Is It Possible to Automatically Port Kernel Modules ?

Yanjie Zhen
Tsinghua University
zhenyj17@mails.tsinghua.edu.cn

Wei Zhang
Tsinghua University
zhangwei15@mails.tsinghua.edu.cn

Zhenyang Dai
Tsinghua University
daizy15@mails.tsinghua.edu.cn

Junjie Mao
Tsinghua University
junjie.mao@enight.me

Yu Chen
Tsinghua University
yuchen@tsinghua.edu.cn

ABSTRACT

As essential components in Linux kernel, kernel modules (kmods) account for over 70% of Linux source code and are heavily dependent on fast evolving and non-stable kernel internal interfaces. Forward and back porting kmods to target versions of Linux kernel is hard but necessary. We conducted a comprehensive study to investigate the characteristics of kernel internal interface changes by analyzing 256 representative patches selected from Linux development history in last 7 years. We gained some new insights into challenges and opportunities on automatic porting of kernel modules. The study allows us a better understanding of the problem and it is useful for designing automated tools to assist in porting kmods.

KEYWORDS

Linux, Kernel modules, Automatic porting

ACM Reference Format:
Yanjie Zhen, Wei Zhang, Zhenyang Dai, Junjie Mao, and Yu Chen. 2018. Is It Possible to Automatically Port Kernel Modules ?. In *9th Asia-Pacific Workshop on Systems (APSys '18), August 27–28, 2018, Jeju Island, Republic of Korea.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3265723.3265732

1 INTRODUCTION

Kernel modules (kmods) provide a flexible way to extend the functionality of Linux kernel. However, interfaces(APIs) between the core part of Linux kernel and kmods are non-stable, and kmods written for a certain version of Linux kernel generally fail to work on other versions. To enjoy enhanced security, boosted performance and new functionality from the fast evolving kernel, developers have to make a continuous effort to forward port their kmods to the rapidly changing APIs. Also, potential users of kmods may rely on earlier kernel versions for stability requirements, and thus have to back port kmods. However, the code base of Linux kernel is large and evolves fast, making forward and back porting kmods a non-trivial task.

Porting kmods to target versions of Linux kernel is both time-consuming and error-prone. Developers have to spend time understanding the compilation error/warning messages and find the corresponding patch that changes the related interfaces manually. They may query through Linux repository using commands like `git log`. However it is a painful task due to the large number of patches, for example 484308 patches were submitted from Linux v3.0 to v4.16. Googling directly is another common way to learn interface changes. Whichever way the developer chooses, he is likely to get distracted by massive irrelevant information. Fortunately, our empirical study shows that nearly 90% of kernel interfaces are used multiple times, and quite a few modules interact with the kernel interface in similar ways, thus it is feasible to build automated tools to learn the change patterns from concrete cases.

While porting kmods manually remains the common way in the Linux community, many automated tools have been built in the last decade. These automated tools mainly fall into two categories. Tools from the first category employ automated program repair techniques [4, 11–13, 27], and rely heavily on compilation error/warning messages to find the correct API-usage change pattern. Tools in the second category employ instances-based inference techniques [1, 9, 10, 17, 20], attempting to directly extract change patterns from code modifications done by kernel developers. Most pattern inference tools are not pragmatic, since Linux kernel patches have some special features, *e.g.* macro, callback

APSys '18, August 27–28, 2018, Jeju Island, Republic of Korea

ACM ISBN 978-1-4503-6006-7/18/08…$15.00
https://doi.org/10.1145/3265723.3265732

function pointers *etc.*, hindering accurate analysis on change patterns.

To have a better understanding of the challenges and feasibility of porting kmods automatically, we selected 200 representative patches from Linux v3.0 to v4.16 and inferred patterns of API changes manually. We investigated the characteristics and representation of API changes in real world. The important results are summarized as follows.

- 38.4% kernel API changes do not trigger any compiler error/warning message, and therefore tools based on automated program repair techniques are severely limited.
- 87.5% of patterns have multiple instances, indicating that automated tools are likely to infer these patterns from multiple instances and provide reference to other use cases of the API. Instances-based inference technique is an alternative way.
- 55.5% of commits involve changes that are irrelevant to API usage updates, and one third of the commits involve multiple patterns. Filtering out irrelevant changes and untangling multiple patterns are major challenges.
- 36.5% of changes are context dependent and over 30% of changes have macro invocation. Current approaches are not able to handle them effectively.

In this paper, we also provide our insights into building automated tools for porting kmods. The rest of this paper is organized as follows. Section 2 gives an overview of porting kmods. Section 3 shows our empirical study on the characteristics and representation of kernel API changes. Section 4 analyzes principle, capability and limitation of existing approaches to kmod porting. Section 5 discusses the challenges and possible techniques in porting kmods.

2 AN OVERVIEW OF PORTING KMODS

One of the difficulties in porting kernel modules is that the Linux kernel code base is extremely large and grows at a fast rate. Yet kmods play an important role in Linux kernel, and account for over 70% of Linux source code. Figure 1 shows the change in the amount of kernel code from kernel version v3.0 to v4.15.

The complexity of Linux kernel makes matters worse, as it often requires kernel developers to have decent knowledge in the C programming language, the assembly language, the computer architecture, *etc.* But in fact, many kernel module developers are not experts in Linux kernel, but merely experts in their own kernel modules. It is also hard for kernel experts to fully understand such a large number of kernel modules. The gap between kernel module developers and kernel developers increases the complexity of porting kernel modules.

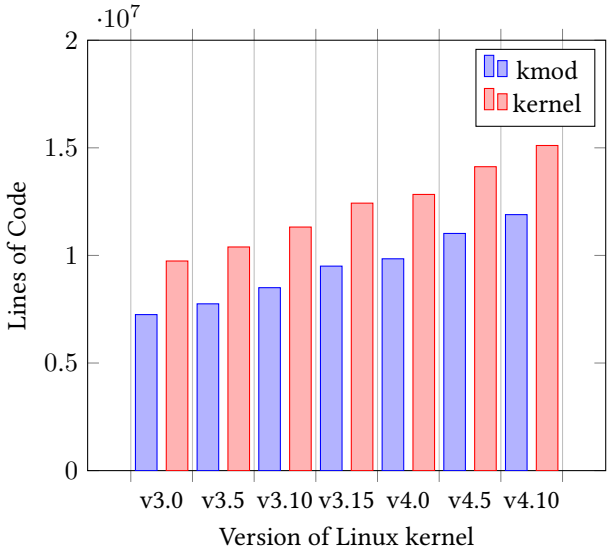

Figure 1: size of kernel and kmods

Porting kernel modules is mainly done manually by developers, using a trial-and-error approach. Firstly, they compile the given kmod against kernel of target version and try to understand the resulting error/warning messages. Secondly, they grep keywords that occur in the messages using commands like `git log -G` in Linux repository, or search keywords through Google. Thirdly, they locate the commits related to the API changes, or Google provides some useful references. Based on these information they could fix the errors/warnings. Finally, developers recompile the modified kmods, if compilation is successful, the modification is considered to be valid. Otherwise, they repeat the above steps. Obviously, this is time-consuming and error-prone.

3 EMPIRICAL STUDY

Understanding internal API changes has been a hot topic in literature, and related studies can be found on the categories, frequency, complexity, impact and trend of API updates [2, 7, 8, 18, 23, 25, 28]. Differing from the above work, our empirical study focuses on the characteristics that help to understand the challenges and applicability of pattern inference techniques for porting kmods.

We present our study on 200 commits from Linux repository over the last 7 years (from v3.0 to v4.16). We use "change pattern" to describe similar changes to an API, which can provide references for automatically porting. In our study, we inferred change patterns manually.

3.1 Change pattern

Firstly, we would briefly explain the meaning of "change pattern" in this paper. Our empirical study shows most of

internal APIs would be used more than once. We call each use case as an instance. We could find a pattern to describe the API changes by abstracting their differences reasonably. Patterns can provide references to other usage instances.

Prior approaches describe "change patterns" in different ways. The following example illustrates the definition of change pattern in our study. We use Semantic Patch Language (SmPL) to describe patterns, which is widely recognized in the Linux community. Listing 1 is part of the code in a commit from the Linux repository. The commit adds a new parameter to the functions acting as the callback `f2` in any ops. Two change patterns are found in these functions, one using the given `inode` instead of fetching one from the given `dentry` (line 5-6 and 23-24), and the other referring to the super block by function `dsb()` instead of `d->d_sb` (line 7-8 and 15-16). Listing 2 shows the semantic patch we generated manually from Listing 1, which describes the first pattern in line 13-14 and the second pattern in line 21-22.

```
1  #define d_inode(d) ((d)->i)
2
3  -int f2(dentry *d) {
4  +int f2(dentry *d, inode *i){
5  -    if(d_inode(d)->flags & F2) { ... }
6  +    if(i->flags & F2) { ... }
7  -    return d->d_sb->mode;
8  +    return dsb(d)->mode;
9  }
10 struct ops ops1 = { .f1 = f1, .f2 = f2, };
11
12 -int f3(dentry *x) {
13 +int f3(dentry *x, inode *i) {
14      ...
15 -    return __f3(x->parent->d_sb);
16 +    return __f3(dsb(x->parent));
17 }
18 struct ops ops2 = { .f2 = f3, };
19
20 -int f4(dentry *d) {
21 +int f4(dentry *d, inode *i) {
22      int ret;
23 -    if (d_inode(d)->flags & F3) { ... }
24 +    if (i->flags & F3) { ... }
25      return ret;
26 }
27 struct ops ops3 = { .f2 = f4, };
```

Listing 1: Example Code

3.2 Dataset

To collect commits with API changes, we select a target module M, fetch all commits that are introduced during the development from version v3.0 to v4.16 (which covers 7 years), randomly pick commits changing both files in M and out of M, and filter out non-essential commits, such as updating

```
1  @rule@
2  identifier s, f;
3  @@
4  struct ops s = { .f2 = f, };
5  @@
6  typedef dentry, inode;
7  identifier rule.f;
8  identifier _arg1, _arg2;
9  @@
10 - f (dentry *_arg1) {
11 + f (dentry *_arg1, inode *_arg2) {
12 <...
13 -   d_inode(_arg1)->flags
14 +   _arg2->flags
15 ...>
16 }
17
18 @@
19 expression E
20 @@
21 - E->d_sb
22 + dsb(E)
23 }
```

Listing 2: Semantic patch describing the pattern in Listing 1

comments or typo fixes, by manual inspection. As is shown in Figure 2, the process is applied to 10 kernel modules in `fs` and `drivers`. There are 2081 commits changing both files in M and out of M. Some commits are related to more than one modules, so we got 1753 commits after removing duplicates. Due to time constraint, a total number of 200 commits are collected in our study.

A few developer could change API in one commit and change API calls in following commits. These cases are rare, so they are out of our consideration.

Target modules	Commits
fs/btrfs	282
fs/ext2	141
fs/ext4	342
fs/jfs	104
fs/xfs	219
drivers/block/drbd	137
drivers/gpu/drm/radeon	291
drivers/net/.../ixgbe	68
drivers/usb/gadget	367
drivers/input/keyboard	130
Total	2081

Figure 2: Our target modules and the number of commits changing both files in M and out of M.

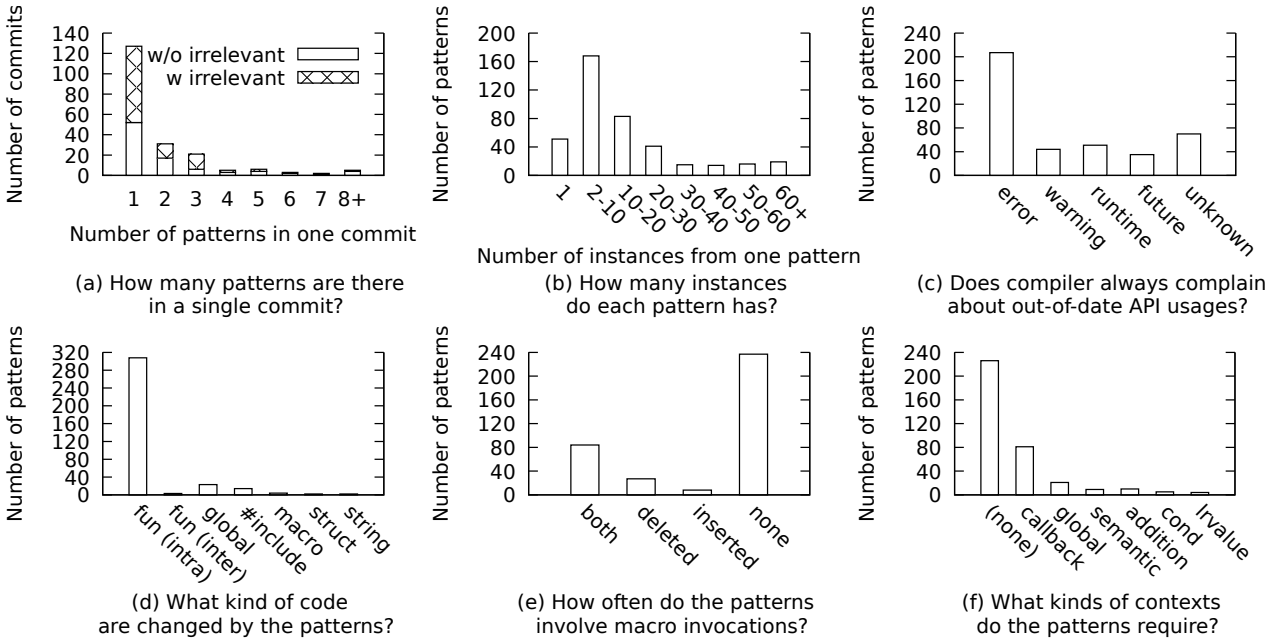

Figure 3: Results of our study

3.3 Validity of the Assumptions on Inputs

It is common in prior work on pattern inference to have some assumptions on the given input. Some of the techniques take manual API usage changes as input and infer patterns by abstracting away irrelevant details [1, 17, 20]. They are applicable only when the given change instances are of the same pattern. Other techniques leverage compilation errors to pinpoint out-of-date API usages, which requiring that such errors are always triggered whenever a pattern is missed and may not apply to certain kernel modules.

This section discusses the characteristics of raw changes in a commit, showing that raw changes do not typically meet the assumptions mentioned above.

(Q1) How many change patterns are there in a single commit? A single commit may resolve multiple issues [5], which is more likely to occur in commits with API updates [8, 19]. While the "one logical change per patch" policy is widely adopted in the Linux community, it does not prevent multiple API updates in a single commit as long as these APIs are considered "logically" related by developers.

Figure 3(a) presents a histogram of the number of API updates in each commit. The 200 commits have 407 patterns in total, and 36.5% (73 out of 200) of the commits involve multiple patterns. Typical cases include changing multiple logically related APIs at the same time or updating an existing API call according to its parameters.

Another fact worth noting is that changes irrelevant to API usage updates are common in the commits. 111 commits (55.5%) in our study involve this kind of changes, such as implementation updates, dead code cleanups, refactoring and module-specific functionality modifications.

(Q2) How many instances does each pattern have? We are concerned about the proportion of APIs that are used more than once, which is the basis of the effectiveness of pattern inference techniques. Figure 3(b) presents the distribution of the number of instances per pattern, which shows that 87.5% of patterns (356 out of 407) have multiple instances.

(Q3) Does compiler always complain about out-of-date API usages? Figure 3(c) shows when developers may notice that a pattern is missed. 251 out of the 407 patterns, accounting for 61.6%, lead to compiler complains, either warning or error, if they are missed. 51 patterns (12.5%) change the runtime behavior without triggering any message during compilation. The other 105 patterns, when missed, neither cause compiler error / warning nor alter program runtime behavior. But 35 of them introduce wrappers to functionalities, such as field accesses and lock operations, which will soon be updated according to the commit logs. APIs introduced by the remaining 70 patterns are not known to be updated in the following commits, but nothing prevents them to be changed in the future.

Summary. In real-world commits, it is common to have multiple patterns (36.5%), as well as irrelevant changes (55.5%), tangled in the same commit. Most patterns (87.5%) have multiple instances and thus can probably be inferred by extracting commonalities among the instances. On the other hand, the coverage of current techniques depending on compiler messages is limited by the fact that a considerable amount of patterns (38.4%), when missed, do not trigger compilation errors or warnings.

3.4 Representation of Changes

Raw changes in a commit should be represented in a structural way so that irrelevant changes can be filtered, instances of different patterns can be untangled and commonalities among instances can be extracted. This part discusses what information should be preserved in such representations.

In this section we focus on the 356 patterns with at least two instances.

(Q4) What kinds of code are changed by the patterns? Figure 3(d) shows how many change patterns touch function definitions (*fun*), global variable definitions (*global*), header inclusion (*#include*) or macro definition (*macro*). 86.5% of the patterns (308 out of 356) apply to function definitions and only 3 of them require changing multiple functions simultaneously. Another 6.4% of the patterns (23 out of 356) update definitions of global values, mostly adding/deleting field initializers to/from structures. The other patterns are seen in header inclusions, macro definitions, conditional compilation directives, structure declarations and string literals. The results show that a majority of patterns can be inferred by sole intra-procedural analysis.

(Q5) How often do the patterns involve macro invocations? Macros in the C language are famous for the complexity they bring to source-to-source transformations [3, 6, 14, 15, 21]. For example, Listing 1 uses `((d)->i)` instead of `d_inode(d)` if it is inferred without considering macro invocations. Such a difference is unacceptable because the incorrect pattern matches nothing in the code and thus does not help upgrade out-of-tree modules.

Figure 3(e) shows the number of patterns that have one or more macro invocations in the deleted lines (*deleted*), the inserted lines (*inserted*) or both (*both*). In our study, 33.4% (119 out of 356) of the patterns have macro invocations in the deleted lines, which must be retained for the patterns to work. Thus, pattern inferences techniques must be able to to preserve macro invocations in the generated patterns.

(Q6) What kinds of contexts do the patterns require? In this paper we refer to code that affects what kind of expression should be updated as contexts required by the pattern. Such contexts should be included in an inferred pattern to accurately locate where the patterns are applied and decide what kinds of changes should be done.

Figure 3(f) shows the kinds of contexts involved in the studied patterns. Apart from 226 patterns (63.5%) requiring no context, the contexts a change pattern depends on mainly fall into the following categories.

- **callback**: 81 patterns in our study are specific to functions assigned to pointers in structures of a particular type, like the pattern in Listing 1. These functions typically act as callbacks on certain events, and are changed due to modifications to the callback signatures.
- **global**: 21 change patterns add or remove field initializers of global structures of a particular type. Fields of the same name but in different types of structures must not be touched.

The other kinds of contexts include high-level semantic information (*semantic*), adjacent statements of an inserted statement (*addition*), and whether an expression matching the deleted line is used as conditions (*conditional*) or location values (*lrvalue*).

Summary. Change patterns of API usage mostly apply locally to definitions of functions (86.5%) and global variables (6.4%), indicating that intra-procedural analysis is a reasonable way to start with. Preserving macro invocations is required in 33.4% of the patterns. Among all kinds of the contexts required by the patterns, callbacks and type of global variables account for the majority (78.5%).

4 CURRENT APPROACHES ON KMOD UPGRADE

Many automated approaches of porting kmods have been proposed. In this section, we analyze the principle, capability, and limitation of current approaches.

4.1 Automated Program Repair

Approaches employing automated program repair techniques mainly assume that unchanged code will cause exceptions when compiling or running directly on kernels of other versions. Nine-tenths of current approaches of porting are based on compilation messages only, while our study shows not all porting issues can be detected by compilation error/warning. Nearly 40% of API changes don't cause any compilation complains.

F. Thung *et al.* [27] propose an automated back-porting approach for Linux kernel drivers. For compilation errors generated when a new version of a driver is compiled against an old version of kernel, they use dichotomy to locate commits in development history that lead to compilation errors, and try to fix compilation errors with change patterns in

the commit. It is limited to drivers in which the compiler generates only one error message. Prequel [11] and Gcc-reduce [11] are a combination of tools for forward and back porting Linux device drivers. They propose an approach that automatically searches commit history in the git repository using GCC compilation error/warning messages, and filters information related to target error/warning. The proposed approach is based on the observation that many drivers interact with the kernel API in similar ways, thus change examples are likely to be available in the code history. Prequel searches git commit history for matching queries to get more accurate results than `git log -G/-S`.

There are some techniques currently focusing on fixing runtime bugs caused by improper conditions, and it is not yet known how these techniques may help concerning forward and back porting of kmods. Angelix [13] synthesizes fixes by introducing additional constraints and controlled symbolic execution. GenProg [4] and SPR [12] search the fix space generated by some predefined sets of transformations.

4.2 Change Pattern Inference

In Linux kernel community, Semantic Patch Language (SmPL) [24] can be used to describe change patterns. It is an abstracted form of patches synthesized from multiple change instances. Coccinelle [22] could automatically generate concrete patches based on SmPL, providing further assistance to developers. But it can't automatically generate semantic patches. When an API is changed, developers must abstract the patch manually and save it for future kmods developers for references.

Several approaches are proposed to infer pattern from multiple instances. Spdiff [1] represents a change instance as a replacement of subexpression, while a change pattern is a replacement of subexpression with a meta variable (which can match any expression). In order to infer change patterns from multiple change instances, Spdiff enumerates all possible ways in which it can introduce metavariable for each concrete replacement, and gets a set of candidate change patterns. Then it take the intersection of all the above sets and pattern with the largest number of non-meta-variables are selected as the final result. In our experience it fails to represent over a half of the instances in the commits, which is probably related to the adoption of imprecise term trees instead of standard ASTs. And it fails to handle changes involving macro. Libsync [20] represents the interface function call and related data and control dependencies in the form of a graph, which is called API usage mode. Each change instance is transformed into addition, deletion, update and other modification operations of nodes in API usage model. Pattern is the maximal frequent subset of edit operations. LASE [17] represents each exemplar as a sequence of AST node edit operations and adopts the longest common subsequence algorithm to extract a template change and generalize identifier names when necessary. Neither Libsync nor LASE discusses how tangled change instances can be divided, how irrelevant changes can be filtered out or how callback-related contexts can be retained in the generated template changes.

In addition to automatically inferring change patterns from multiple instances, some prior works try to introduce additional information or assumptions to infer change patterns. LSDiff [10] and Ref-Finder [9] use predefined rules or templates to identify common types of systematic edits. Inferring template changes from a single exemplar is also known to work with predefined generalization heuristics [16] or interactive guides from developers [26, 29].

5 DISCUSS

In this section, we discuss the feasibility and challenges of porting kmods automatically. And we provide our insights in building an efficient automated tools.

5.1 Feasibility and challenges

Our study shows that 87.5% of patterns have multiple instances. In other words, nearly 90% of kernel API changes can infer patterns to provide assistance to porting. Moreover, 38.4% of kernel API change do not trigger any compiler error or warning message, the capability of approaches based on compilation messages is severely limited.

Based on our empirical study and analysis of current approaches, we present three challenges.

- We found out that 55.5% of commits involve changes that are irrelevant to API usage changes, and one third of the commits involve multiple patterns. It means there are lots of irrelevant concrete changes in kernel patches, which are noises for change pattern inference.
- We found out that 36.5% of changes are context dependent. Callbacks and type of global variables accounting for the majority (78.5%). It is hard for general analysis to handle this situation.
- Kernel is mostly written in the C programming language, adopting some abusive language features, *e.g.* various macro invocations. We found some flexible C macro usages impede the semantic analysis and generation of change pattern.

5.2 Our insights

To build an efficient automated tools of pattern inference for porting, the following key technologies must be addressed.

- The quality of the change pattern depends on the minimal granularity of change instances inside one patch.

We need filter out changes irrelevant to API usage change and untangle multiple patterns in one commit.
- The precision of change pattern depends on the definition of similarity in changes instances. Based on the similarity, we could merge change instances into high-level change patterns, which can provide reference to porting kernel modules.
- The coverage of change pattern depends on the capability to handle changes with callback-related contexts, macros *etc*. Macro invocations and pointers (a callback is essentially a pointer) are challenges in source-to-source transformations, but our study shows that they are prevalent in API changes. For macros, we need to expand macros to get semantic information, at the same time, macro information should be retained. For callback, we need to realize that it's a special kind of pointer, and there is no need for a general but complex method of handling pointers.

6 CONCLUSION

The study of Linux kernel internal interface change shows that pattern inference is a candidate way to automatically port kernel modules. We highlight that the challenges are filtering out irrelevant changes, assessing similarity of change instances to group and infer patterns, handling changes with macro and callback-related context. We hope our study can motivate the design of automated tools to assist in forward and back porting kernel modules.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for helpful comments and suggestions on an earlier version of this paper. This work is supported by National Natural Science Foundation of China (Grant No. 61772303).

REFERENCES

[1] Andersen, J., and Lawall, J. L. Generic patch inference. *Autom. Softw. Eng 17*, 2 (2010), 119–148.

[2] Dig, D., and Johnson, R. E. The role of refactorings in API evolution. In *ICSM* (2005), IEEE Computer Society, pp. 389–398.

[3] Gazzillo, P., and Grimm, R. SuperC: parsing all of C by taming the preprocessor. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012* (2012), J. Vitek, H. Lin, and F. Tip, Eds., ACM, pp. 323–334.

[4] Goues, C. L., Nguyen, T., Forrest, S., and Weimer, W. Genprog: A generic method for automatic software repair. *IEEE Trans. Software Eng 38*, 1 (2012), 54–72.

[5] Herzig, K., and Zeller, A. The impact of tangled code changes. In *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, San Francisco, CA, USA, May 18-19, 2013* (2013), T. Z. 0001, M. D. Penta, and S. Kim, Eds., IEEE Computer Society, pp. 121–130.

[6] Kästner, C., Giarrusso, P. G., Rendel, T., Erdweg, S., Ostermann, K., and Berger, T. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011* (2011), C. V. Lopes and K. Fisher, Eds., ACM, pp. 805–824.

[7] Kim, M., 0001, T. Z., and Nagappan, N. An empirical study of refactoringchallenges and benefits at microsoft. *IEEE Trans. Software Eng 40*, 7 (2014), 633–649.

[8] Kim, M., Cai, D., and 0001, S. K. An empirical investigation into the role of API-level refactorings during software evolution. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu, HI, USA, May 21-28, 2011* (2011), R. N. Taylor, H. C. Gall, and N. Medvidovic, Eds., ACM, pp. 151–160.

[9] Kim, M., Gee, M., Loh, A., and Rachatasumrit, N. Ref-finder: a refactoring reconstruction tool based on logic query templates. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010* (2010), G.-C. Roman and K. J. Sullivan, Eds., ACM, pp. 371–372.

[10] Kim, M., and Notkin, D. Discovering and representing systematic code changes. In *ICSE* (2009), IEEE, pp. 309–319.

[11] Lawall, J., Palinski, D., Gnirke, L., and Muller, G. Fast and precise retrieval of forward and back porting information for linux device drivers. In *2017 USENIX Annual Technical Conference* (2017).

[12] Long, F., and Rinard, M. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015* (2015), E. D. Nitto, M. Harman, and P. Heymans, Eds., ACM, pp. 166–178.

[13] Mechtaev, S., Yi, J., and Roychoudhury, A. Angelix: scalable multi-line program patch synthesis via symbolic analysis. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016* (2016), L. K. Dillon, W. Visser, and L. Williams, Eds., ACM, pp. 691–701.

[14] Medeiros, F. Safely evolving preprocessor-based configurable systems. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016 - Companion Volume* (2016), L. K. Dillon, W. Visser, and L. Williams, Eds., ACM, pp. 668–670.

[15] Medeiros, F., Kästner, C., Ribeiro, M., Nadi, S., and Gheyi, R. The love/hate relationship with the C preprocessor: An interview study. In *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic* (2015), J. T. Boyland, Ed.,

vol. 37 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, pp. 495–518.

[16] Meng, N., Kim, M., and McKinley, K. S. Systematic editing: generating program transformations from an example. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011* (2011), M. W. Hall and D. A. Padua, Eds., ACM, pp. 329–342.

[17] Meng, N., Kim, M., and McKinley, K. S. LASE: locating and applying systematic edits by learning from examples. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013* (2013), D. Notkin, B. H. C. Cheng, and K. Pohl, Eds., IEEE Computer Society, pp. 502–511.

[18] Negara, S., Chen, N., Vakilian, M., Johnson, R. E., and Dig, D. A comparative study of manual and automated refactorings. In *ECOOP 2013 - Object-Oriented Programming - 27th European Conference, Montpellier, France, July 1-5, 2013. Proceedings* (2013), G. Castagna, Ed., vol. 7920 of *Lecture Notes in Computer Science*, Springer, pp. 552–576.

[19] Negara, S., Vakilian, M., Chen, N., Johnson, R. E., and Dig, D. Is it dangerous to use version control histories to study source code evolution? In *ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings* (2012), J. Noble, Ed., vol. 7313 of *Lecture Notes in Computer Science*, Springer, pp. 79–103.

[20] Nguyen, H. A., Nguyen, T. T., Jr., G. W., Nguyen, A. T., Kim, M., and Nguyen, T. N. A graph-based approach to API usage adaptation. In *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA* (2010), W. R. Cook, S. Clarke, and M. C. Rinard, Eds., ACM, pp. 302–321.

[21] Overbey, J. L., Behrang, F., and Hafiz, M. A foundation for refactoring C with macros. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014* (2014), S.-C. Cheung, A. Orso, and M.-A. D. Storey, Eds., ACM, pp. 75–85.

[22] Padioleau, Y., Lawall, J. L., Hansen, R. R., and Muller, G. Documenting and automating collateral evolutions in linux device drivers. In *Proceedings of the 2008 EuroSys Conference, Glasgow, Scotland, UK, April 1-4, 2008* (2008), J. S. Sventek and S. Hand, Eds., ACM, pp. 247–260.

[23] Padioleau, Y., Lawall, J. L., and Muller, G. Understanding collateral evolution in linux device drivers. In *Proceedings of the 2006 EuroSys Conference, Leuven, Belgium, April 18-21, 2006* (2006), Y. Berbers and W. Zwaenepoel, Eds., ACM, pp. 59–71.

[24] Padioleau, Y., Lawall, J. L., and Muller, G. SmPL: A domain-specific language for specifying collateral evolutions in linux device drivers. *Electr. Notes Theor. Comput. Sci 166* (2007), 47–62.

[25] Santos, G., Anquetil, N., Etien, A., Ducasse, S., and Valente, M. T. System specific, source code transformations. In *2015 IEEE International Conference on Software Maintenance and Evolution, ICSME 2015, Bremen, Germany, September 29 - October 1, 2015* (2015), R. Koschke, J. Krinke, and M. P. Robillard, Eds., IEEE Computer Society, pp. 221–230.

[26] Santos, G., Etien, A., Anquetil, N., Ducasse, S., and Valente, M. T. Recording and replaying system specific, source code transformations. In *15th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2015, Bremen, Germany, September 27-28, 2015* (2015), M. W. Godfrey, D. Lo, and F. Khomh, Eds., IEEE Computer Society, pp. 221–230.

[27] Thung, F., Le, X.-B. D., Lo, D., and Lawall, J. L. Recommending code changes for automatic backporting of linux device drivers. In *ICSME* (2016), IEEE Computer Society, pp. 222–232.

[28] Wang, S., Lo, D., and Jiang, L. Understanding widespread changes: A taxonomic study. In *17th European Conference on Software Maintenance and Reengineering, CSMR 2013, Genova, Italy, March 5-8, 2013* (2013), A. Cleve, F. Ricca, and M. Cerioli, Eds., IEEE Computer Society, pp. 5–14.

[29] Zhang, T., Song, M., Pinedo, J., and Kim, M. Interactive code review for systematic changes. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1* (2015), A. Bertolino, G. Canfora, and S. G. Elbaum, Eds., IEEE Computer Society, pp. 111–122.