Who writes what checkers? — Learning from bug repositories

Takeshi Yoshimura, Kenji Kono
Keio University

Abstract

Static code checkers have been useful for finding bugs in large-scale C code. Domain-specific checkers are particularly effective in finding deep/subtle bugs because they can make use of domain-specific knowledge. To develop domain-specific checkers, however, typical bug patterns in certain domains must first be extracted. This paper explores the use of machine learning to help extract bug patterns from bug repositories. We used natural language processing to analyze over 370,000 bug descriptions of Linux and classified them into 66 clusters. Our preliminary work with this approach is encouraging: by investigating one of the 66 clusters, we were able to identify typical bug patterns in PCI device drivers and developed static code checkers to find them. When applied to the latest version of Linux, the developed checkers found two unknown bugs.

1 Introduction

Static code checkers are one of the most useful tools for finding bugs in software. Numerous tools and techniques for static code checkers have been proposed to debug large-scale C code. A good example of large-scale C code is an operating system such as Linux and Windows because the code bases of such OSs are huge and complicated. In fact, static code checkers are widely used to find bugs in Linux or Windows, especially in their device drivers [7] [19] [22] [4] [11] [18] [3] [1] [8].

The key concept of static code checkers is identifying typical bug patterns in code. Some checkers embody well-known rules of programming. A NULL-pointer checker, for example, searches for code locations in which a potentially NULL pointer is dereferenced without checking its nullity. In addition to checkers that embody well-known programming rules, domain-specific checkers are very effective for finding bugs [24] [22] [18] [11]. A field study of Linux bugs in file systems [14] reveals that domain-specific bugs are dominant. An example of domain-specific checkers that makes file system knowledge is one that verifies the file system code does not invoke a memory allocator that may invoke the swapper process. If the swapper is invoked, it will recursively invoke the file system code, which can lead to the deadlock of an entire system.

Unfortunately, it is not straightforward to develop static code checkers that make good use of domain-specific knowledge. Existing code checkers are derived from the insights of experienced developers, from careful field studies of bug fixes [15] and/or from a developer's own experience. Although checkers derived from experience can uncover defects in large-scale software systems, the "experience-based" approach is ad hoc and it is not easy to use it for extracting useful bug patterns. In open-source projects such as Linux, there are many contributors but no formal means to share experiences. In addition, a large-scale software system consists of many components, each of which requires different expertise to develop domain-specific checkers.

In this paper, we explore "mining" domain-specific bug knowledge from the past repositories of bug fixes. We use machine learning to extract bug patterns whose occurrence is statistically frequent in the bug-fix repositories. Using state-of-the-art natural language processing, the bug reports in the repositories are classified based on topic. Closely related bug reports are expected to contain similar bug patterns and to capture domain-specific patterns of bugs. In this work, bug descriptions in English are analyzed instead of code patches that describe actual code changes in C because code patches do not contain any semantic information. If one function call is added to the code, we cannot understand why it has been added to the code only from the patch itself. In contrast, the bug description clearly states why the function call is added. It describes, for example, the function unmasks interrupts.

In this paper, we report our preliminary work with writing checkers based on knowledge extracted from a large number of bug reports. We applied our method to the Linux bug repository (git log) and grouped 370,403 patches in Linux from 2.6.12-rc2 to 3.12-rc5 into 66 clusters based on their content similarity. We then identified nine bug patterns in one cluster that contained device-driver bugs related to interrupt handling and developed checkers for those patterns. These checkers discovered two bugs in PCI device drivers in the latest Linux (version 3.15).

The rest of this paper is organized as follows. Section 2 gives an overview of existing checkers and discusses our motivation. Section 3 describes our strategy for writing checkers and explains natural language processing

and clustering. Section 4 reports our preliminary results with learning bug patterns, our checker implementation, and bugs that we found in Linux device drivers. Section 5 concludes this paper.

2 Background and Motivation

To the best of our knowledge, this work is the first attempt to help checker implementers extract bugs by machine learning. Checker implementers must recognize certain bug patterns or system rules to check before implementing checkers. In this section, we briefly discuss existing methods for extracting bug patterns and our motivation for the work.

2.1 Learning from Bugs

In the context of model checking, the biggest obstacle to finding bugs is simply knowing which rules to check, as mentioned by Engler et al. [8]. Checker implementation faces the same issue. The current methods of extracting bug patterns are to perform field studies on bugs or to investigate implicit coding rules.

Bug field study: Studying real bugs is usually performed with past failure logs and bug reports for target systems such as the Linux kernel [6], Linux file systems [14], other open source systems [26] [15], and Windows [9]. A typical study defines the authors' bug criteria: for example, aging-related Mandelbugs [6], file system semantic bugs [14], misconfiguration types [26], and concurrency bugs (atomicity violation and order violation) [15]. The results of such field studies are useful for extracting bug patterns. However, the identified patterns of bugs closely depend on the bug criteria. Defining a useful and effective set of bug criteria requires intimate knowledge of the target software systems.

Bug studies usually suffer from a large amount of noise in the target resources. Information retrieval and/or machine learning is useful for extracting bug fixes [23] and de-duplicating bug reports [20] [10]. Similar to previous efforts, we analyze textual data in bug reports. However, while the other works focus on extracting a particular collection of bug reports by analyzing the textual characteristics of the reports, we focus on extracting the content of bug descriptions using state-of-the-art natural language processing, as described later.

Extracting system rules from code: Bug patterns can be identified as behavior that deviates from the coding rules. Engler et al. pioneered the idea of checking API usage rules [7] and propose searching for them in the form of pairs of functions that occur together frequently [8], while Yang et al. [25] extract system models from source code by tracking system behavior. Lawall et al. [13] use the insights of experienced developers to extract Linux API protocols. Saha et al. [22] focus on resource release operations inside a function in order to reduce the number of false positives.

While these techniques are all effective for finding bugs, we can only learn patterns defined by some developers and by the methods they propose. Note that these methods also derive from the authors' experience in field studies and software development.

Using specification: Given the complete specification of coding rules, it is easier to recognize bug patterns because all deviations from the specification can be considered bugs. In the case of Windows, the specification of the kernel interfaces to the device drivers is provided. Static Driver Verifier [1] makes use of the specification to find bugs in Windows device drivers. SeL4 [12] enables the formal verification of an entire code by designing a verification-friendly OS architecture. In Linux, device drivers can avoid bugs by automatic synthesis of a formal specification [21].

One approach for reducing many bugs is to check as many common and known bug patterns as possible [5] [17]. The sheer number of the previous works for finding bug patterns ([24] [22] [11] [13] [18] [25] [8] [7], etc.) implies that new bug patterns will continue to show up in the future. Our method of organizing learning bugs in this work assists with prioritizing bug patterns to check on the basis of found bugs in the past.

2.2 Organizing Learning Bugs

In this work, we organize the method of learning bugs so that we can relax the limit of the obtained bug patterns. We do not use the method of extracting system rules from code because this requires insights that might be difficult to obtain. Ultimately, we aim to help field studies define appropriate bug criteria to write checkers.

The root cause of the challenge of obtaining appropriate bug criteria is that the size of target resources is too large to be read in full. Thus, sampling is necessary. A simple approach is to limit the number of investigation by selecting random resources within a certain period of time. The drawback of this approach is that the representativeness is lost. We cannot determine whether the contents of sampled resources are rare cases or not. Another simple approach is to sample the resources that contain bug-like keywords that we already know (e.g., "NULL" or "race"). However, we do not always know such keywords, especially when checker implementers attempt to learn bug patterns related to OS semantics.

Our solution to these challenges is to group similar resources into a cluster based on the resource content, i.e., bug descriptions written in English. Past bug descriptions are useful in that they contain the insights of many developers and their efforts to solve real problems

in target software. We can learn representative bug descriptions from the relatively large clusters for effective checking. Although this approach does not uncover unknown problems in target software, it enables developers to check if problems that many developers have encountered in the past occur in other places for improving the quality of target software. Mining bug patterns from existing bug fixes in a system can give us the insight required to implement effective, domain-specific checkers for the system. One drawback is that we still require learning bug patterns for each target software. Automatic synthesis of domain-specific checkers can reduce the difficulty of this, but it remains challenging to identify violated semantics from bug fixes. In this paper, we report our experience with manual checker implementation.

3 Proposed Method

3.1 Overview

By our methodology, learning and checking for bugs is organized into patch clustering, defining bug criteria, and implementing checkers. Patch clustering is performed through patch weighting with latent Dirichlet allocation (LDA) [2] and top-down clustering [16], with the weight calculated by LDA. Top-down clustering enables us to group similar patches while ensuring their contents appear frequently among all target patches. Top-down clustering cannot calculate the similarity by raw patches written in English, so we need natural language processing and LDA.

LDA lets us weight documents with abstract "topics" that occur in a collection of documents. For example, when a developer finds a bug, he or she describes it using such as "bug", "problem", "failure", "crash", or similar. Such words co-occurrences shape the document's semantics or context and appear more often in documents that contain similar topics. LDA enables us to automate the process of learning the topics by recognizing the pattern of word co-occurrences. We find better parameters by running LDA several times. Each run is around 16 to 40 hours depending on the given parameters. In LDA, topics are represented as the probability distribution of words, i.e., the extent to which each word appears in each topic.

In this work, we use bug reports from the Linux upstream git repository excluding merge commits. Our target resource is a collection of 370,403 patch descriptions from Linux 2.6.12-rc2 on April 2005 to Linux 3.12-rc5 on October 2013.

3.2 Natural Language Processing

Before running LDA, we stem and drop noisy words. By "stemming" we mean grouping together words that have the same meaning but different grammatical variations (e.g., 'leaks', 'leak', and 'leaking'). Noisy words to be reduced include not only well-known "stopwords" (frequently appearing in general English documents: 'is', 'a', 'that', etc.) but also decimal ('8', '16', etc.) and hexadecimal ('0xff', '1e', etc.) numbers, except for ones that are bonded to other letters e.g., "x86". Since information on the development process is not relevant to our work here, we also get rid of paragraphs that contains "Signed-off-by:", which always appears in the signature paragraph in Linux patch reports.

The Linux coding style is well organized and function names frequently appear in bug descriptions for Linux patches. For example, "fat_alloc_inode()" and "ext3_alloc_inode()" are functions for an inode allocation in FAT and ext3. In this case, we should probably mine general inode allocation failures by regarding them as similar. To do so, we use non-alphabet and -digit words as split tokens in addition to spaces. From the example functions, we obtain two sets of words, e.g., ("fat", "alloc", "inode") and ("ext3", "alloc", "inode").

LDA is one of the main topic models in natural language processing. In LDA, a document is assumed to be generated over multiple topics and the topics are assumed to be generated by multiple words that appear in the collection of documents. A topic is inferred by the frequency of word co-occurrences in a given document. For example, if "memory" and "leak" frequently appear in the same report, LDA assigns a topic that weights these two words high and other words low. LDA then weights the assigned topic for reports that contain "memory" and "leak". Using LDA enables documents to be translated into probability sets of topics that can be used as the weight of each document. In this work, we use the Apache Mahout implementation of LDA.

3.3 Top-Down Clustering

The results produced by LDA are not human understandable because of the large amounts of data involved. Therefore, we need an additional grouping method. Top-down clustering is a method of hierarchical clustering in which the algorithm starts from one cluster containing all the data and divides the cluster into two recursively until all clusters becomes small enough to understand (currently, less than 5,000 patches). We implement the division part of the algorithm by 2-means (k-means given k = 2) due to its scalable and concurrent-friendly properties, suitable for the processing of large amounts of data.

LDA does not consider the structure of documents

Table 1: Cluster examples

From left, ID of a cluster, number of patches, and three most probable topics of centroids for a cluster. Each topic is expressed by two most probable words of a topic (if highest probability is >0.9, expressed only by the word).

ID	Size	3 top topics for cluster	ID	Size	3 top topics for cluster
C0	7021	(http, org), (bug, show), (id, cd)	C8	5733	(block, transact), (queue, blk), (length, sg)
C1	8921	(thank, cc), (manag, appli), (miss, add)	C9	5025	(drm, radeon), (auto, engin), (i915, pipe)
C2	5005	(tx, rx), (queue, blk), (packet, skb)	C10	5348	(pci, slot), (bu, driver), (cmd, pcie)
C3	5334	(irq), (interrupt, msi), (handler, c)	C11	5296	(memori, leak), (cpu, hotplug), (node, numa)
C4	5514	(dma, channel), (map), (id, cd)	C12	8078	(page, insert), (map), (scan, direct)
C5	8243	(write), (complet, abort), (caus, race)	C13	5152	(gcc, git), (like, low), (version, increment)
C6	5005	(x86, iommu), (pci, slot), (max, min)	C14	5697	(lock, unlock), (lock, poll), (lock, protect)
C7	5331	(arm, mach), (h, asm), (omap, omap2)	C15	6955	(null, derefer), (pointer, cast), (close, cap)

(e.g., the order of words), which may result in poor results in terms of analyzing Linux patches. A lot of patches for Linux contain various common phrases such as references to bugzilla's URL, error logs, oops call traces, device ID tables, or test scripts. These can frequently appear in bug descriptions, and LDA may mistakenly weight the phrases higher and weaken the actual bug descriptions. To avoid this issue, we divide bug descriptions into paragraphs (most such phrases in Linux appear as a paragraph) and then merge their results when top-down clustering checks the convergence. This merging strategy is just to regard patches in the same cluster as their employing paragraphs. For example, when a cluster has three paragraphs, p1, p2, and p3, whose clusters are c1, c2, and c1, respectively, we regard the clusters to which the patch belongs as c1 and c2 (i.e., we ignore how frequently each cluster appears).

4 Preliminary Results

This section reports the preliminary results of learning bug patterns using the proposed method.

4.1 Patch Clustering

We obtained 66 clusters from over 370,000 patches by extracting clusters containing 5,000 to 10,000 patches. We performed 1,000 iterations on 500 topics for inferring LDA parameters. Other hyper parameters were given as suggested by Mahout's documentation. We stemmed words by using the Porter's stemmer, which is the de facto standard for stemming English [16]. It utilizes suffix stripping based on a rule to conflate inflected words to a root. Stemming does not always preserve the root as a valid word, so the results contain partially corrupted words such as "memori" instead of "memory".

Table 1 shows 16 examples of clusters and their topics. Each cluster is characterized by words that represent topics for the centroid of a cluster. The biggest categories of clusters have topics about OS semantics, including common features (C3: interrupt, C4: DMA, C12: page), devices (C2: network, C8: block, C9: graphics, C10: PCI), and platforms (C6: x86, C7: arm). We found that general, well-known bugs appear frequently in Linux patches (C11: memory leak, C5 and C14: concurrency bugs, C15: null dereference). Other clusters consist of words that often show up in development discussions (C0: URL, C1: thanks, C13: development tools).

Table 2: Nearest bug description of a cluster centroid.

Cluster C7, commit 9cff337 3rd paragraph Topic: (arm, mach), (watchdog, nmi), (specif, code)
So far as I am aware this problem is ARM specific, because only ARM supports software change of the CPU (memory system) byte sex, however the partition table parsing is in generic MTD code. The patch below has been tested on NSLU2 (an IXP4XX based system) with a patch, 10-ixp4xx-copy-from.patch (submitted to Linux-arm-kernel - it's ARM specific) required to make the maps/ixp4xx.c driver work with an LE kernel.

Table 2 shows an example of the bug description and calculated topics in cluster C7. We can confirm words that represent topics characterized bug descriptions. We also investigated more than 30 patches for each cluster in Table 1 and confirmed that the centroid topics mostly represent the patches of each cluster.

4.2 Defining Bug Criteria

As a demonstration, we learn bug patterns from cluster C3 by focusing on patches containing the topic "(free, descriptor)", which appears in 331 patches in this cluster. We identified 160 device driver bugs in the 331 patches. Most bug patterns are identified as mistakes

Table 3: Bug patterns in cluster C3.

ID	Bug description	No.
B1	free_irq() with inconsistent dev_id	41
B2	missing free_irq() (initialization error)	25
B3	free_irq() with an invalid irq number	25
B4	missing free_irq() (module unload)	13
B5	double free_irq()	9
B6	releasing other src before free_irq()	7
B7	releasing pages with interrupt disabled	7
B8	missing free_irq() before device suspend	6
B9	freeing shraed irq with interrupt disabled	5
B10	other	22
B11	not bug	171

on the release of interrupt handlers in device drivers. In our observation, request_irq(), request_threaded_irq() and free_irq() are frequently used APIs. In order to classify reports, we refer to the words representing the topics of each report. For example, we classified a report as "irq leak on an error path during a device probing" (a subclass of B2 in Table 3) because the report was represented by five topics of which the highly probable words are "(memori, leak)", "(irq)", "(probe, driver)", "(error)", and "(path)". Note that topics do not always reflect a bug directly, so we perform manual analyses to learn the bugs precisely.

Request_irq() and request_threaded_irq() are used for the registration of interrupt handler. They require various arguments, including an interrupt request number (irq), a flag of interrupt types, a function pointer for the interrupt handler corresponding to the irq, and an extra variable (dev_id). Dev_id is used when multiple drivers share the irq. free_irq() is used for releasing a registered irq by specifying the irq and dev_id.

Table 3 shows our observation results. We define nine bug patterns (B1-B9). Inconsistent arguments (B1 and B3) are the major category of bug patterns, since a bug finding tool (Coccinelle [13]) found many of them. Missing free_irq() is the second largest category in our observation (B2, B4, B8). Like general bugs, we observed double-frees (B5), order violations (B6 and B9), and deadlock bugs (B7). Only B7 is a major category that is not for interrupt handler registrations; all other categories, including B10, are for them.

4.3 Implementing Checkers

Our observation shows that the balance of request_irq() (or request_threaded_irq()) and free_irq() should be checked. However, pair operations are not called deterministically in the case of B4 and B8. Note that other cases, even B1, can be non-deterministic since an irq is often freed when users attempt to stop drivers. Checking non-deterministic resource releases is challenging.

Our solution is to specify the model that simulates the invocation sequence of typical drivers by instrumenting the driver code as a function and then using it as the entry point of symbolic executions. We assume PCI device drivers, since they are a major source of interrupts. In our model, drivers are invoked sequentially and no device interrupts are delivered because we only need to check the balance of the operations. Although this model potentially ignores the cases in B6, B7, and B9, most of the cases in Table 3 can be covered.

The sequence follows the life-cycle of a typical PCI driver: a device probe, power management (e.g., suspend and resume), and device removal. Each step of the driver invocation is implemented simply by utilizing the corresponding driver callbacks (e.g., for a device probe, we call a registered function; struct pci_driver::probe()).

We used the Clang static analyzer for implementing checkers and checked 593 PCI drivers in Linux 3.15. Our checkers ultimately obtained 230 bug reports and found at least two cases of real bugs (the others are currently under investigation). One of the found cases belongs to B2 in a Cardbus driver and is manifested only when pcmcia_register_socket() fails and the device delivers an interrupt. Missing free_irq() lets the interrupt handler read from or write to resources freed by the error handling of pcmcia_register_socket(). Another case we found belongs to B4 in a network driver. The callback for resetting device states before system shutdown is deleted in order to avoid other problems. This results in a missing free_irq().

5 Conclusion

We proposed a method for checking bug patterns that employs LDA and top-down clustering. Preliminary results show that organizing learning bugs is effective for detecting bugs to improve the quality of target software. Although our bug criteria contain non-deterministic bugs, we were able to detect bugs that derive from insights obtained in this work. As future work, we will continue to find bugs through patch clustering, defining bug criteria, and implementing checkers.

Acknoledgements

We thank the anonymous reviewers for their helpful feedback. This work was partially supported by funding from JSPS Research Fellowships for Young Scientists, Hitachi, Ltd. and Nippon Telegraph and Telephone Corportaion (NTT).

References

[1] BALL, T., BOUNIMOVA, E., COOK, B., LEVIN, V., LICHTENBERG, J., MCGARVEY, C., ONDRUSEK, B., RAJAMANI, S. K., AND USTUNER, A. Thorough static analysis of device drivers. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006 (EuroSys '06)* (2006), pp. 73–85.

[2] BLEI, D. M., NG, A. Y., AND JORDAN, M. I. Latent dirichlet allocation. *Journal of Machine Learning Research 3* (2003), 993 – 1022.

[3] CADAR, C., DUNBAR, D., AND ENGLER, D. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI '08)* (2008), pp. 209–224.

[4] CHIPOUNOV, V., KUZNETSOV, V., AND CANDEA, G. S2e: A platform for in-vivo multi-path analysis of software systems. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)* (2011), pp. 265–278.

[5] CHOU, A., YANG, J., CHELF, B., HALLEM, S., AND ENGLER, D. An empirical study of operating systems errors. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP '01)* (2001), pp. 73–88.

[6] COTRONEO, D., GROTTKE, M., NATELLA, R., PIETRANTUONO, R., AND TRIVEDI, K. Fault triggers in open-source software: An experience report. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE '13)* (2013), pp. 178–187.

[7] ENGLER, D., CHELF, B., CHOU, A., AND HALLEM, S. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation (OSDI '00)* (2000).

[8] ENGLER, D., CHEN, D. Y., HALLEM, S., CHOU, A., AND CHELF, B. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP '01)* (2001), pp. 57–72.

[9] GANAPATHI, A., GANAPATHI, V., AND PATTERSON, D. Windows xp kernel crash analysis. In *Proceedings of the 20th Conference on Large Installation System Administration (LISA '06)* (2006).

[10] JALBERT, N., AND WEIMER, W. Automated duplicate detection for bug tracking systems. In *International Conference on Dependable Systems and Networks (DSN '08)* (2008), pp. 52–61.

[11] KADAV, A., RENZELMANN, M. J., AND SWIFT, M. M. Tolerating hardware device failures in software. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP '09)* (2009), pp. 59–72.

[12] KLEIN, G., ELPHINSTONE, K., HEISER, G., ANDRONICK, J., COCK, D., DERRIN, P., ELKADUWE, D., ENGELHARDT, K., KOLANSKI, R., NORRISH, M., SEWELL, T., TUCH, H., AND WINWOOD, S. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP '09)* (2009), pp. 207–220.

[13] LAWALL, J., BRUNEL, J., PALIX, N., HANSEN, R., STUART, H., AND MULLER, G. Wysiwib: A declarative approach to finding api protocols and bugs in linux code. In *IEEE/IFIP International Conference on Dependable Systems Networks (DSN '09)* (2009), pp. 43–52.

[14] LU, L., ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., AND LU, S. A study of linux file system evolution. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST'13)* (2013), pp. 31–44.

[15] LU, S., PARK, S., SEO, E., AND ZHOU, Y. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII)* (2008), pp. 329–339.

[16] MANNING, C. D., RAGHAVAN, P., AND SCHUETZE, H. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[17] PALIX, N., THOMAS, G., SAHA, S., CALVÈS, C., LAWALL, J., AND MULLER, G. Faults in linux: Ten years later. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)* (2011), pp. 305–318.

[18] PARK, S., LU, S., AND ZHOU, Y. Ctrigger: Exposing atomicity violation bugs from their hiding places. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIV)* (2009), pp. 25–36.

[19] RENZELMANN, M. J., KADAV, A., AND SWIFT, M. M. Symdrive: Testing drivers without devices. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)* (2012), pp. 279–292.

[20] RUNESON, P., ALEXANDERSSON, M., AND NYHOLM, O. Detection of duplicate defect reports using natural language processing. In *Proceedings of the 29th International Conference on Software Engineering (ICSE '07)* (2007), pp. 499–510.

[21] RYZHYK, L., CHUBB, P., KUZ, I., LE SUEUR, E., AND HEISER, G. Automatic device driver synthesis with termite. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP '09)* (2009), pp. 73–86.

[22] SAHA, S., LOZI, J.-P., THOMAS, G., LAWALL, J. L., AND MULLER, G. Hector: Detecting resource-release omission faults in error-handling code for systems software. In *Proceedings of the 2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '13)* (2013), pp. 1–12.

[23] TIAN, Y., LAWALL, J., AND LO, D. Identifying linux bug fixing patches. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)* (2012), pp. 386–396.

[24] WANG, X., ZELDOVICH, N., KAASHOEK, M. F., AND SOLAR-LEZAMA, A. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)* (2013), pp. 260–275.

[25] YANG, J., TWOHEY, P., ENGLER, D., AND MUSUVATHI, M. Using model checking to find serious file system errors. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation (OSDI '04)* (2004).

[26] YIN, Z., MA, X., ZHENG, J., ZHOU, Y., BAIRAVASUNDARAM, L. N., AND PASUPATHY, S. An empirical study on configuration errors in commercial and open source systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)* (2011), pp. 159–172.