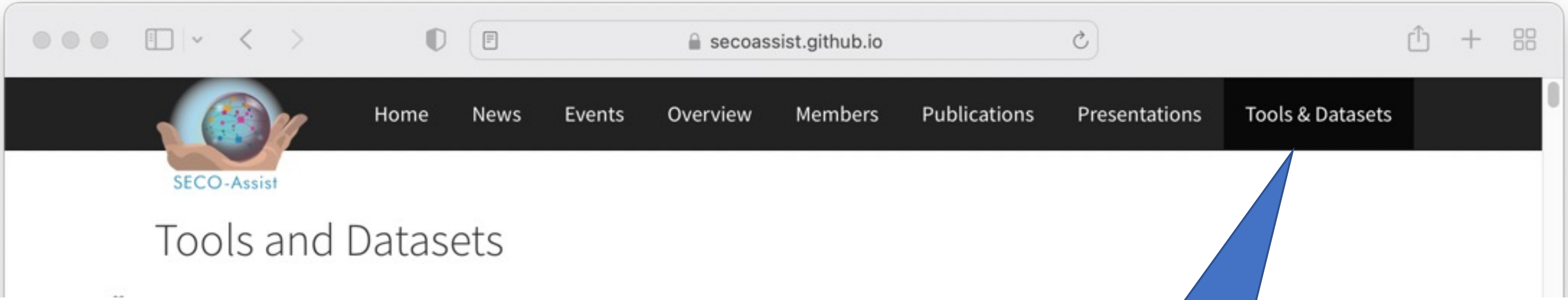
(Keynote REVE-WEESR 2022)

Prof. Serge Demeyer

AnSyMo

Universiteit Antwerpen

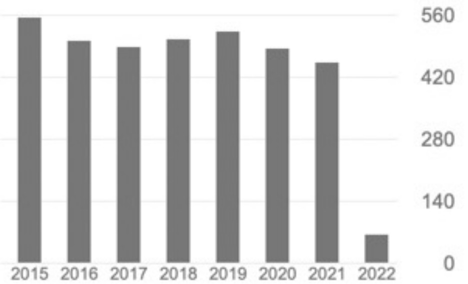
https://secoassist.github.io/



Open Science

Cited by VIEW ALL

	All	Since 2017
Citations	9287	2521
h-index	43	23
i10-index	107	57

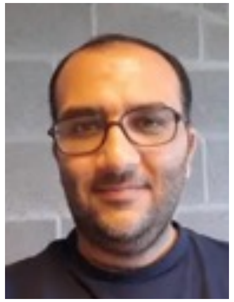
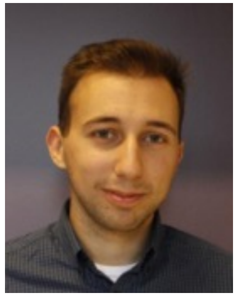


Collaborators

SECO-Assist

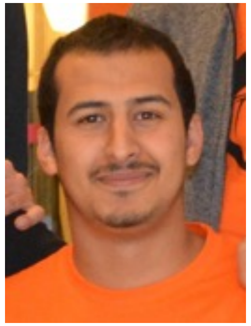
Universiteit Antwerpen



UMONS
Université de Mons


VUB VRIJE UNIVERSITEIT BRUSSEL




POLYTECHNIQUE MONTRÉAL
TECHNOLOGICAL UNIVERSITY

Variant Forks – Motivations and Impediments

John Businge,* Ahmed Zerouali,‡ Alexandre Decan,† Tom Mens,† Serge Demeyer,* and Coen De Roover,‡
*University of Antwerp, Antwerp, Belgium
{ john.businge | serge.demeyer }@uantwerpen.be
†University of Mons, Mons, Belgium
{ alexandre.decan | tom.mens }@umons.ac.be
‡ Vrije Universiteit Brussels, Brussels, Belgium
{ ahmed.zerouali | coen.de.roover }@vub.be

***Abstract*—Social coding platforms centred around git provide explicit facilities to share code between projects: forks, pull requests, cherry-picking to name but a few. Variant forks are an interesting phenomenon in that respect, as they permit for different projects to peacefully co-exist, yet explicitly acknowledge the common ancestry. Several researchers analysed forking practices on open source platforms and observed that variant forks get created frequently. However, little is known on the motivations for launching such a variant fork. Is it mainly technical (e.g., diverging features), governance (e.g., diverging interests), legal (e.g., diverging licences), or do other factors come into play? We report the results of an exploratory qualitative analysis on the motivations behind creating and maintaining variant forks. We surveyed 105 maintainers of different active open source variant projects hosted on GitHub. Our study extends previous findings, identifying a number of fine-grained common motivations for launching a variant fork and listing concrete impediments for maintaining the co-existing projects.**

***Index Terms*—Mainlines, Variants, GitHub, Software ecosystems, Maintenance, Variability**

I. Introduction

The collaborative nature of open source software (OSS) development has led to the advent of social coding platforms centred around the git version control system, such as GitHub, BitBucket, and GitLab. These platforms bring the collaborative nature and code reuse of OSS development to another level, via facilities like forking, pull requests and cherry-picking. Developers may fork a *mainline repository* into a new *forked repository* and take governance over the latter while preserving the full revision history of the former. Before the advent of social coding platforms, forking was rare and was typically intended to compete with the original project [1]–[6].

With the rise of pull-based development [7], forking has become more common and the community typically characterises forks by their purpose [8]. *Social forks* are created for isolated development with the goal of contributing back to the mainline. In contract, *variant forks* are created by splitting off a new development branch to steer development into a new direction, while leveraging the code of the mainline project [9].

Several studies have investigated the motivations behind variant forks in the context of OSS projects [1]–[6]. However, most have been conducted before the rise of social coding platforms and it is known that GitHub has significantly changed the perception and practices of forking [8]. In this social coding era, variant projects often evolve out of social forks rather than being planned deliberately [8]. To this end, social coding platforms often enable mainlines and variants to peacefully co-exist rather than compete. Little is known on the motivations for creating variants in the social coding era, making it worthwhile to revisit the motivation for creating variant forks (*why?*).

Social coding platforms offer many facilities for code sharing (e.g., pull requests and cherry-picking). So if projects co-exist, one would expect variant forks to take advantage of this common ancestry, and frequently exchange interesting updates (e.g., patches) on the common artefacts. Despite advanced code-sharing facilities, Businge et al. observed very limited code integration, using the *git* and GitHub facilities, between the mainline and its variant projects [10]. This suggests that code sharing facilities in themselves are not enough for graceful co-evolution, making it worthwhile to investigate impediments for co-evolution (*how?*).

We therefore explore two research questions:

RQ1: Why do developers create and maintain variants on GitHub? The literature pre-dating *git* and social coding platforms identified four categories of motivations for creating variant forks: technical (e.g., diverging features), governance (e.g., diverging interests), legal (e.g., diverging licences), and personal (e.g., diverging principles). *RQ*1 aims to investigate whether those motivations for variant forks are still the same, or whether new factors have come into play.

RQ2: How do variant projects evolve with respect to the mainline? If, despite advanced code sharing facilities, there is limited code integration between the mainline and the variant projects, a possible cause could be related to how the teams working on the variants and the mainline are structured. Therefore, *RQ*2 investigates the overlap between the teams maintaining the mainline and variant forks, and how these teams interact. As such we hope to identify impediments for co-evolution.

The investigations are based on an online survey conducted with 105 maintainers involved in different active variant forks hosted on GitHub.

Our contributions are manifold: we identify new reasons for creating and maintaining variant forks; we identify and categorize different code reuse and change propagation practices between a variant and its mainline; we confirm that little code integration occurs between a variant and its mainline, and uncover concrete reasons for this phenomenon. We discuss

Variant forks - motivations and impediments.
Proceedings SANER 2022

PaReco: Patched Clones and Missed Patches among the Divergent Variants of a Software Family

Poedjadevie Kadjel Ramkisoen[1], John Businge[1,5], Brent van Bradel[1], Alexandre Decan[2], Serge Demeyer[1], Coen De Roover[3], and Foutse Khomh[4]
poedjadevie.ramkisoen@student.uantwerpen.be
(john.businge,Brent.vanBladel,serge.demeyer)@uantwerpen.be
alexandre.decan@umons.ac.be,Coen.De.Roover@vub.be,foutse.khomh@polymtl.ca
([1]Universiteit Antwerpen & Flanders Make, [2]F.R.S.-FNRS & University of Mons, [3]Vrije Universiteit Brussel), Belgium, [4]Polytechnique Montreal, Canada, [5]University of Nevada, Las Vegas, U.S.A.

ABSTRACT

Re-using whole repositories as a starting point for new projects is often done by maintaining a variant fork parallel to the original. However, the common artifacts between both are not always kept up to date. As a result, patches are not optimally integrated across the two repositories, which may lead to sub-optimal maintenance between the variant and the original project. A bug existing in both repositories can be patched in one but not the other (we see this as a missed opportunity) or it can be manually patched in both probably by different developers (we see this as effort duplication). In this paper we present a tool (named PaReco) which relies on clone detection to mine cases of missed opportunity and effort duplication from a pool of patches. We analyzed 364 (source→target) variant pairs with 8,323 patches resulting in a curated dataset containing 1,116 cases of effort duplication and 1,008 cases of missed opportunities. We achieve a precision of 91%, recall of 80%, accuracy of 88%, and F1-score of 85%. Furthermore, we investigated the time interval between patches and found out that, on average, missed patches in the target variants have been introduced in the source variants 52 weeks earlier. Consequently, PaReco can be used to manage variability in "time" by automatically identifying interesting patches in later project releases to be backported to supported earlier releases.

CCS CONCEPTS

• **Software and its engineering** → *Software version control*; *Software defect analysis*; **Software maintenance tools**; **Software configuration management and version control systems**.

KEYWORDS

Github, Clone&own, Variants, Software family, Forking, Social coding, Bug-fixes, Effort duplication, Clone detection

1 INTRODUCTION

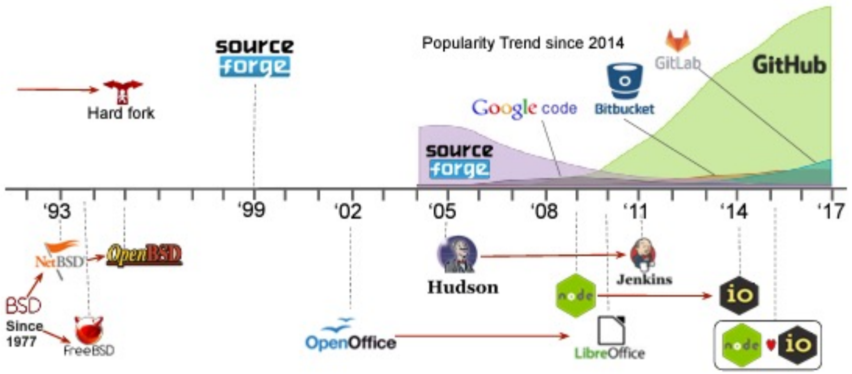
Code reuse is the practice of using existing code to speed up the development process. "Traditional" code reuse is performed by declaring a dependency towards another library or another package [21]. An alternative code reuse is the "clone&own" paradigm [9, 13, 14, 37, 51]. One would opt for the paradigm of "clone&own" over the "traditional" code reuse because the involved projects have traceability links and easily share new updates.

The "clone&own" paradigm is a commonly adopted approach for developing multi-variant software systems, where a new variant of a software system is created by copying and adapting an existing one and the two continue to evolve in parallel [9, 13, 14, 37, 51]. As a result, two or more software projects will share a common code base as well as independent, project-specific code. The multi-variant software systems are referred to as a *software family*, or *family* in short [13, 14]. With an increasing number of variants in the family, development becomes redundant and maintenance efforts rapidly grow [7, 23, 45, 54]. For example, if a bug is discovered and fixed in one variant, it is often unclear which other variants in the family are affected by the same bug and how this bug should be fixed in these variants. Although clone&own development paradigm has limitations, studies have reported their prevalence on social coding platforms like GitHub [9, 14].

This study aims to empirically quantify the extent to which *divergent variants* exhibit redundancy and missed essential updates concerning bug-fixes. Therefore, we present a tool (named PaReco) that can support the maintenance of divergent variants. PaReco mines bugfixes (patches) from a pool of updates in a source variant and relies on clone detection to classify the patches as interesting (i.e., redundant, missed) or uninteresting in the target variants. We present the illustration of the source / target variants in Fig. 1.

To the best of our knowledge, this is the first large-scale study on automatically identifying (and recommending) relevant bug fixes to developers of "clone&own" variants. Our contributions are threefold. (1) We analyzed 364 (source→target) variant pairs and validated the tool's output. This results in a curated dataset containing 1,116 cases of effort duplication and 1,008 cases of missed opportunities. The curated datasets can be accessed in our replication package [3]. (2) We quantify how many cases of effort duplication and missed opportunities exist between divergent variants. Next, we investigated the time interval between such patches to assess the window of opportunity for relevant bug fixes. (3) We developed PaReco which can be used as-is to support the management of variability in "space" (concurrent variations of the system at a single point in time). This can be achieved through mining interesting patches from one variant (source) and classify the patches as interesting or not interesting to the target variants. Existing tools in the GitHub marketplace notify projects about bug fixes, but are

Patched clones and missed patches among the divergent variants of a software family.
Proceedings ESEC/FSE 2022



November 2021

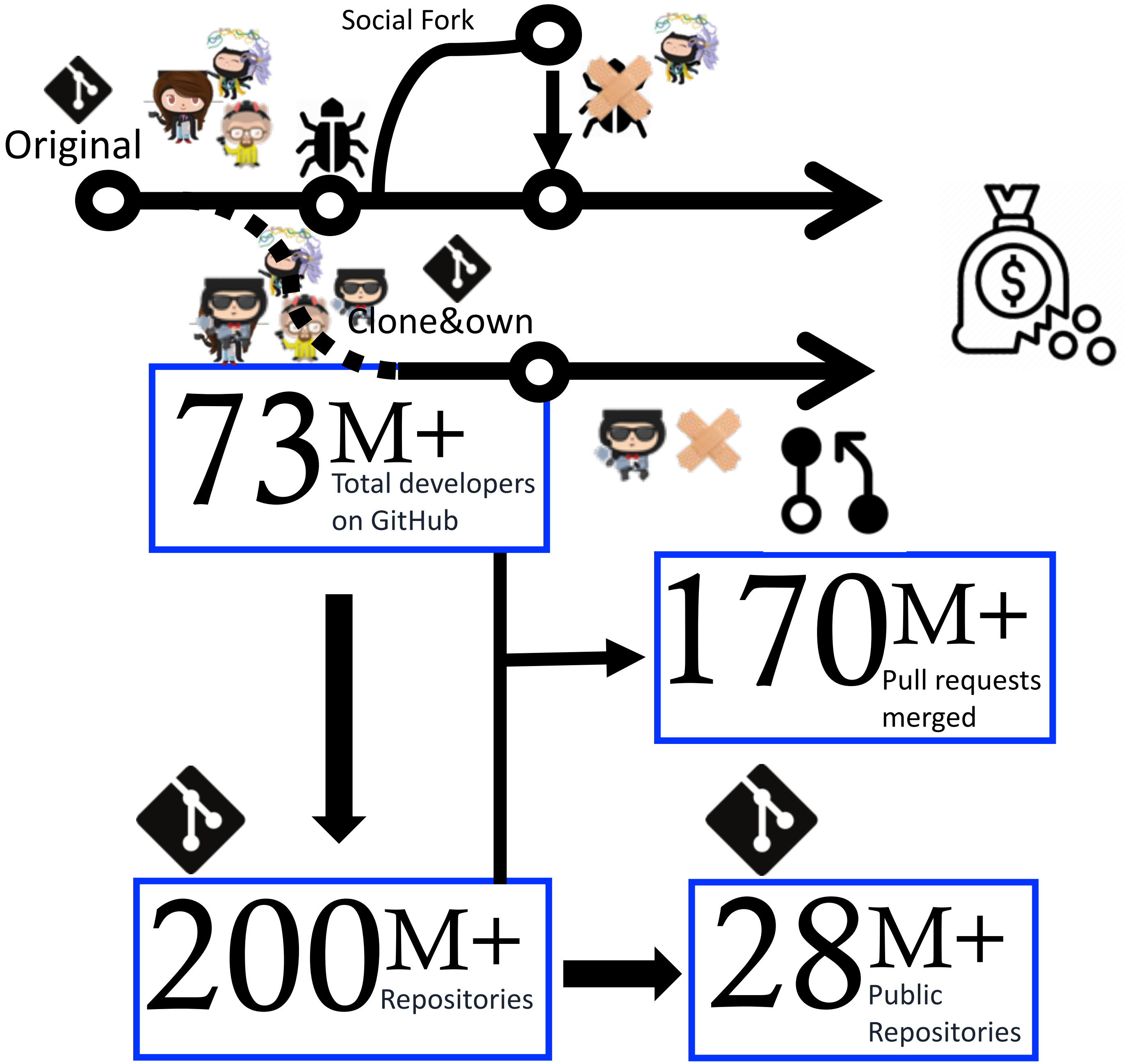
SECO-Assist

73M+ Total developers on GitHub

170M+ Pull requests merged

200M+ Repositories

28M+ Public Repositories


Social Fork
Original
Clone&own
73M+ Total developers on GitHub
170M+ Pull requests merged
200M+ Repositories
28M+ Public Repositories

SECO-Assist

$425M

The Equifax data breach occurred between **May and July 2017** at the American credit bureau Equifax. Private records of 147.9 million Americans along with 15.2 million British citizens and about 19,000 Canadian citizens were compromised in the breach, making it one of the largest cybercrimes related to identity theft.

Wired Magazine, “Equifax has no excuse”, September 2017

SECO-Assist

open source

March 2017

CVE-2017- 5638

EQUIFAX

DATA BREACH

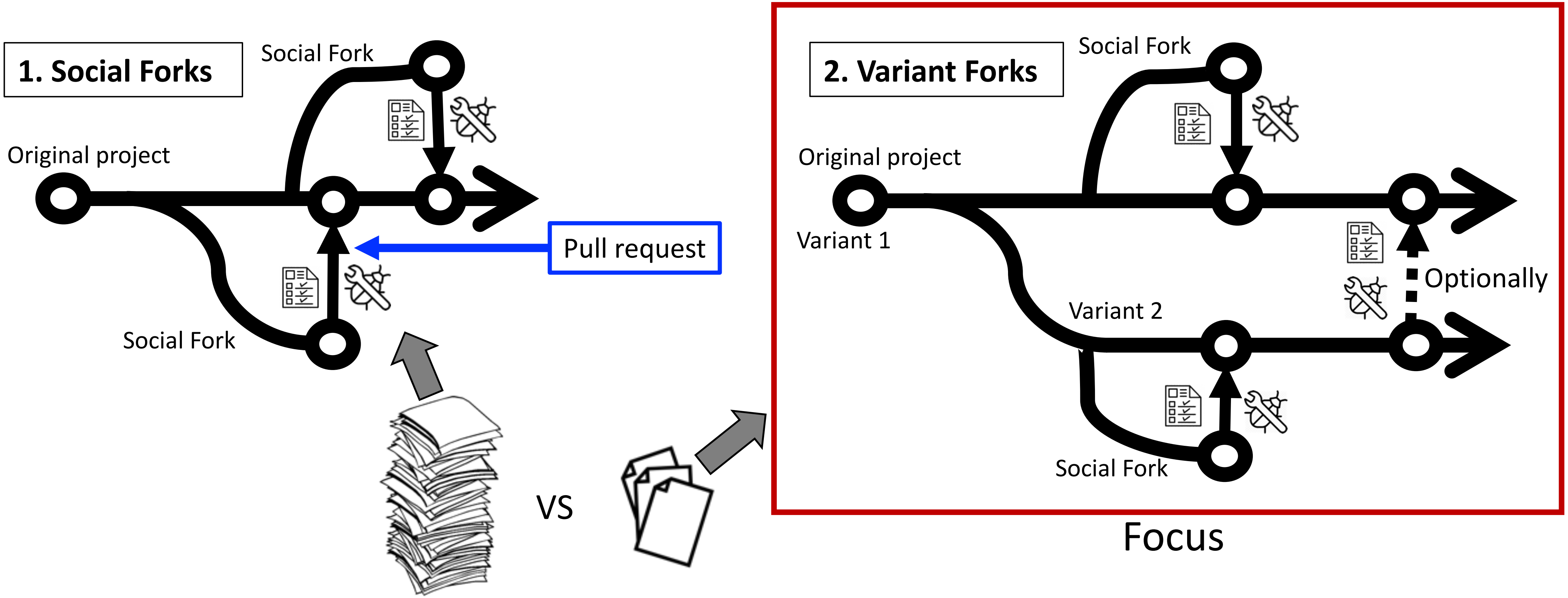
May 2017

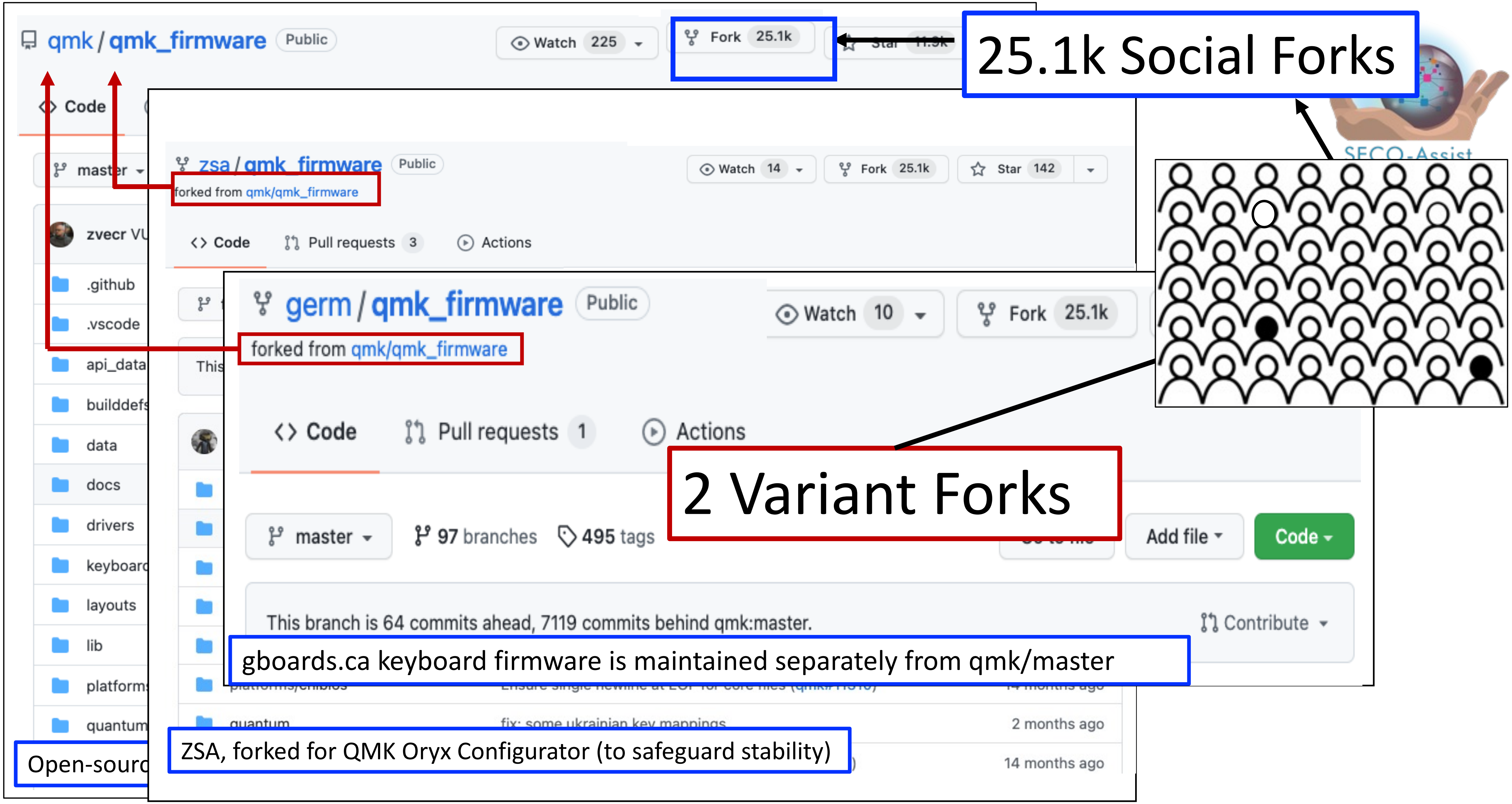
BIZ & IT —

Failure to patch two-month-old bug led to massive Equifax breach

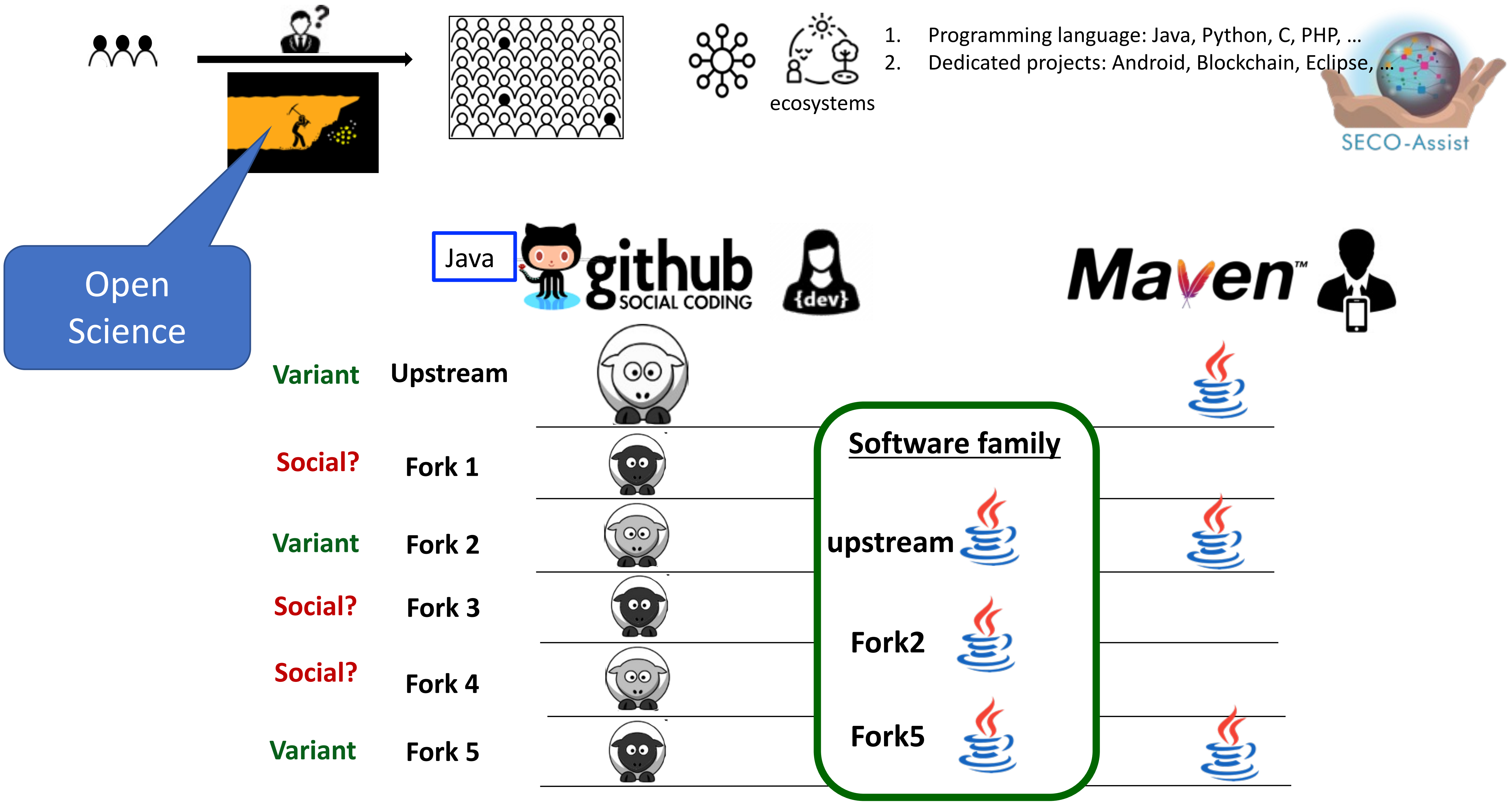
Critical Apache Struts bug was fixed in March. In May, it bit ~143 million US consumers.


apache / kafka Public
Watch 1.1k
Fork 11.3k
Star 21.5k
Pull requests 953 Open ✓ 11,016 Closed
ijuma KAFKA-13418: Support key updates with TLS 1.3 (#11966)
5aed178 12 hours ago 9,874 commits
R: Adding kafka-storage.bat file (similar to kafka-storage.sh) fo... 16 days ago
R: Fix class comparison in `AlterConfigPolicy.Req
config MINOR
connect KAFKA
core MINOR
docs KAFKA
examples
generator/src
gradle
jmh-benchmarks
licenses MINOR: Add missing licenses and update versions in LICENSE-binary... 7 months ago
Contributors 884
1.7k
+ 873 contributors
Languages
Java 74.2%
Scala 22.7%
Python 2.7%
Shell 0.2%
Roff 0.1%
Batchfile 0.1%
kafka
DOWNLOAD
3.1.0
Released January 24, 2022
3.0.0
Released September 21, 2021
2.8.0
Released April 19, 2021
2.7.0
Released Dec 21, 2020
2.6.0
Released Aug 3, 2020
Group: Apache Kafka
Sort: popular | newest
1. Apache Kafka
org.apache.kafka » kafka-clients
Apache Kafka
Last Release on Jan 21, 2022
1,747 usages
Apache
2. Apache Kafka
org.apache.kafka » kafka
Apache Kafka
Last Release on Jan 21, 2022
1,215 usages
Apache
3. Apache Kafka
org.apache.kafka » connect-api
Apache Kafka
Last Release on Jan 21, 2022
745 usages
Apache
4. Apache Kafka
org.apache.kafka » connect-transforms
Apache Kafka
Last Release on Jan 21, 2022
526 usages
Apache
5. Apache Kafka
org.apache.kafka » kafka-streams
Apache Kafka
Last Release on Jan 21, 2022
348 usages
Apache


1. Social Forks
Social Fork
Original project
Pull request
Social Fork
VS
2. Variant Forks
Social Fork
Original project
Variant 1
Optionally
Variant 2
Social Fork
Focus



1. https://thenounproject.com/icon/stand-out-938131/



Open Science
ecosystems
1. Programming language: Java, Python, C, PHP, …
2. Dedicated projects: Android, Blockchain, Eclipse, …
SECO-Assist
Java
github SOCIAL CODING
{dev}
Maven™
Variant Upstream
Social? Fork 1
Variant Fork 2
Social? Fork 3
Social? Fork 4
Variant Fork 5
Software family
upstream
Fork2
Fork5

Variant Forks – Motivations and Impediments

John Businge,* Ahmed Zerouali,‡ Alexandre Decan,† Tom Mens,† Serge Demeyer,* and Coen De Roover,‡
*University of Antwerp, Antwerp, Belgium
{ john.businge | serge.demeyer }@uantwerpen.be
†University of Mons, Mons, Belgium
{ alexandre.decan | tom.mens }@umons.ac.be
‡ Vrije Universiteit Brussels, Brussels, Belgium
{ ahmed.zerouali | coen.de.roover }@vub.be

***Abstract*—Social coding platforms centred around git provide explicit facilities to share code between projects: forks, pull requests, cherry-picking to name but a few. Variant forks are an interesting phenomenon in that respect, as they permit for different projects to peacefully co-exist, yet explicitly acknowledge the common ancestry. Several researchers analysed forking practices on open source platforms and observed that variant forks get created frequently. However, little is known on the motivations for launching such a variant fork. Is it mainly technical (e.g., diverging features), governance (e.g., diverging interests), legal (e.g., diverging licences), or do other factors come into play? We report the results of an exploratory qualitative analysis on the motivations behind creating and maintaining variant forks. We surveyed 105 maintainers of different active open source variant projects hosted on GitHub. Our study extends previous findings, identifying a number of fine-grained common motivations for launching a variant fork and listing concrete impediments for maintaining the co-existing projects.**

***Index Terms*—Mainlines, Variants, GitHub, Software ecosystems, Maintenance, Variability**

I. Introduction

The collaborative nature of open source software (OSS) development has led to the advent of social coding platforms centred around the git version control system, such as GitHub, BitBucket, and GitLab. These platforms bring the collaborative nature and code reuse of OSS development to another level, via facilities like forking, pull requests and cherry-picking. Developers may fork a *mainline repository* into a new *forked repository* and take governance over the latter while preserving the full revision history of the former. Before the advent of social coding platforms, forking was rare and was typically intended to compete with the original project [1]–[6].

With the rise of pull-based development [7], forking has become more common and the community typically characterises forks by their purpose [8]. *Social forks* are created for isolated development with the goal of contributing back to the mainline. In contract, *variant forks* are created by splitting off a new development branch to steer development into a new direction, while leveraging the code of the mainline project [9].

Several studies have investigated the motivations behind variant forks in the context of OSS projects [1]–[6]. However, most have been conducted before the rise of social coding platforms and it is known that GitHub has significantly changed the perception and practices of forking [8]. In this social coding era, variant projects often evolve out of social forks rather than being planned deliberately [8]. To this end, social coding platforms often enable mainlines and variants to peacefully co-exist rather than compete. Little is known on the motivations for creating variants in the social coding era, making it worthwhile to revisit the motivation for creating variant forks (*why?*).

Social coding platforms offer many facilities for code sharing (e.g., pull requests and cherry-picking). So if projects co-exist, one would expect variant forks to take advantage of this common ancestry, and frequently exchange interesting updates (e.g., patches) on the common artefacts. Despite advanced code-sharing facilities, Businge et al. observed very limited code integration, using the *git* and GitHub facilities, between the mainline and its variant projects [10]. This suggests that code sharing facilities in themselves are not enough for graceful co-evolution, making it worthwhile to investigate impediments for co-evolution (*how?*).

We therefore explore two research questions:

RQ1: Why do developers create and maintain variants on GitHub? The literature pre-dating *git* and social coding platforms identified four categories of motivations for creating variant forks: technical (e.g., diverging features), governance (e.g., diverging interests), legal (e.g., diverging licences), and personal (e.g., diverging principles). $RQ1$ aims to investigate whether those motivations for variant forks are still the same, or whether new factors have come into play.

RQ2: How do variant projects evolve with respect to the mainline? If, despite advanced code sharing facilities, there is limited code integration between the mainline and the variant projects, a possible cause could be related to how the teams working on the variants and the mainline are structured. Therefore, $RQ2$ investigates the overlap between the teams maintaining the mainline and variant forks, and how these teams interact. As such we hope to identify impediments for co-evolution.

The investigations are based on an online survey conducted with 105 maintainers involved in different active variant forks hosted on GitHub.

Our contributions are manifold: we identify new reasons for creating and maintaining variant forks; we identify and categorize different code reuse and change propagation practices between a variant and its mainline; we confirm that little code integration occurs between a variant and its mainline, and uncover concrete reasons for this phenomenon. We discuss

PaReco: Patched Clones and Missed Patches among the Divergent Variants of a Software Family

Poedjadevie Kadjel Ramkisoen[1], John Businge[1,5], Brent van Bradel[1], Alexandre Decan[2], Serge Demeyer[1], Coen De Roover[3], and Foutse Khomh[4]
poedjadevie.ramkisoen@student.uantwerpen.be
(john.businge,Brent.vanBladel,serge.demeyer)@uantwerpen.be
alexandre.decan@umons.ac.be,Coen.De.Roover@vub.be,foutse.khomh@polymtl.ca
([1]Universiteit Antwerpen & Flanders Make, [2]F.R.S.-FNRS & University of Mons, [3]Vrije Universiteit Brussel), Belgium, [4]Polytechnique Montreal, Canada, [5]University of Nevada, Las Vegas, U.S.A.

ABSTRACT

Re-using whole repositories as a starting point for new projects is often done by maintaining a variant fork parallel to the original. However, the common artifacts between both are not always kept up to date. As a result, patches are not optimally integrated across the two repositories, which may lead to sub-optimal maintenance between the variant and the original project. A bug existing in both repositories can be patched in one but not the other (we see this as a missed opportunity) or it can be manually patched in both probably by different developers (we see this as effort duplication). In this paper we present a tool (named PaReco) which relies on clone detection to mine cases of missed opportunity and effort duplication from a pool of patches. We analyzed 364 (source→target) variant pairs with 8,323 patches resulting in a curated dataset containing 1,116 cases of effort duplication and 1,008 cases of missed opportunities. We achieve a precision of 91%, recall of 80%, accuracy of 88%, and F1-score of 85%. Furthermore, we investigated the time interval between patches and found out that, on average, missed patches in the target variants have been introduced in the source variants 52 weeks earlier. Consequently, PaReco can be used to manage variability in "time" by automatically identifying interesting patches in later project releases to be backported to supported earlier releases.

CCS CONCEPTS

• **Software and its engineering** → *Software version control*; *Software defect analysis*; **Software maintenance tools**; **Software configuration management and version control systems**.

KEYWORDS

Github, Clone&own, Variants, Software family, Forking, Social coding, Bug-fixes, Effort duplication, Clone detection

1 INTRODUCTION

Code reuse is the practice of using existing code to speed up the development process. "Traditional" code reuse is performed by declaring a dependency towards another library or another package [21]. An alternative code reuse is the "clone&own" paradigm [9, 13, 14, 37, 51]. One would opt for the paradigm of "clone&own" over the "traditional" code reuse because the involved projects have traceability links and easily share new updates.

The "clone&own" paradigm is a commonly adopted approach for developing multi-variant software systems, where a new variant of a software system is created by copying and adapting an existing one and the two continue to evolve in parallel [9, 13, 14, 37, 51]. As a result, two or more software projects will share a common code base as well as independent, project-specific code. The multi-variant software systems are referred to as a *software family*, or *family* in short [13, 14]. With an increasing number of variants in the family, development becomes redundant and maintenance efforts rapidly grow [7, 23, 45, 54]. For example, if a bug is discovered and fixed in one variant, it is often unclear which other variants in the family are affected by the same bug and how this bug should be fixed in these variants. Although clone&own development paradigm has limitations, studies have reported their prevalence on social coding platforms like GitHub [9, 14].

This study aims to empirically quantify the extent to which *divergent variants* exhibit redundancy and missed essential updates concerning bug-fixes. Therefore, we present a tool (named PaReco) that can support the maintenance of divergent variants. PaReco mines bugfixes (patches) from a pool of updates in a source variant and relies on clone detection to classify the patches as interesting (i.e., redundant, missed) or uninteresting in the target variants. We present the illustration of the source / target variants in Fig. 1.

To the best of our knowledge, this is the first large-scale study on automatically identifying (and recommending) relevant bug fixes to developers of "clone&own" variants. Our contributions are three-fold. (1) We analyzed 364 (source→target) variant pairs and validated the tool's output. This results in a curated dataset containing 1,116 cases of effort duplication and 1,008 cases of missed opportunities. The curated datasets can be accessed in our replication package [3]. (2) We quantify how many cases of effort duplication and missed opportunities exist between divergent variants. Next, we investigated the time interval between such patches to assess the window of opportunity for relevant bug fixes. (3) We developed PaReco which can be used as-is to support the management of variability in "space" (concurrent variations of the system at a single point in time). This can be achieved through mining interesting patches from one variant (source) and classify the patches as interesting or not interesting to the target variants. Existing tools in the GitHub marketplace notify projects about bug fixes, but are

**Variant forks - motivations and impediments.
Proceedings SANER 2022**

Patched clones and missed patches among the divergent variants of a software family.
Proceedings ESEC/FSE 2022

Why do developers create and maintain variants on GitHub?

SECO-Assist

A Comprehensive Study of Software Forks: Dates, Reasons and Outcomes

Gregorio Robles an

GSyC/Libresoft, Uni

Perspectives on Code Forking and Sustainability in Open Source Software

dman[1], and Martin Fougère[1]
Helsinki, Finland
nken.fi
y, Tampere, Finland

Code Forking, Governance, and Sustainability in Open Source Software

Linus Nyman and Juho Lindman

Forks impacts and motivations in free and open source projects

R. Viseur

All of these studies and many others were conducted in the pre-GitHub days

and is typically found in the free and open source software field. As a failure of cooperation in a context of open innovation, forking is a practical and informative subject of study. In-depth researches concerning the fork phenomenon are uncommon. We therefore conducted a detailed study of 26 forks from popular free and open source projects. We created fact sheets, highlighting the impact and motivations to fork. We particularly point to the fact that the desire for greater technical

prevent forks.

II. BACKGROUND

A. *Perception of fork*

If the fear of forks is visible with companies, Gosain also points to the sensitivity of the open source community beside the forks and the fragmentation of projects [10].

Clone-Based Variability Management in the Android Ecosystem

John Businge, Moses Openja, Sarah Nadi, Engineer Bainomugisha, and Thorsten Berger
Mbarara University of Science and Technology, Mbarara, Uganda
Makerere University, Kampala, Uganda
University of Alberta, Edmonton, Canada

How Has Forking Changed in the Last 20 Years? A Study of Hard Forks on GitHub

Current GitHub days only two studies

ABSTRACT

The notion of f… sion control sy… Traditionally f… opment branch… conducted mostly in pre-GitHub days showed that hard forks were often seen critical as they may fragment a community. Today, in social coding environments, open-source developers are encouraged to fork a project in order to contribute to the community (which we call *social forks*), which may have also influenced perceptions and practices around hard forks. To revisit hard forks, we identify, study, and classify 15,306 hard forks on GitHub and interview 18 owners of hard forks or forked repositories. We find that, among others, hard forks often evolve out of social forks rather than being planned deliberately and that perception about hard forks have

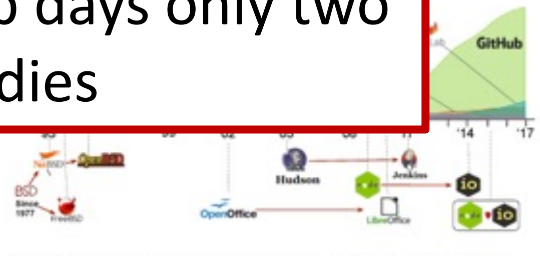


Figure 1: Timeline of some popular open-source forking events; popularity approximated with *Google Trends*.

Little is known about the motivations of creating variants on social coding platforms.

Survey

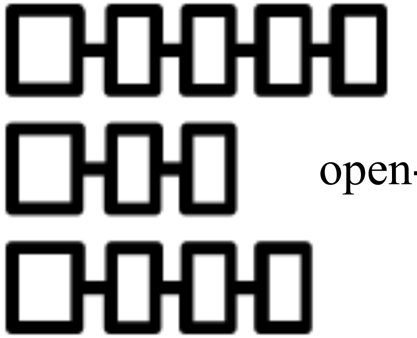
105 fork variant developers

GitHub

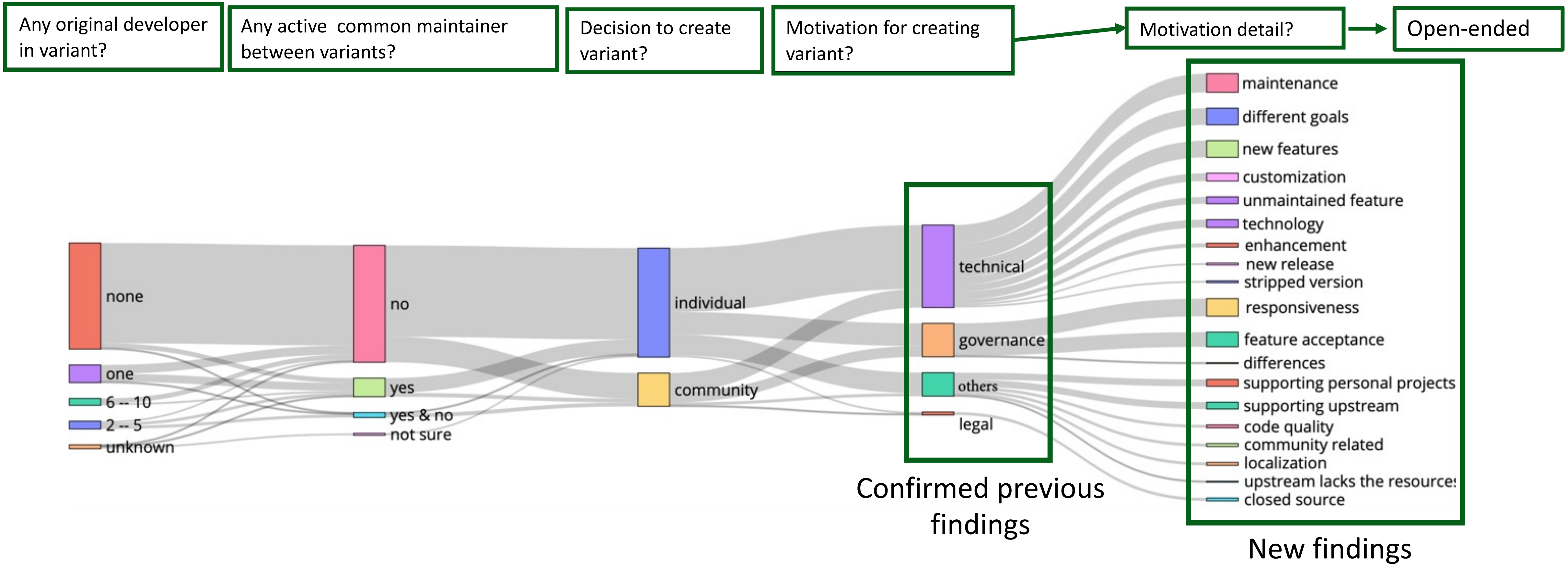
105 repositories

We designed a 12-question survey that included both closed and open-ended questions.

card-sorting: from text to themes



open-ended questions



Any original developer in variant?
Any active common maintainer between variants?
Decision to create variant?
Motivation for creating variant?
Motivation detail?
Open-ended
none
one
6 -- 10
2 -- 5
unknown
no
yes
yes & no
not sure
individual
community
technical
governance
others
legal
maintenance
different goals
new features
customization
unmaintained feature
technology
enhancement
new release
stripped version
responsiveness
feature acceptance
differences
supporting personal projects
supporting upstream
code quality
community related
localization
upstream lacks the resources
closed source
Confirmed previous findings
New findings

Why do developers create and maintain variants on GitHub?

Previous studies identified four categories of motivations for creating variant forks:

- technical (e.g., diverging features),
- governance (e.g., diverging interests),
- legal (e.g., diverging licenses), and
- personal (e.g., diverging principles).

do they still hold?

[R59]. **Motivation** – governance.``*The PR to merge the fork's new capabilities into the mainline code was too large, [...] and my attempts to incorporate feedback into the PR [...] ended upsetting the primary maintainer who has been studiously ignoring the pull request for three years.* ☹
Motivation-detail – responsiveness.

{dev}

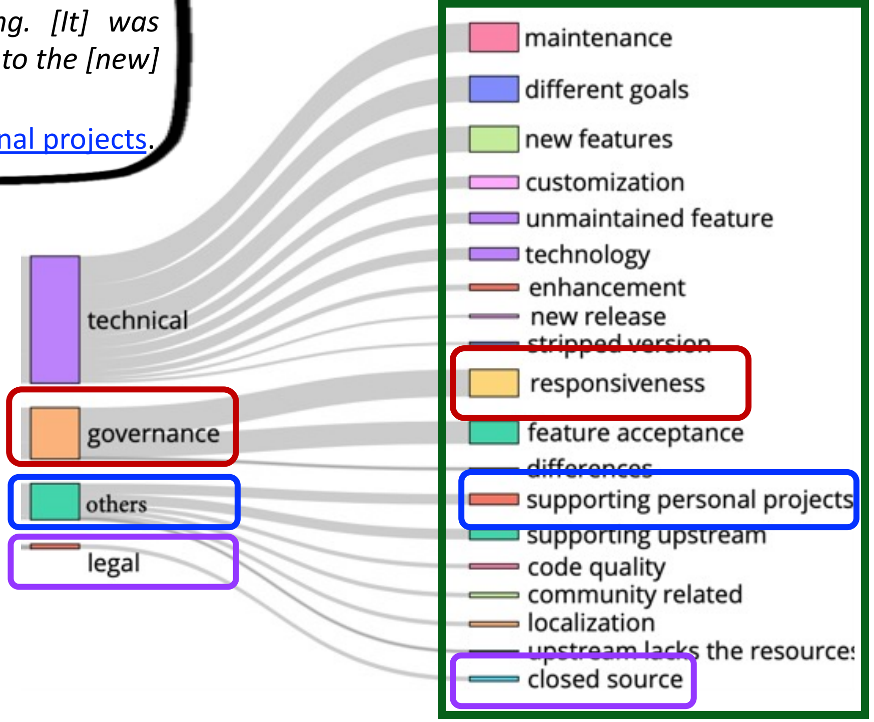
[R18]. motivation – others. "*[The] maintainer was not interested in a PR that added functionality needed by a project I'm developing. [It] was considerably easier to add the logic into the [new] library than bolt it on*".
Motivation-detail – supporting personal projects.

{dev}

[R36]. **Motivation** – legal. "*The founders of the mainline had been absent from the project for several years, but came back and booted the maintainers off and [...] shifted the project to a closed source.*"
Motivation detail – closed source.

{dev}

Motivation detail? → Open-ended

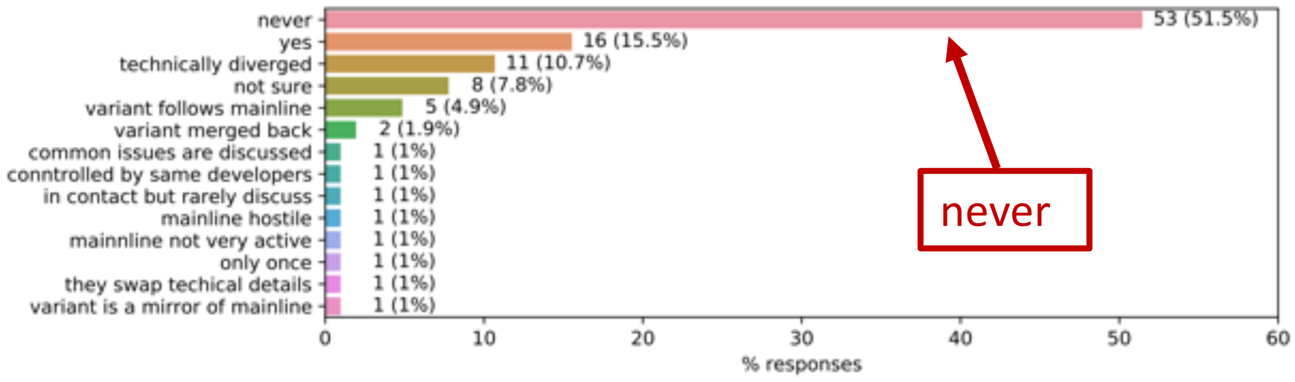


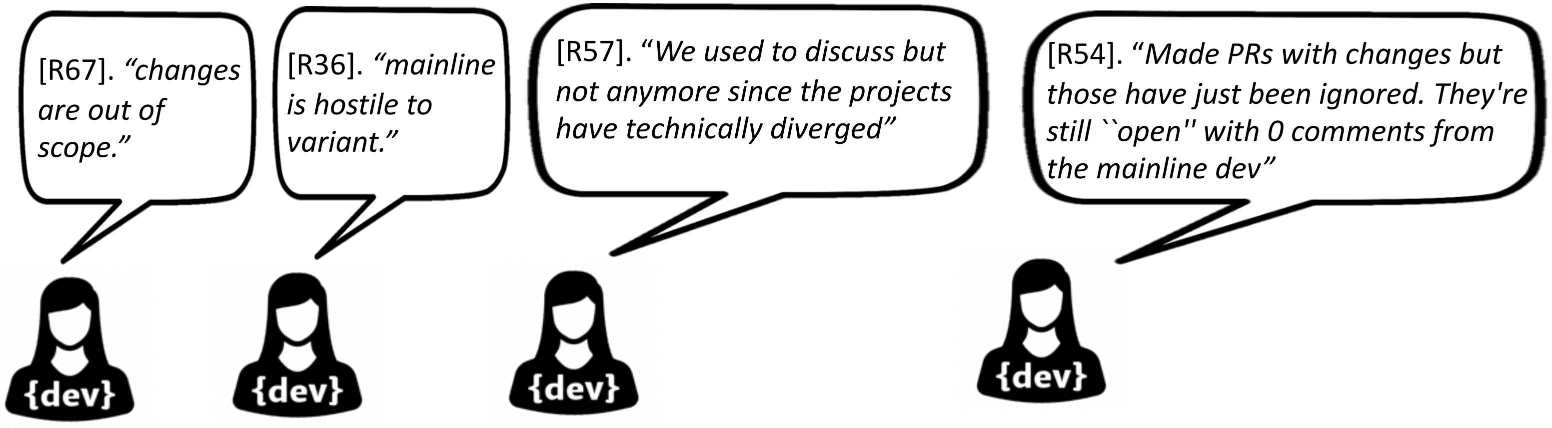
New findings

100 of the 105 variant developers answered the optional open-ended question

How do variant projects evolve with respect to the mainline?

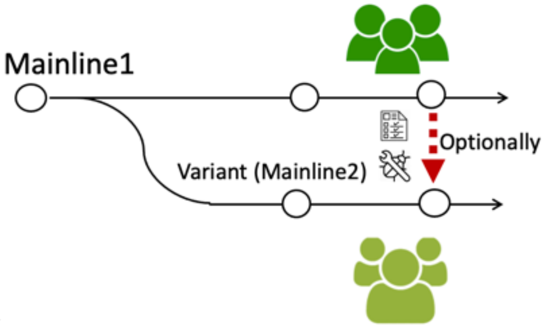
SQ. Do the variant forks and the original project still discuss the main directions of the project?





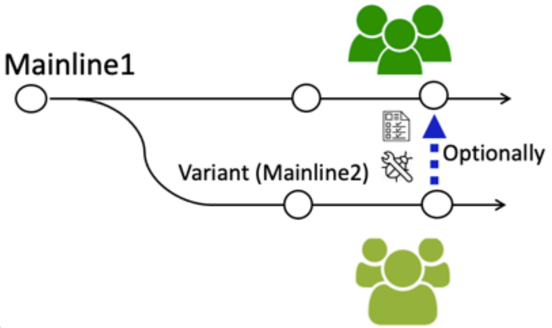
How do variant projects evolve with respect to the mainline?

SQ. How often do the maintainers of the variant integrate the following types of changes to and from the mainline?

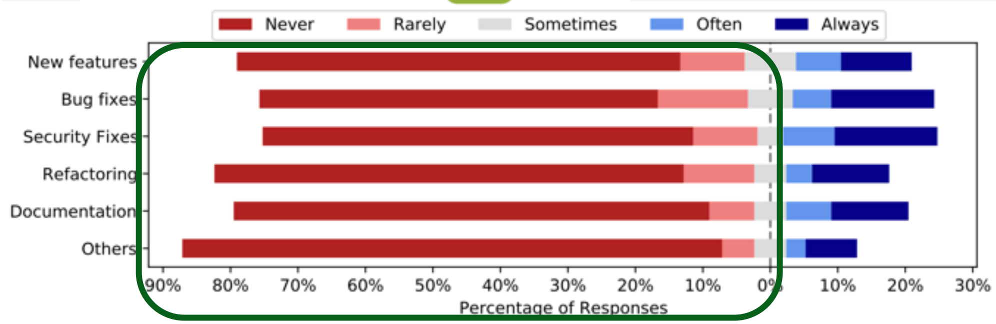


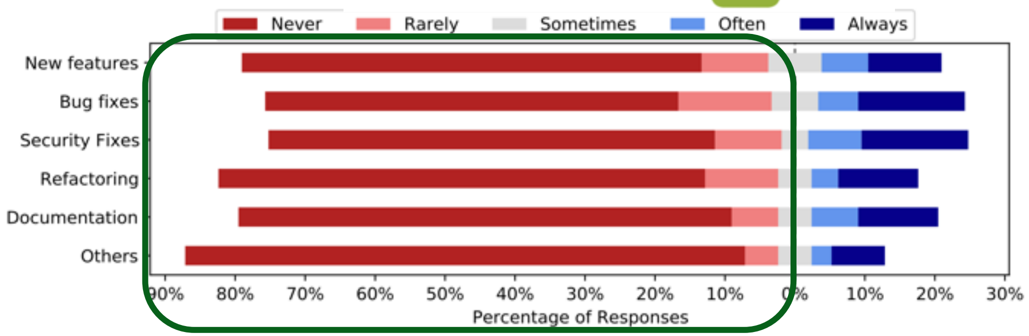
From

- New features
- Bug fixes
- Security fixes
- Refactorings
- Documentation
- Others



To





Reasons for lack of interaction (Impediments):

i. technical divergence
ii. governance disputes
iii. diverging licenses,
iv. distinct development teams

Variant Forks – Motivations and Impediments

John Businge,* Ahmed Zerouali,‡ Alexandre Decan,† Tom Mens,† Serge Demeyer,* and Coen De Roover,‡
*University of Antwerp, Antwerp, Belgium
{ john.businge | serge.demeyer }@uantwerpen.be
†University of Mons, Mons, Belgium
{ alexandre.decan | tom.mens }@umons.ac.be
‡ Vrije Universiteit Brussels, Brussels, Belgium
{ ahmed.zerouali | coen.de.roover }@vub.be

PaReco: Patched Clones and Missed Patches among the Divergent Variants of a Software Family

Poedjadevie Kadjel Ramkisoen[1], John Businge[1,5], Brent van Bradel[1], Alexandre Decan[2], Serge Demeyer[1], Coen De Roover[3], and Foutse Khomh[4]

poedjadevie.ramkisoen@student.uantwerpen.be

(john.businge,Brent.vanBladel,serge.demeyer)@uantwerpen.be

alexandre.decan@umons.ac.be,Coen.De.Roover@vub.be,foutse.khomh@polymtl.ca

([1]Universiteit Antwerpen & Flanders Make, [2]F.R.S.-FNRS & University of Mons, [3]Vrije Universiteit Brussel), Belgium, [4]Polytechnique Montreal, Canada, [5]University of Nevada, Las Vegas, U.S.A.

ABSTRACT

Re-using whole repositories as a starting point for new projects is often done by maintaining a variant fork parallel to the original. However, the common artifacts between both are not always kept up to date. As a result, patches are not optimally integrated across the two repositories, which may lead to sub-optimal maintenance between the variant and the original project. A bug existing in both repositories can be patched in one but not the other (we see this as a missed opportunity) or it can be manually patched in both probably by different developers (we see this as effort duplication). In this paper we present a tool (named PaReco) which relies on clone detection to mine cases of missed opportunity and effort duplication from a pool of patches. We analyzed 364 (source→target) variant pairs with 8,323 patches resulting in a curated dataset containing 1,116 cases of effort duplication and 1,008 cases of missed opportunities. We achieve a precision of 91%, recall of 80%, accuracy of 88%, and F1-score of 85%. Furthermore, we investigated the time interval between patches and found out that, on average, missed patches in the target variants have been introduced in the source variants 52 weeks earlier. Consequently, PaReco can be used to manage variability in "time" by automatically identifying interesting patches in later project releases to be backported to supported earlier releases.

CCS CONCEPTS

• **Software and its engineering** → *Software version control*; *Software defect analysis*; **Software maintenance tools**; **Software configuration management and version control systems**.

KEYWORDS

Github, Clone&own, Variants, Software family, Forking, Social coding, Bug-fixes, Effort duplication, Clone detection

1 INTRODUCTION

Code reuse is the practice of using existing code to speed up the development process. "Traditional" code reuse is performed by declaring a dependency towards another library or another package [21]. An alternative code reuse is the "clone&own" paradigm [9, 13, 14, 37, 51]. One would opt for the paradigm of "clone&own" over the "traditional" code reuse because the involved projects have traceability links and easily share new updates.

The "clone&own" paradigm is a commonly adopted approach for developing multi-variant software systems, where a new variant of a software system is created by copying and adapting an existing one and the two continue to evolve in parallel [9, 13, 14, 37, 51]. As a result, two or more software projects will share a common code base as well as independent, project-specific code. The multi-variant software systems are referred to as a *software family*, or *family* in short [13, 14]. With an increasing number of variants in the family, development becomes redundant and maintenance efforts rapidly grow [7, 23, 45, 54]. For example, if a bug is discovered and fixed in one variant, it is often unclear which other variants in the family are affected by the same bug and how this bug should be fixed in these variants. Although clone&own development paradigm has limitations, studies have reported their prevalence on social coding platforms like GitHub [9, 14].

This study aims to empirically quantify the extent to which *divergent variants* exhibit redundancy and missed essential updates concerning bug-fixes. Therefore, we present a tool (named PaReco) that can support the maintenance of divergent variants. PaReco mines bugfixes (patches) from a pool of updates in a source variant and relies on clone detection to classify the patches as interesting (i.e., redundant, missed) or uninteresting in the target variants. We present the illustration of the source / target variants in Fig. 1.

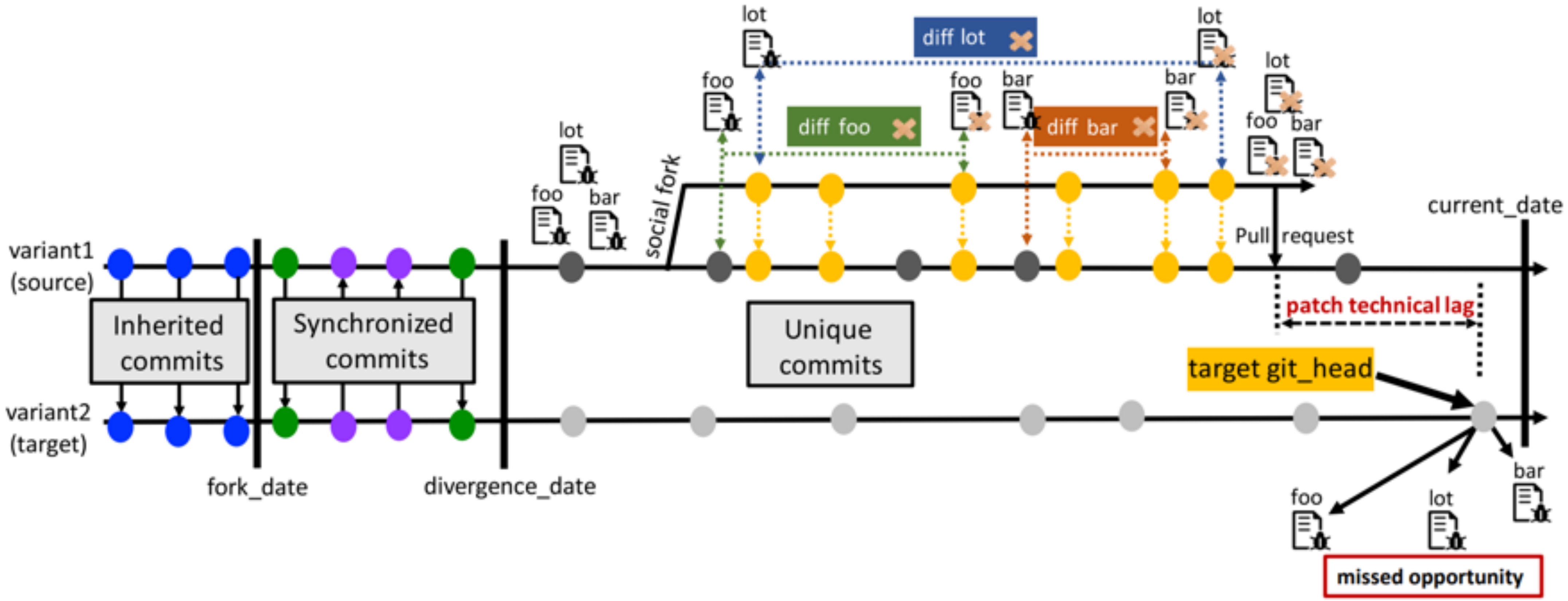
To the best of our knowledge, this is the first large-scale study on automatically identifying (and recommending) relevant bug fixes to developers of "clone&own" variants. Our contributions are threefold. (1) We analyzed 364 (source→target) variant pairs and validated the tool's output. This results in a curated dataset containing 1,116 cases of effort duplication and 1,008 cases of missed opportunities. The curated datasets can be accessed in our replication package [3]. (2) We quantify how many cases of effort duplication and missed opportunities exist between divergent variants. Next, we investigated the time interval between such patches to assess the window of opportunity for relevant bug fixes. (3) We developed PaReco which can be used as-is to support the management of variability in "space" (concurrent variations of the system at a single point in time). This can be achieved through mining interesting patches from one variant (source) and classify the patches as interesting or not interesting to the target variants. Existing tools in the GitHub marketplace notify projects about bug fixes, but are

Variant forks - motivations and impediments. Proceedings SANER 2022

Patched clones and missed patches among the divergent variants of a software family. Proceedings ESEC/FSE 2022

SECO-Assist

Problem


lot
diff lot
lot
foo
diff foo
foo
bar
diff bar
bar
lot
foo
bar
lot
foo
bar
social fork
current_date
Pull request
variant1
(source)
Inherited
commits
Synchronized
commits
Unique
commits
patch technical lag
target git_head
variant2
(target)
fork_date
divergence_date
bar
foo
lot
missed opportunity

Concrete Example: Missed Opportunity

SECO-Assist

Buggy code from upstream

```
                return;
            }
        } while (p < (uint16_t *)SYMVAL(__eeprom_workarea_end__));
        flashend = (uint32_t)((uint16_t *)SYMVAL(__eeprom_workarea_end__) - 1);
    }
```

Buggy line

Patched code from upstream

```
                return;
            }
        } while (p < (uint16_t *)SYMVAL(__eeprom_workarea_end__));
        flashend = (uint32_t)(p - 1);
    }
```

Patched line

Diff for patch in upstream

```
@@ -363,7 +363,7 @@

        } while (p < (uint16_t *)SYMVAL(__eeprom_workarea_end__));
-       flashend = (uint32_t)((uint16_t *)SYMVAL(__eeprom_workarea_end__) - 1);
+       flashend = (uint32_t)(p - 1);
```

Hunk

File from divergent fork at `git_head`

```
                return;
            }
        } while (p < (uint16_t *)SYMVAL(__eeprom_workarea_end__));
        flashend = (uint32_t)((uint16_t *)SYMVAL(__eeprom_workarea_end__) - 1);
    }
```

Buggy line

Concrete Example: Effort Duplication

SECO-Assist

Buggy code from upstream

```
# http://ss64.com/nt/syntax-esc.html
_escape_re = re.compile(r'(?<!\^)[&<>]|(?<!\^)\^(?![&<>\^])')
_escaper = partial(_escape_re.sub, lambda m: '^' + m.group(0))
```

Buggy line

Patched code from upstream

```
# http://ss64.com/nt/syntax-esc.html
_escape_re = re.compile(r'(?<!\^)[&<>]|(?<!\^)\^(?![&<>\^])|(\|)')
_escaper = partial(_escape_re.sub, lambda m: '^' + m.group(0))
```

Patched line

Diff for patch in upstream

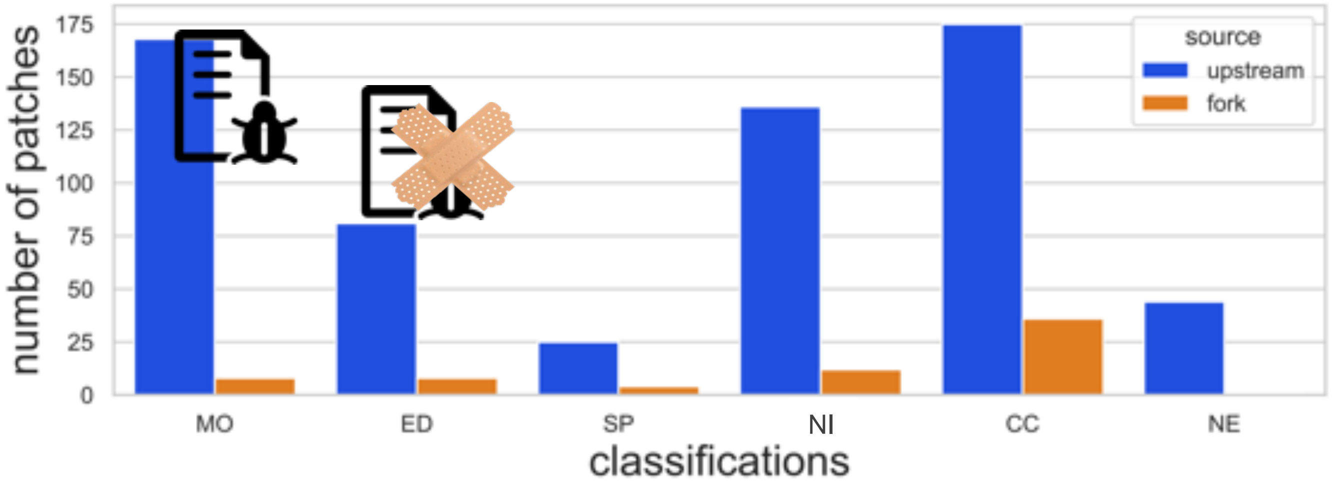
```
@@ -24,7 +24,7 @@

  # http://ss64.com/nt/syntax-esc.html
- _escape_re = re.compile(r'(?<!\^)[&<>]|(?<!\^)\^(?![&<>\^])')
+ _escape_re = re.compile(r'(?<!\^)[&<>]|(?<!\^)\^(?![&<>\^])|(\|)')
  _escaper = partial(_escape_re.sub, lambda m: '^' + m.group(0))
```

Hunk

File from divergent fork at `git_head`

```
# http://ss64.com/nt/syntax-esc.html
_escape_re = re.compile(r'(?<!\^)[&<>]|(?<!\^)\^(?![&<>\^])|(\|)')
_escaper = partial(_escape_re.sub, lambda m: '^' + m.group(0))
```

Patched line

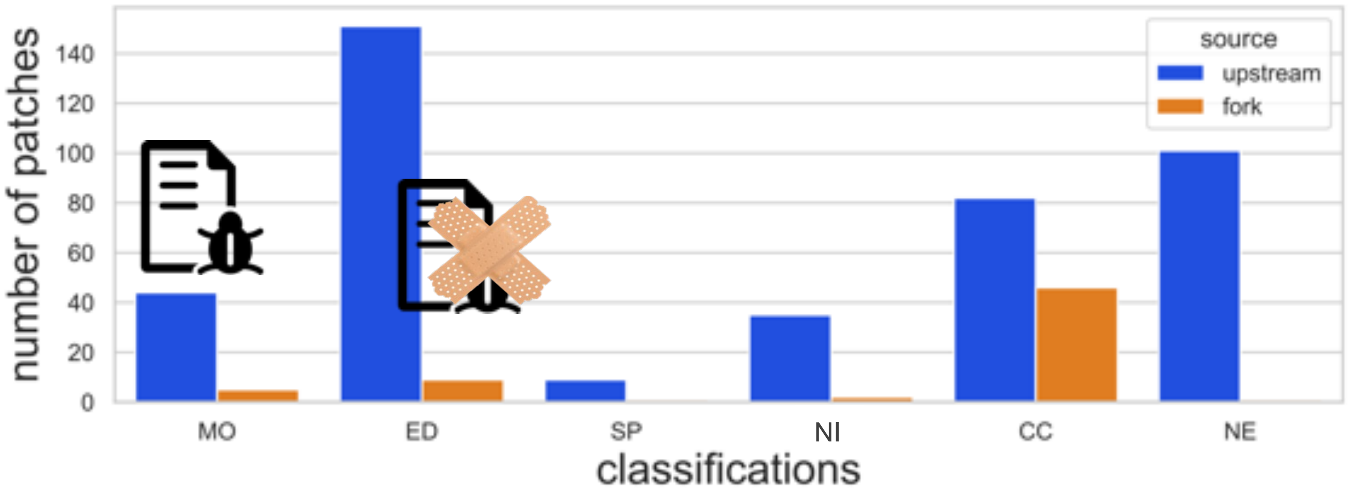


apache/kafka (upstream) - linkedin/kafka (fork)

MO – Missed opportunity
ED – Effort duplication
SP – Both buggy and patched lines
NI – Uninteresting
CC – Unhandled programming language
NE – Missing file in target
EE – Error

apache/kafka (upstream)

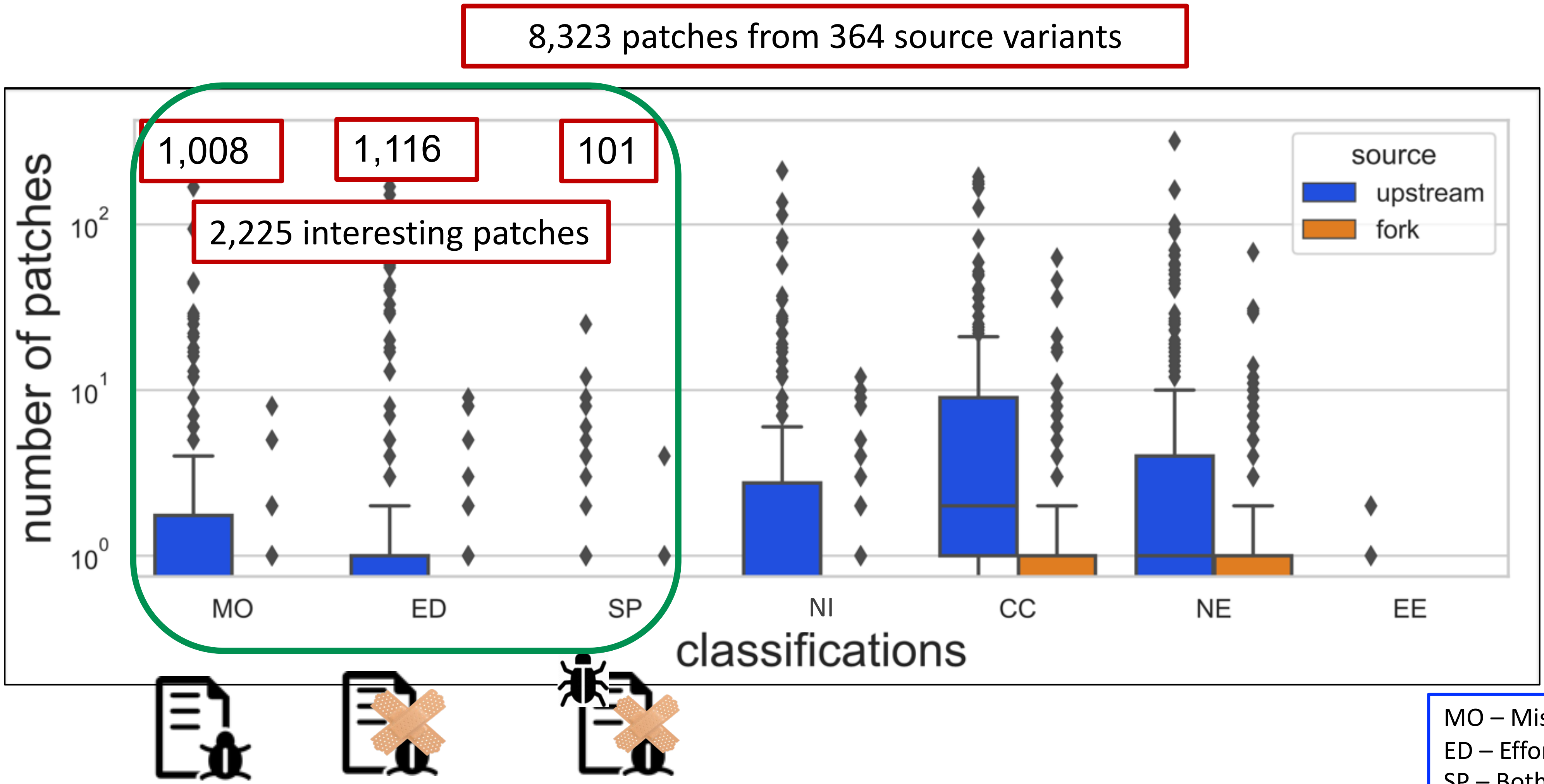
linkedin/kafka (fork)



microsoft/azure-tools-for-java (upstream) - JetBrains/azure-tools-for-intellij (fork)

microsoft/azure-tools-for-java (upstream)

JetBrains/azure-tools-for-intellij (fork)



Precision	Recall	Accuracy	F1-Score
91.0%	80.2%	88.0%	85.3%

MO – Missed opportunity
ED – Effort duplication
SP – Both buggy and patched lines
NI – Not interesting
CC – Unhandled programming language
NE – Missing file in target
EE – Error

Reengineering Project 2021—2022

LinkedIn is a clone-and-own variant of Apache Kafka that was created by copying and adapting the existing code of Apache Kafka.

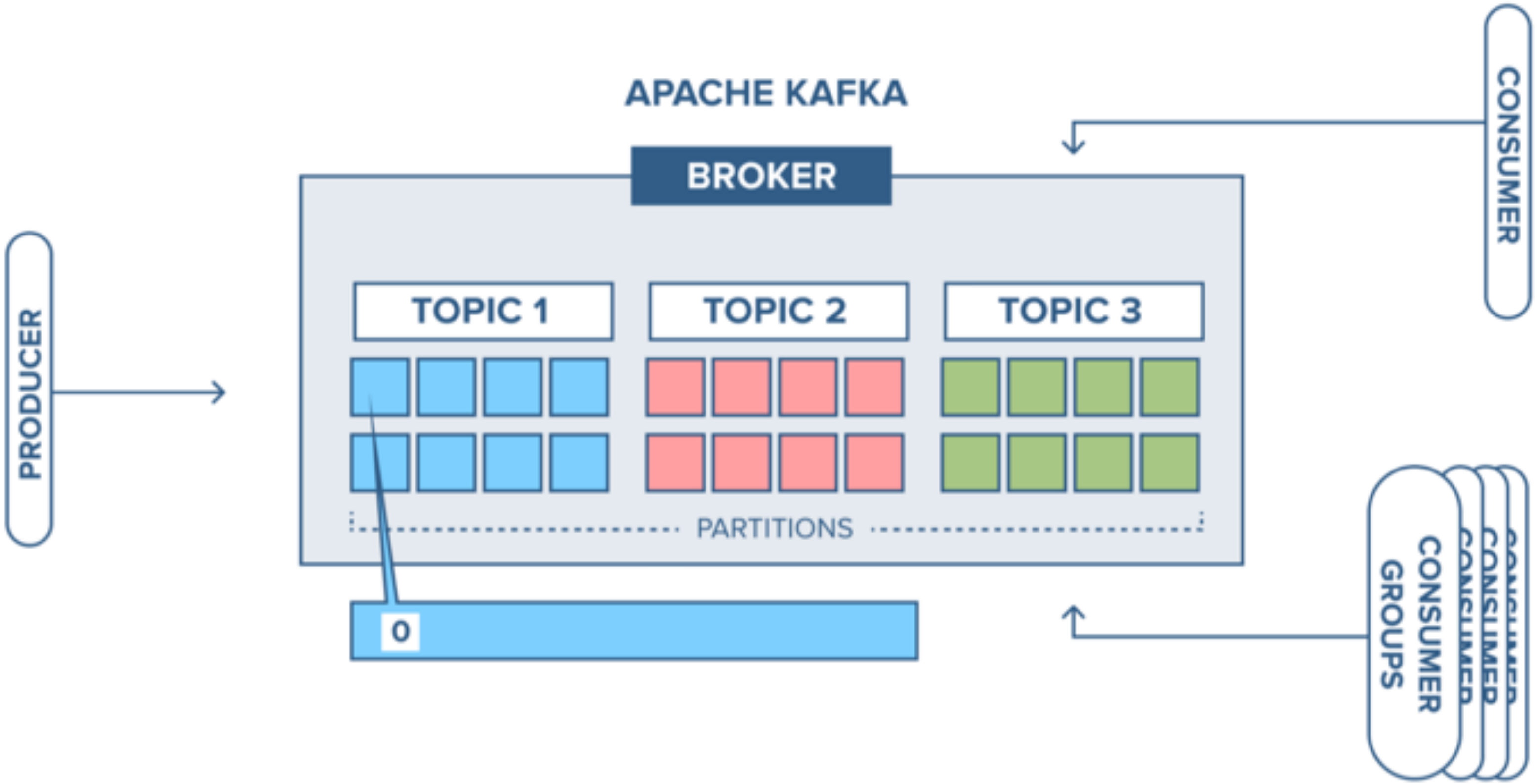
…

LinkedIn has 500 individual commits, and Apache Kafka has 3,103 individual commits.

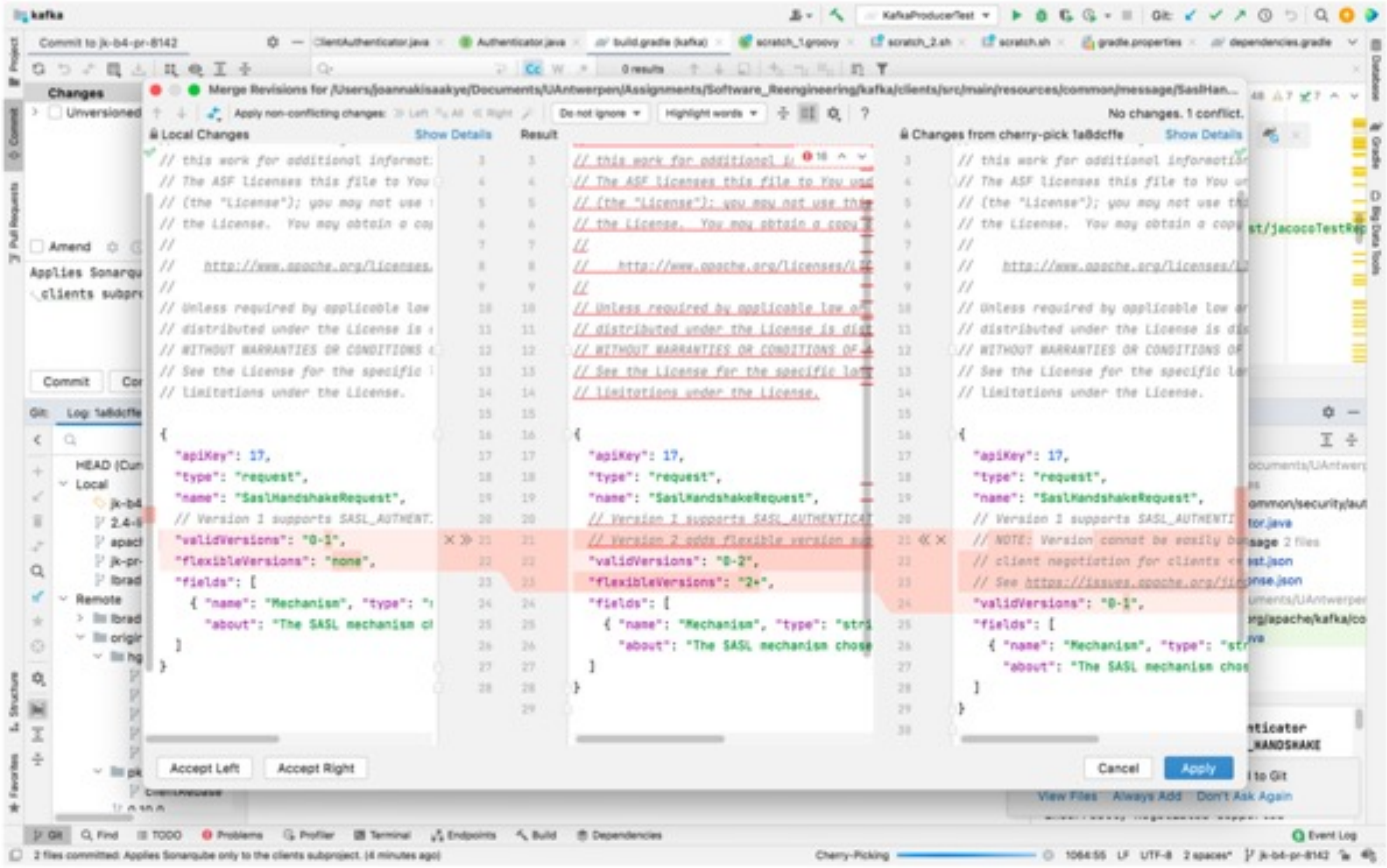
…

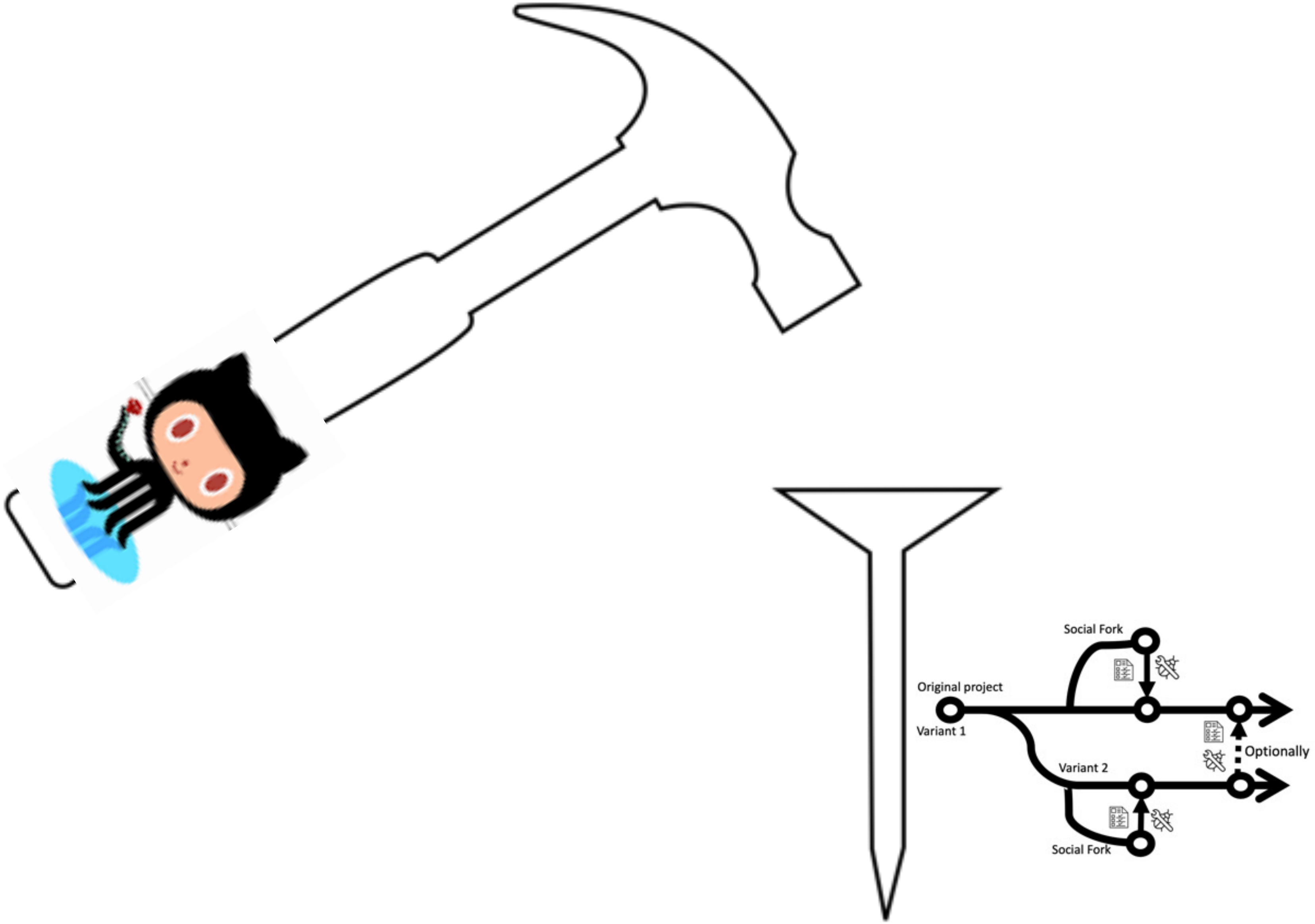
Your assignment is to identify numerous patches from patches.xls that are of different sizes and integrate them in the source variant LinkedIn. The size can be measured in terms of number of commits, files_changed, added_lines, deleted_lines, modules.

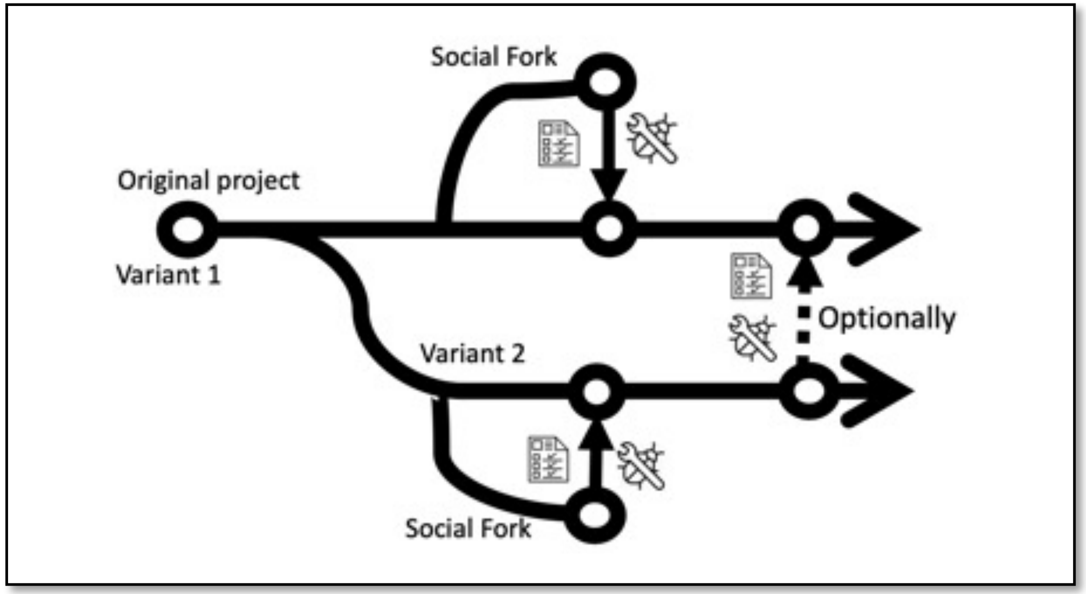
Apache Kafka

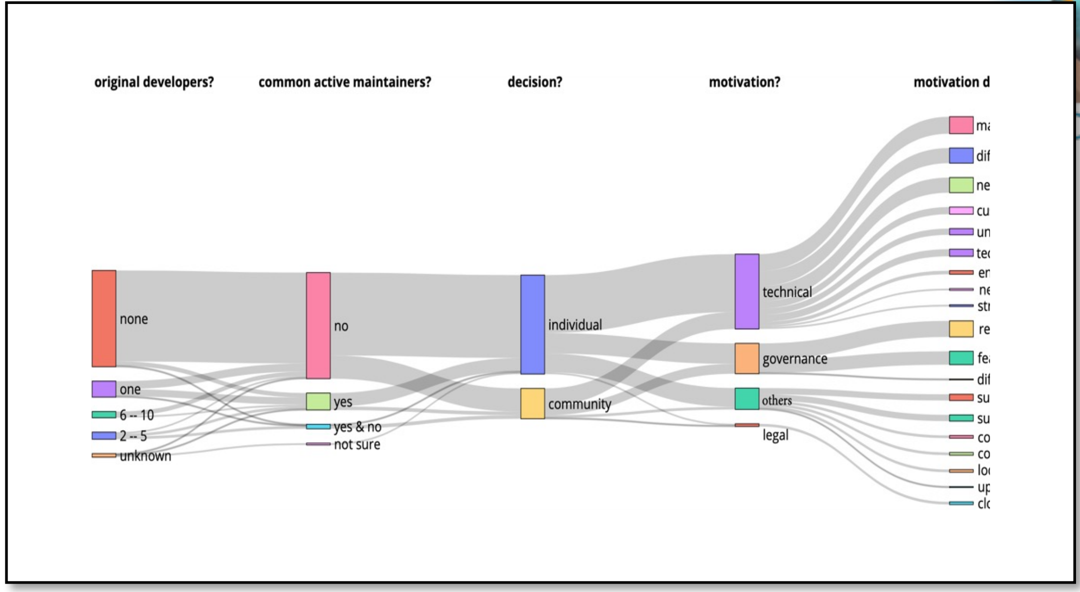


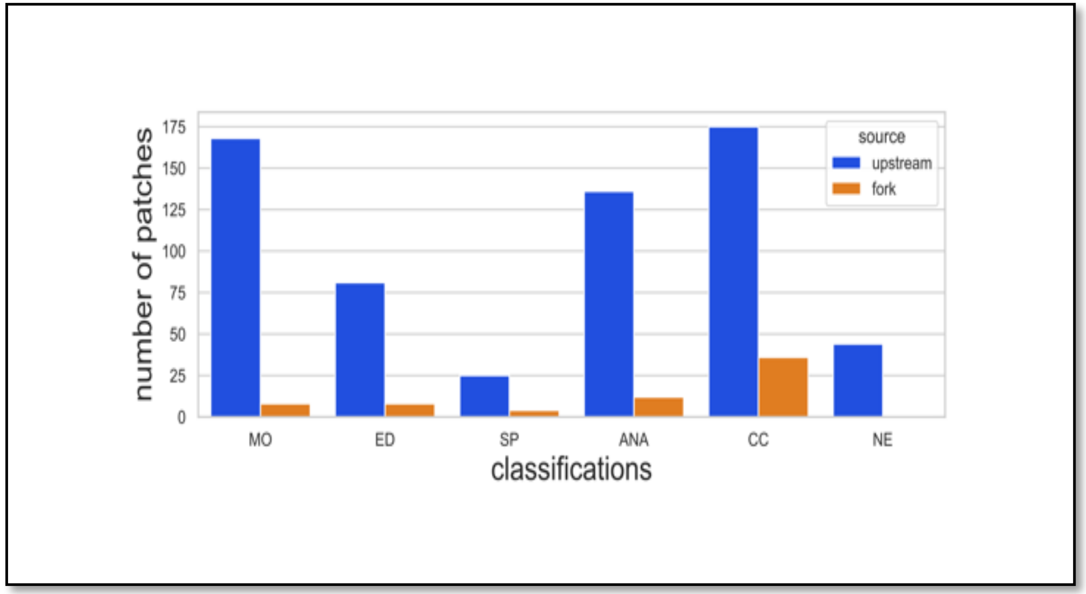
Cherry Picking – Merge Conflicts




Social Fork
Original project
Variant 1
Optionally
Variant 2
Social Fork


Social Fork
Original project
Variant 1
Optionally
Variant 2
Social Fork


original developers?
common active maintainers?
decision?
motivation?
none
one
6 -- 10
2 -- 5
unknown
no
yes
yes & no
not sure
individual
community
technical
governance
others
legal


number of patches
175
150
125
100
75
50
25
0
MO
ED
SP
ANA
CC
NE
classifications
source
upstream
fork