PALLAS: Semantic-Aware Checking for Finding Deep Bugs in Fast Path

Jian Huang
Georgia Institute of Technology
Atlanta, GA
jian.huang@gatech.edu

Michael Allen-Bond
Washington State University
Vancouver, WA
michael.allen-bond@wsu.edu

Xuechen Zhang
Washington State University
Vancouver, WA
xuechen.zhang@wsu.edu

Abstract

Software optimization is constantly a serious concern for developing high-performance systems. To accelerate the workflow execution of a specific functionality, software developers usually define and implement a *fast path* to speed up the critical and commonly executed functions in the workflow. However, producing a bug-free fast path is nontrivial. Our study on the Linux kernel discloses that a committed fast path can have up to 19 follow-up patches for bug fixing, and most of them are deep semantic bugs, which are difficult to be pinpointed by existing bug-finding tools.

In this paper, we present such a new category of software bugs based on our *fast-path bug* study across various system software including virtual memory manager, file systems, network, and device drivers. We investigate their root causes and identify five error-prone aspects in a fast path: path state, trigger condition, path output, fault handling, and assistant data structure. We find that many of the deep bugs can be prevented by applying static analysis incorporating simple semantic information. We extract a set of rules based on our findings and build a toolkit PALLAS to check fast-path bugs. The evaluation results show that PALLAS can effectively reveal fast-path bugs in a variety of systems including Linux kernel, mobile operating system, software-defined networking system, and web browser.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging; D.4.5 [*Operating Systems*]: Reliability

Keywords Software Optimization; Fast Path; Semantic Bugs; Static Analysis

ASPLOS'17, April 8-12, 2017, Xi'an, China.

© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ISBN 978-1-4503-4465-4/17/04. . . $15.00

DOI: http://dx.doi.org/10.1145/3037697.3037743

1. Introduction

Implementing a *fast path* is a well-known approach to performance optimization, it has been widely adopted in various software systems [6, 15, 17, 23, 33, 45]. A fast path represents the optimization of the commonly executed sequence of instructions in a workflow, with the goal of accelerating the common cases. The remaining piece of code in the workflow is the normal path or slow path, which is responsible for handling corner cases, errors, and other less common cases. The adoption of fast paths in core systems such as operating systems, file systems, and networking software demonstrably yields boosted performance [28, 41].

However, generating a bug-free fast path is a nontrivial task. A fast path should be triggered in specific conditions, executed without violating the expected logic, and it should be able to return to normal path correctly even when an exception or fault occurs during its execution. To guarantee the correctness of a fast path or specialized code in system software, associated guards and verification have been proposed to be inserted in the workflow to enforce the essential checking for predefined semantics by developers. However, these guards are only implemented in specific systems with end-to-end proof and verification of implementation correctness such as seL4 [16], Ironclad [11], and Optimistic Incremental Specialization [41]. In general, systems such as the Linux kernel, which is widely used on a diversity of hardware platforms, have no such verification framework or mechanism to guarantee the correct implementation of a fast path.

The bugs in fast paths not only hinder the software development process, but also introduce potential new errors, making the fast path a new source of bugs. According to our patch study on fast paths in the Linux kernel, including its virtual memory manager, file systems, network and device drivers (see Table 2), fast-path bugs are pervasive and exist in most of the performance-driven software systems. For a committed fast path that has passed the tests and code reviews, there are up to 19 follow-up patches for fixing the bugs introduced by the fast path itself. And these bug fixes

Table 1. Summary of fast-path bugs detected by PALLAS in our study. PALLAS checks five aspects of the correctness of a fast path: path state, trigger condition, path output, fault handling, and assistant data structures. With simple semantic information provided by users, PALLAS revealed 155 deep bugs in the Linux memory manager (*MM*), file systems (*FS*), network software (*NET*), device drivers (*DEV*), the Chromium web browser (*WB*), the SDN software Open vSwitch (*SDN*), and the Android mobile OS (*MOB*). The last column shows the number of validated bugs (*B*) and warnings (*W*) reported by PALLAS.

Component	Bug Finding	MM	FS	NET	DEV	WB	SDN	MOB	B/W
Path State	immutable states are overwritten	1	1	1	1	3	1	2	10/16
	immutable states are not initialized	1	1	2	1	2	1	2	10/16
	one state does not refer to its correlated state	1	1	1	1	1	1	3	9/15
Trigger Condition	the condition checking for path switch is missing	5	1	3	2	3	2	3	19/21
	the implementation of trigger condition is incomplete	1	1	1	3	2	1	5	14/18
	the order of condition checking is incorrect	1	1	1	1	1	2	1	8/15
Path Output	the return values of slow and fast path should be the same	1	1	2	1	2	1	4	12/19
	the returned values should be one of the defined values	1	1	2	1	3	2	2	12/14
	the returned value should be checked	1	2	1	1	2	1	3	11/18
Fault Handling	the fault handler is missing	2	4	2	4	7	3	5	27/37
Assistant Data Structures	not all elements in a data structure are used in fast path	2	2	1	2	4	2	2	15/21
	an update on a data structure should be followed by an update on its cached version	1	1	1	1	1	1	2	8/14

Table 2. Fast path is buggy. The figure shows the total number of fast paths and patches and the average and maximum number of bugs per fast path observed in the Linux virtual memory manager (*MM*), file systems (*FS*), network (*NET*) and device drivers (*DEV*).

	MM	*FS*	*NET*	*DEV*
Num. of fast paths	16	21	14	14
Num. of bug-fix patches	62	41	41	28
Num. of bugs per path (avg.)	4	2	3	2
Num. of bugs per path (max)	19	17	11	5
Fix time (days on average)	3	8	5	12

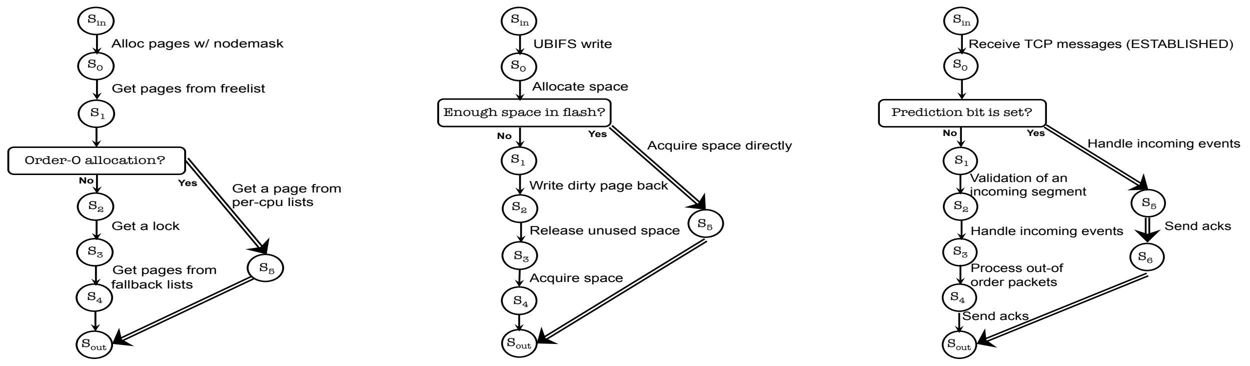
are time-consuming, taking about 3-12 days on average.[1] Furthermore, our study (Section 3) shows that most of the fast-path bugs have led to catastrophic failures such as data loss, system crashes, and incorrect execution of critical functions in software systems (see Table 4).

In practice, a fast path is normally derived from the existing code of a workflow by using techniques like code reduction, code shifting, and state caching [13], making the code base of the workflow complex and error-prone. To make matters worse, the fast path requires developers and code specialization tools to understand the semantics of the target workflow, making bugs difficult to detect. To pinpoint these deep semantic bugs, model checking has proven to be an effective approach. However, most accurate model checkers require developers and testers to manually create models for targeted systems [5, 19, 21], which inhibits their wide adoption. For example, a model checker for detecting bugs in a virtual memory manager cannot be directly applied to file systems, though their bugs are of the same type. This situation inspires us to ask the questions: *is there any common pattern in semantic bugs in fast paths across various software systems, and can we address the reliability challenges presented by fast-path bugs with a general approach?*

In this paper we introduce and examine a new categorization of software bugs. We decompose the first question mentioned above into three basic ones: (1) how is the fast path generated? (2) how does a fast path introduce new bugs? (3) are there any patterns in fast-path bugs that are correlated to software semantics? To help us answer these questions we abstract the essential elements of a fast path into five aspects: path state, trigger condition, path output, fault handling, and assistant data structures. We carefully examined 172 fast-path relevant patches (65 committed fast paths and 117 committed patches for bug fixes) across four core subsystems in the Linux kernel: the virtual memory manager, file systems, network and device drivers. Interestingly, we find that many of the serious bugs are caused by missing one or more of the identified essential elements for a fast path, and a set of findings are identified for each aspect of a fast path as shown in Table 1. Examples include the fast path implementation ignoring the fault handling, absence of a trigger condition for path switch between fast path and slow path, and absence of verification of the output of a fast path before continuing executing the workflow.

Most importantly, we find that most of these bugs can be checked with a static analysis technique using only simple semantic information about the fast path. The rules required for applying this semantic knowledge could be the same across various software systems. For example, once a flag for a trigger condition for path switch is specified, the rule

[1] The time for a bug fix is calculated based on the time difference between the bug-report time and the commit time, it provides a rough sense of how hard to fix the fast-path bugs.



(a) Page allocation (b) UBIFS write (c) TCP receiving

Figure 1. The examples of fast path. (a) Page allocation in the Linux virtual memory manager; (b) file write in the UBIFS file system; (c) packet receiving in a TCP/IP stack. *S* represents the path states at different stages. A path with bolder lines indicates the fast path.

for detecting the missed condition checking can be applied in different software systems. Instead of generating models for each fast path in every target system, the static analysis can significantly reduce the burden for developers and testers.

Based on these findings, we extract a set of simple rules for each aspect described above and build a semantics-aware static checking toolkit PALLAS with the Clang C/C++ compiler front-end [4] to examine the correctness of a fast path. PALLAS is generic and its rules are applicable to fast paths in many kinds of software systems varying from operating systems to user-space applications. PALLAS is systematic in that it implements a set of checks to fast-path bugs in each of the identified aspects. PALLAS is simple as fast path developers and testers only need to initiate the rules with simple protocols that describe the critical data structures and variables which represent specific semantics (e.g., immutable variables and the flags for trigger condition).

To evaluate the applicability and efficiency of PALLAS, we use it to check 90 fast paths in four subsystems of the Linux kernel and three open-source software: the Chromium web browser [3], the Android mobile operating system [2], and an open-source implementation of the software-defined networking Open vSwitch [40]. PALLAS reported 224 warnings with which we manually reproduced and validated 72 bugs in the fast paths in the Linux kernel, and 83 bugs in other open-source software (see Table 1), showing that PALLAS demonstrated an accuracy of 69%. These validated bugs have existed for 3.1 years on average.

The rest of the paper is organized as follows. In Section 2 we present three examples to illustrate the concept of a fast path and demonstrate how a fast path is generated. Section 3 discusses our findings on fast-path bugs in the Linux kernel and present the rules distilled from these findings. Section 4 presents the toolkit PALLAS built based on these rules. We evaluate the applicability of PALLAS in Section 5, and discuss its limitations and related work in Sections 6 and 7 respectively. We conclude this paper in Section 8.

2. Motivating Examples

As a widely used optimization technique for computer software, a fast path is a kind of code specialization that accelerates the common cases of a program via a faster execution workflow. To help us understand what is fast path and how fast path is generated, we use three interesting cases in Linux: page allocation in the virtual memory manager, file write in the *UBIFS* file system, and packet receiving in the TCP/IP network stack.

2.1 Page Allocation in Virtual Memory Manager

Page allocation plays an important role in the virtual memory manager for quickly allocating pages from resource pool to consumers. To allocate pages, the high-level API *__alloc_pages* calls the function *__alloc_pages_nodemask* and *get_page_from_freelist* to locate free pages. Figure 1(a) shows that two paths are generated based on the number of physical pages that will be allocated. For requests that allocate the smallest memory block (i.e., *order* == *0*), the buddy allocator executes the fast path (the path with bolder lines in Figure 1(a)) as they can be served from per-cpu lists without holding a lock. For requests of the high-order allocation, the slow path is executed to acquire a lock and then serve the requests with memory blocks being split or merged in its *fallback* lists.

Such a simple optimization may introduce serious bugs. For instance, both the slow path and fast path share a global variable *gfp_mask* that determines the behavior of the page allocator (e.g., which pages can be allocated and whether the allocator can wait). The global variable should not be modified in page allocation paths. An inadvertent modification in the paths led to an incorrect input state for the following page allocations, further resulting in unexpected results for next page allocations. Without the semantic information provided for this global variable, it is difficult to detect such a deep semantic bug.

2.2 File Write in UBIFS

UBIFS [14, 39] is a file system designed to exploit the potential performance benefits of raw flash. Similar to other file systems, file write is a hot path which is critical for application performance. As indicated in Figure 1(b), a slow path needs to budget pages before writing them to ensure there is enough space in physical flash. The budgeting procedure may trigger write-back operations which add overhead to the normal writes. A fast path is executed for the common case that there is enough space in flash and the budget procedure can be opportunistically skipped.

However, such a fast path may incur data loss if exception handling is not implemented properly. For example, *UBIFS* will switch from the fast path to slow path once it discovers that there is no space in flash. Before switching, a page state returned by the exception handler should be changed from up-to-date to dirty, indicating the page has been written and the write is incomplete. Returning an incorrect page state in the fast path could have a serious consequence that future reads return incorrect data, and the data written by the previous write is lost.

2.3 Packet Receiving in the TCP/IP Stack

Packet processing in the TCP/IP stack is critical to the performance of networked systems. In the Linux kernel, the fast path for TCP/IP protocol is turned on by default to process packets following the specifications defined in RFC793 [44]. It is switched to the slow path under certain conditions such as out-of-order segment arrival, exhaustion of buffer space, receipt of unexpected TCP flags, etc.

As shown in Figure 1(c), *tcp_rcv_established* uses a header prediction bit as the trigger condition for its fast path. If the bit matches *pred_flags*, the fast path is selected to handle incoming messages and the validation checking for each segment will be skipped to reduce the packet processing overhead. In contrast, all the segments will be checked in the slow path. For the packet receiving procedure, both the slow path and fast path should end with the same output state after finishing packet processing, but this is not always true in real implementations. For example, an *if...else* statement in the path could make the path transition to an unexpected state [43], which caused a socket object to be doubly freed.

To summarize, a fast path is developed to speed up commonly executed and critical functions in programs. Fast and slow paths share the same start and end entries in a workflow, but a fast path is derived from a slow path with software optimizations. In practice, many bugs are introduced when a fast path is developed and committed. We refer to the bugs caused by a fast-path commit as *fast-path bugs*.

The use of fast paths is pervasive and they are implemented in many software systems that have performance concerns. However, detecting fast-path bugs is difficult because it requires semantic knowledge. The existing solutions such as using guards and model checking for correctness verification are not widely adopted in practice because of their complicated and strict requirements on code implementations, motivating us to exploit alternative approaches to address this challenge.

3. Characterization Study on Fast-Path Bugs

Through the examples shown in Figure 1, all the workflows for these fast paths are similar, although they are generated with different semantics. However, few studies can be found on the analysis of fast paths and fast-path bugs are far less well-understood. In this section, we present our study on more fast paths across a variety of core software systems. To the best of our knowledge, this is the first characterization study on fast-path bugs.

3.1 Methodology

We identified 404 fast-path patches that account 7% of the total patches committed during the years from 2009 to 2015 and studied 172 bugs that caused serious consequences in 65 fast paths reported in four core subsystems of the Linux kernel, including virtual memory manager, file systems, network and device drivers. We focused on the Linux kernel because it is a well developed, open source operating system and widely deployed as the core software system on a diversity of platforms.

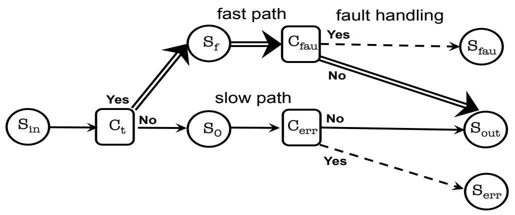


Figure 2. The key elements of a fast path. They are generalized from the committed fast paths in the Linux kernel. S_{in}, S_f, and S_o denote input state, fast path, and slow path state respectively. C_t, C_{fau}, and C_{err} denote conditions to trigger fast path, fault handling, and error output respectively. S_{out}, S_{err}, and S_{fau} denote return values of paths under normal execution, error execution, and fault.

Based on the intuition that a fast path is typically derived from a normal path, we built a tool with the Clang C/C++ compiler front-end [4] to compare the code difference between a fast path and slow path on the same functionality to narrow down our focus on specific data structures, variables, and functions. Besides the code comparison tool, we adopted the methods discussed in [12, 13, 48] for the patch study. Specifically, we manually tag each committed patch and classify them with appropriate labels (e.g., consequences and causes). Online discussions and relevant source code for a fast path are also studied to understand how the fast-path bug was introduced and what its root causes are.

We abstract the fast path and categorize its key elements into five types as shown in Figure 2: path state (e.g., S_{in}, S_f,

Table 4. Fast-path bugs cause serious consequences, including data loss, system crash, system hang, memory leak, incorrect result (silent errors) and performance degradation. For each category of fast path bugs (e.g., path state), the table shows the number of bugs among different types of consequence and its ratio (%) to the total number of bugs in the category.

Consequence	*Path State*	*Trigger Condition*	*Path Output*	*Fault Handling*	*Assistant Data Structures*
Incorrect results	15 (44%)	12 (40%)	12 (33%)	14 (45%)	16 (39%)
Data loss	0 (0%)	0 (0%)	8 (22%)	4 (13%)	7 (17%)
System hang	5 (15%)	2 (7%)	3 (8%)	1 (3%)	4 (10%)
System crash	6 (18%)	4 (13%)	8 (22%)	3 (10%)	6 (15%)
Performance degradation	7 (21%)	11 (37%)	2 (6%)	5 (16%)	7 (17%)
Memory leak	1 (3%)	1 (3%)	3 (8%)	4 (13%)	1 (2%)

Table 3. A distribution of the fast-path bugs in the Linux virtual memory manager (MM), file systems (FS), network (NET), and device drivers (DEV). The bugs are categorized into five types including path state, path condition, path output, fault handling, and assistant data structures. We show the number of patches in a type and its ratio to the total number of patches in a subsystem.

	MM	*FS*	*NET*	*DEV*
Path state	21 (34%)	4 (10%)	5 (12%)	4 (14%)
Conditions	10 (16%)	3 (7%)	14 (34%)	3 (11%)
Path output	12 (19%)	13 (32%)	6 (15%)	5 (18%)
Fault handling	9 (15%)	7 (17%)	5 (12%)	10 (36%)
Data structures	10 (16%)	14 (34%)	11 (27%)	6 (21%)
Total bugs	62	41	41	28

S_o), trigger condition (e.g., C_t, C_{err}, and C_{fau}), path output (e.g., S_{out}, S_{err}, and S_{fau}), fault handling and assistant data structure. Path states represent the input states, intermediate states and final states of a path, e.g., the input parameter for a function and the initial state of a global variable. Trigger conditions determine whether a workflow will switch from one path to another. Path output represents the return value of a path. Fault handling is implemented to process exceptions that happen along with the execution of a path. In many cases, a fast path would also use a set of data structures (e.g., state cache) to assist it to accomplish the optimization.

When a fast path is implemented, bugs could be introduced from these five aspects discussed above. We categorized the fast-path bugs in Table 3. It is interesting to find that these fast-path bugs from different software systems have specific patterns across the five types, although their implementations fulfilled different semantics. This gives us the hint that these semantic bugs could be addressed with general approaches. We will discuss our findings in detail and demonstrate how these findings could help us pinpoint fast-path bugs in the following sections.

3.2 Path State

A large portion of fast-path bugs come from path states (see Table 3). When a fast path is generated, it usually shares the same variables as defined in the slow path, almost all of its initial states are the same as that for the slow path. The state management becomes complicated as the path diverges.

The bugs caused by path states could lead to serious consequences as illustrated in Table 4. For example, an unexpected modification on a node mask could result in incorrect page allocations (e.g., the pages are allocated from a different node, violating the memory allocator policies), an incorrect setting of a page state could make a memory page become unavailable (i.e., memory leak). Therefore, it is important to verify the correct usage of the involved data structures and variables for path states when a fast path is generated.

Although the path states would be changed along with the path execution and their values are determined at runtime, our study on these 34 bugs related to path states in the Linux kernel reveals that about half of these bugs can be detected with static analysis by taking simple semantic information about the involved data structures and variables. We describe the major subtypes of the bugs relevant to path states and the proportion of each subtype, and discuss the real-world examples of these bugs in the following.

Overwriting immutable variables (51%): For immutable variables, it is assumed that they will not be modified by the fast path during the path execution. According to our study, we find that half of the path-state bugs are caused by violating this semantic rule. For instance, in the fast path of page allocation, it has a set of immutable variables *nodemask*, *migratetype*, and *high_zoneidx* to represent the mask bits of candidate nodes for page allocation, the migration type for memory isolation, and the zone type respectively. As shown in Figure 3, the fast path links the value of the immutable variable *migratetype* to *page→private* for future reference. However, the *page→private* would be overwritten when the page was freed to the *freelist* of buddy allocator, causing incorrect value reference for *migratetype* and incorrect page state. With the knowledge of these immutable variables, this type of bugs can be quickly pinpointed.

Correlated variables (20%): In programs, correlated variables appear frequently to express the inherent correlations between multiple variables [25]. This is also true in fast-path implementation. The correlations between multiple path states (i.e., inherent program semantic) are implemented with the correlated variables. Taking the fast path in

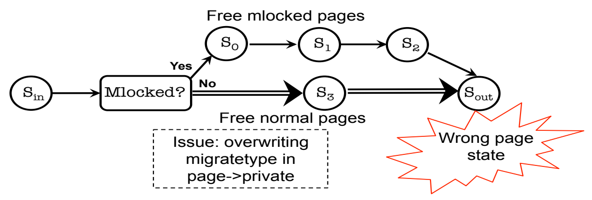


Figure 3. A bug caused by overwriting the immutable variable *migratetype* in the fast path for memory allocation.

page allocation as an example, the variable *preferred_zone* is defined to track the memory zone that is the best candidate for page allocation, but the value of this variable should be a node specified in the immutable variable *nodemask*. Such a constraint defines the correlation between these two variables. In practice, if this semantic is not implemented, an incorrect page allocation (e.g., [31]) could happen.

We find that many such fast-path bugs incurred due to the incomplete implementation of the correlation semantic. In a complicated program like memory management that has many states, guaranteeing the correctness of state correlations is a challenging task. But with the specification of the correlations between variables, many of these bugs can be detected with static analysis. For example, for the correlation semantic *A*→*B*, we can verify it by checking the existence of the correlations between variables *A* and *B* in the control flow graph (CFG). Both the direct and indirect accesses on the correlated variables can be counted as a valid correlation edge which may produce false correlations. We can leverage the approaches of inferring variable access correlations in [25] to validate the variable correlations in a fast path to further reduce the false positives.

Uninitialized immutable variables (7%): It is unexpected that the conventional programming bugs (e.g., defined variables are not initialized) still happened in the well-developed systems like the Linux kernel. We find a set of fast-path bugs belong to this category. For example, an uninitialized page flag [32] defined in a fast path for page allocation in memory *cgroup* may cause an unexpected failure.

Finding 1 : *Most of the path state bugs in fast paths are caused by three reasons: (1) uninitialized immutable variables; (2) immutable variables are overwritten; (3) incomplete implementation of correlated variables.*

Rule 1.1 : *For any specified immutable variable X, X should be initialized;* **Rule 1.2 :** *X should never be overwritten;* **Rule 1.3 :** *For any specified correlated variables X and Y, the correlation between them should be detected in a path.*

3.3 Trigger Condition

Trigger condition determines whether a transition from a slow path to fast path and vice versa will happen. Our study discloses that a large number of fast-path bugs were located in this part. As shown in Table 4, these bugs (third column) could cause the incorrect result, performance degradation, and even system crashes. We present our study on this type of bugs as follows.

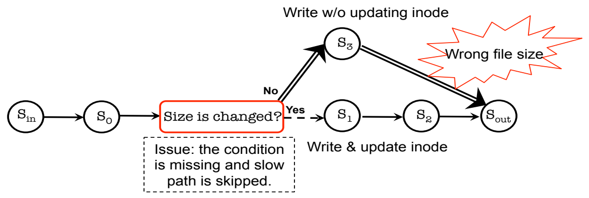


Figure 4. A *missing condition* bug caused incorrect file size in *inode* due to a missing transition from *ocfs2_get_block* fast path to its slow path in the OCFS2 file system.

Missing trigger condition checking (25%): Bugs can happen when the trigger condition is not implemented. The requests which should be served in a slow path, are in fact served in a fast path due to missing the transition from the fast path to slow path, resulting in serious problems such as data inconsistency. Taking a bug [38] reported recently in the Oracle cluster file system (OCFS2) [37] for example, OCFS2 implemented a fast path for fetching disk blocks for the case that the disk file size will not be changed in the near future when a read happens. Once the file size is changed, the slow path in *ocfs2_dio_end_io_write* should be called to update the metadata for the file size. Figure 4 shows that the slow path could be skipped, resulting in data loss.

```
map = rcu_dereference(rxqueue->rps_map)
if(map){
-       if(map->len == 1){
+       if(map->len == 1 &&
+           !rcu_dereference_raw(rxqueue->rps_flow_table)){
+               tcpu = map->cpus[0];
+               if (cpu_online(tcpu))
+                   cpu = tcpu;
```

Figure 5. An incomplete condition checking bug caused a performance bug in network devices.

Incomplete implementation of condition checking (20%): An incomplete implementation of a trigger condition can prematurely force a switch from a slow path to fast path and vice versa. As shown in Figure 5, the condition that whether *rps_flow_table* is ready for the network device to receive packets is not checked [34], resulting in performance degradation as the RPS (Receive Packet Steering) was disabled unexpectedly.

A trigger condition often involves multiple variables whose values will be checked at runtime. Our study on fast-path bugs reveals that many of them are caused by missing the checking for one or two variables. Given that the developed fast path is a ‘white box’ for developers and testers, these bugs can be detected by only taking the semantic information about which variables are used for the trigger condition checking.

Incorrect order of condition checking (12%): A workflow may have multiple path divergences, each of which de-

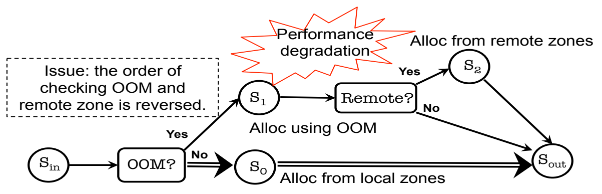


Figure 6. A performance bug caused by the incorrect order of trigger condition checking.

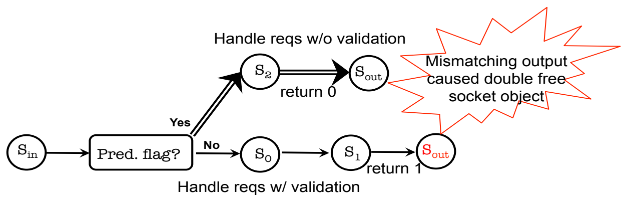

Handle reqs w/o validation
Mismatching output caused double free socket object
S_2
S_{out}
return 0
Yes
S_{in}
Pred. flag?
No
S_0
S_1
return 1
S_{out}
Handle reqs w/ validation

Figure 7. A bug caused by mismatching output in the fast path for $tcp\_rcv\_established$.

pends on one or more trigger conditions. The order of the checks of these trigger conditions determines how the workflow will be executed. We use the memory allocator as an example. When a node does not have enough memory space, there are two possible paths that can satisfy the allocation request: allocating memory from remote memory zone (*Remote*) or execute out-of-memory (*OOM*) procedure to reclaim memory from other processes. Since *OOM* manager may need to kill processes to free pages, its path is considered as a slower path compared to the *Remote* path. Therefore, an optimized path should check *Remote* first, and then try *OOM*. If the order of checking is reversed (see Figure 6), a performance bug [30] was reported. Similar bugs are found in the TCP/IP stack, which caused network packet loss.

Finding 2 : *Most condition checking bugs are caused by three reasons: (1) trigger condition checking for path switch is missing; (2) incomplete implementation of condition checking; (3) incorrect order of condition checking.*

Rule 2.1 : *For any specified variable X for trigger condition checking, X should appear in its flow control statement;* **Rule 2.2 :** *For all specified variables, they should satisfy Rule 2.1;* **Rule 2.3 :** *For any specified trigger conditions X and Y, and X happens before Y, this order should be enforced and detected in the path.*

3.4 Path Output

The output of a fast path depends on its functionality. It could be a page allocated by memory manager, or a socket for TCP connection, or simply an integer denoting the data size that has been successfully written in file systems. Although the output semantic is different, certain generic bug patterns are identified, and these bugs can be avoided at an early stage.

Unexpected output (24%): In many cases, the output states of a fast path are well defined when they are generated by developers. They should belong to a set of expected states (i.e., predefined states or they are highly predictable). We find that 24% of the bugs related to path output are caused by this reason. Upon an unexpected output, serious consequences could happen as shown in Table 4. For example, when a page is allocated in *get_freelist* in the fast path of slab allocation, it must be in *frozen* state [42] to enable per CPU allocations. If the page is returned with other states, the unexpected states could cause incorrect memory allocations or allocation failure.

Mismatching output (39%): Theoretically, for some cases, the output of a fast path should be the same as that of its slow path, e.g., the network packet processing. This is validated in our study. We find that 39% of fast-path output bugs are caused due to the output difference between the fast path and slow path, resulting in failures in upper-level caller functions. As shown in Figure 7, it describes a bug [43] happened in the fast path for receiving network packets in TCP. The caller function assumed both the slow path and fast path should return 0 if success. The incorrect return of the fast path caused double free of the socket object *skb*. In complicated and large-scale systems, the returns of many functions are often well defined (e.g., error codes for returns) [9]. In our study, we observed that many committed fast paths return predefined values such as *EIO*, or values defined by developers themselves. By cross-checking the defined returns of a slow path and fast path, many bugs of this type could be detected.

Missing output checking (8%): Like a slow path, the output of a fast path should be checked to verify whether it executes successfully or not. Unfortunately, we find that this step is usually skipped in real-world implementation, introducing potential risks to the increasingly complicated software system. For example, in the *BtrFS* file system, the return of *btrfs_wait_ordered_range* was not verified in the fast path for IO operations. This caused data loss because its caller function *prepare_page* assumed the optimized version of the IO operation would always execute as expected, even though only a portion of the data is actually written into the file system.

In essence, for this type of bugs, they happened due to the missing of the full implementation of their semantic: whether a fast path's return value should be checked or not. Static analysis method can detect these bugs, but it may suffer from high false positive results as not all the returns of fast paths require validation. Fortunately, this semantic information is simple and straightforward. If this simple knowledge is provided, such type of bugs can be easily detected with static analysis.

Finding 3 : *71% of the fast-path bugs related to path output are caused by three reasons: (1) the output is beyond the predefined states; (2) the output of the fast* *path and slow path does not match; (3) the checking of the fast path's return is missing.*

Rule 3.1 : *For any specified return R of a fast path, R should belong to a set of defined returns or expected states RS ($R \in RS$);* **Rule 3.2 :** *R should be the same as the defined return of the slow path for specified cases;* **Rule 3.3 :** *R should be checked for specified cases.*

3.5 Fault Handling

Fault handling is critical to maintain the correctness and reliability of a fast path. A typical fast path should handle faults timely and correctly.

```
  void transport_generic_free_cmd(struct se_cmd *cmd, ...){
      if (wait_for_tasks)
          transport_wait_for_tasks(cmd);
      /*Handle WRITE failure ...*/
+     if (cmd->state_active){
+         spin_lock_irqsave();
+         target_remove_from_state_list(cmd);
+         spin_unlock_irqrestore();
+     }
```

Figure 8. A bug caused by missing the fault handler implementation in the *SCSI* driver. It caused a memory leak as the failed *cmd* object is not freed.

We study 31 fast-path bugs that are directly related to exception and fault handling and find that the majority of them are caused by **missing fault handler**. No implementation of fault handlers is observed in many fast paths. Such type of bugs can cause serious failures such as data loss in file systems, memory corruption in device drivers as seen in Table 4.

Figure 8 shows an example that a fast path in *SCSI* driver did not handle the failed commands, leading to a memory leak. To fix this bug, developers need to check the command's state, remove the failed command from driver's *state list* and free the failed *cmd* object.

An interesting finding is that these fault handling bugs happened, even though their fault or error states are well defined in the program. These error state definitions actually provide developers and testers the implicit semantics on what errors would happen and should be handled properly in their committed fast paths. Unfortunately, this part is often ignored by developers. Without building a complex model checker or running large scale tests, these deep semantic bugs can be detected with static analysis by leveraging the implicit semantic information.

Finding 4 : *Most of the fault handling bugs in fast paths are caused by missing the fault handling implementation, even though the fault or error codes are well defined.*

Rule 4.1 : *For any specified fault state S, S should appear at least in flow control statement as an indication that it is handled;*

3.6 Assistant Data Structure

Fast path often uses additional data structures to assist it in accomplishing the optimization goal. These assistant structures are typically used to store fast-path states and are critical to the efficiency of the fast path. They can be organized in any format such as *stack*, *binary tree*, *hashmap*, etc.

We investigated 41 fast path bugs related to assistant data structures and find that most of them are caused by the suboptimal organization of these data structures and the uncoordinated updates on them. These bugs can cause catastrophic failures and performance degradation (see Table 4).

Suboptimal organization of data structures (31%): In fast paths, data structures optimization matters as it may significantly affect the performance of these paths. One important aspect is that they are optimized for not touching extra cache lines for better performance. If variables in a data structure are not used in a fast path, they should be separated from others to reduce the size of the data structure. For example, the member variable *i_cindex* was removed from the *inode* structure to fix the bug in [8]. Although storing *i_cindex* only costs 4 bytes on an x86_64 computer, it may have a large memory footprint when hundreds of them are cached in a Linux system. Similar examples can also be found in networking systems. For instance, the variable *struct flowi* was never used in *inet_cork* in the fast path for IPv4, introducing additional overhead for accessing the *inet_cork* data structure. Therefore, with examination of the assistant data structures to find out whether a variable is used or not, a set of performance bugs could be identified.

Stale value caused by uncoordinated updates (26%): An assistant data structure is often used as a cache for path states. However, if the cached entries are not updated or the stale entries are not removed timely, bugs could happen. For example, in the fast path for TCP congestion control, a hash table is used to index multiple TCP congestion control algorithms, each algorithm is associated with a TCP flow. After loading and unloading various modules for the congestion control algorithms, a stale key could be pointed to a wrong module because of the uncoordinated updates [35]. Such a bug could cause abnormal networking performance.

Moreover, when cached states were not cleaned correctly in assistant data structures, file system consistency can be compromised. Taking a bug in NFS for example [36], a fast path is developed to look up a valid inode in *inode cache* for a file system (see Figure 9). If obsoleted inodes were not removed from the cache after a file deletion, the bogus file handle will still be visible to NFS daemons, leading to data inconsistency.

For these bugs, a common pattern is identified: for any data structures that function as caches, when a path state is updated, its corresponding assistant data structure should also be updated accordingly. Otherwise, there is a high possibility that bugs would be introduced by these data structures.

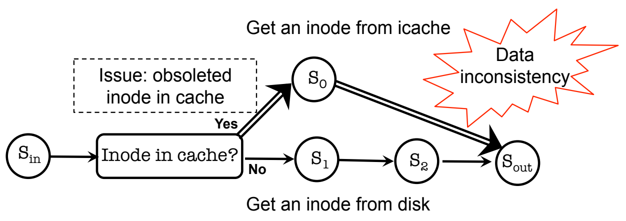


Figure 9. A bug in the fast path of searching a valid inode in *inode cache*(icache) of file system.

Finding 5 : *The assistant data structures in a fast path could introduce new bugs mainly because of two reasons: (1) less care on the organization of the assistant data structures; (2) uncoordinated updates between path states and their cached entries in the assistant data structures.*

Rule 5.1 : *For any specified assistant data structure DS, the unused variables in it should be separated from DS for performance reason;* **Rule 5.2 :** *For any DS which is used for caching path states, an update on one of the path states should be followed by an update on the corresponding DS.*

4. Implementation

According to our study on fast-path bugs discussed in Section 3, certain generic patterns for the root causes of these bugs have been identified. Based on these findings, we extracted a set of simple rules that can be applied to check the correctness of a fast path. Most importantly, these rules can be executed with static analysis to detect deep fast-path bugs by only taking simple semantic information, which dramatically reduces the efforts on building bug-finding tools and running tests. Unlike model checking which requires specific models for a specific system, our approach is generic, which can be applied to a variety of software systems.

Based on the extracted rules, we build a semantic-aware checking toolkit PALLAS which includes five tools: path state checker, trigger condition checker, path output checker, fault handler checker and assistant data structure checker.

The only additional requirement for PALLAS is that it requires users (i.e., developers and testers) to specify the simple semantic information as the input for the static checking rules. We believe such a requirement is less of a concern for several reasons. First, today's developers and testers have much overlap on software development and testing [19, 22]. They know all the semantic knowledge when they develop fast paths. Second, the toolkit PALLAS only needs simple, straightforward and high-level semantic information, for example, the variables which are immutable or will be used for condition checking. PALLAS is built based on Clang. It has several steps to finish the checking for a fast path.

First, it combines the source codes of the target fast path and the relevant header files into a single large file, as the Clang static analyzer cannot execute inter-procedural analysis for multiple files [4, 29].

Second, PALLAS uses Clang to generate the control flow graph which contains all execution paths for the relevant source code within a fast path. A execution path includes four components, function name and input parameters ($Signature$), return value ($Output$), trigger and fault conditions ($Condition$), and updated variables and callee functions ($State$). PALLAS inlines a limited number of callee functions to prevent the path explosion problem. The paths are stored in a database and then symbolically explored by the checkers for bug detection.

Third, users specify the simple semantic information required for each rule as defined in Section 3 with provided protocols. Currently, users need to manually specify the start entry of the slow and fast path, and annotate the semantic information for the five checkers in PALLAS toolkit. We wish to leave the automated approach for extracting semantic information as the future work. As described in Table 5, a few $input$ examples demonstrate how PALLAS' users can specify the input for different rules, in which @$immutable$ represents the specified immutable variables, @$cond$ specifies the variables for trigger condition checking, @$order$ indicates the ordering of condition checking in a fast path.

Fourth, PALLAS filters all the execution paths with the rules for path state, trigger condition, path output, fault handler and assistant data structure respectively. Any checking that violates the rules will be reported as warnings.

Table 5. A simplified example of symbolic extraction for the major components in the execution path of $\_\_alloc\_pages\_nodemask$. The immutable variable gfp_mask was changed, which violates the rule that immutable variables should not be overwritten. S# means a symbolic expression, I# means an integer value, V# means a temporary variable, and E# means a symbol representing the result of an expression. L# denotes a line number.

Type	L#	Symbolic expression
Input		@immutable = gfp_mask
		@cond0 = zone_local
		@cond1 = __GFP_KSWAPD_RECLAIM
		@order0 = @cond0 <@cond1
Signature	50	__alloc_pages_nodemask(gfp_mask, ...)
Condition	32	(E#zone_local(local_zone, zone))
	62	(E#get_page_from_freelist(order, &ac, ...))
	64	(S#gfp_mask) & (I#__GFP_KSWAPD_RECLAIM)
State	54	migratetype = 0
	54	alloc_flags = 0
	67	(V#1) = memalloc_noio_flags(gfp_mask)
	67	**gfp_mask = (V#1)**
	69	(V#2) = __alloc_pages_slowpath(gfp_mask, ...)
	69	page = (V#2)
Output	74	page

Table 6. Software systems evaluated in the paper.

Software	*Version*	*Description*
Linux kernel	4.6	General-purpose OS
Chromium	54.0	Web browser
Android kernel	6.0	OS for mobile devices
Open vSwitch	2.5.0	SDN software

Table 5 shows a path extracted from a simplified example, in which the immutable variable *gfp_mask* was modified in the execution path of *__alloc_pages_nodemask*. The *path state* checker of PALLAS can detect and report it as a warning after static analysis.

5. Evaluation

The goal of our evaluation is to demonstrate that: (1) the efficiency of PALLAS in detecting new semantic bugs in fast paths across a variety of software systems (Section 5.1); (2) the completeness of PALLAS in executing the semantics-aware checking rules (Section 5.2); (3) the plausible reasons for PALLAS to produce false warning reports (Section 5.3).

Experimental Setup: We applied PALLAS to 90 committed fast paths (not including the 65 fast paths discussed in our study) in four subsystems (i.e., virtual memory manager, file systems, network and device drivers) in the Linux kernel, and three different open-source software as shown in Table 6. We check the source code of these fast paths to understand their semantics, and manually specify simple and high-level semantic information for checkers in PALLAS. All the experiments are conducted on a Linux 4.2.0 server machine with Clang 3.6.2 installed.

The execution time for merging source code and building path database using PALLAS can vary from 50 minutes for analyzing the Linux memory manager to 6 hours for analyzing file systems. This is a one-time cost. The input to PALLAS can be written in a few lines of code. In our experience, it took less than an hour to a few days to understand a path until we ensure the input is adequate. Quantifying the effort of using PALLAS is difficult, as it is determined by many factors (e.g., developers' skills and experiences). We expect that much less effort would be required for software developers and testers.

5.1 New Fast Path Bugs

PALLAS reported 224 warnings. The experimental results show that PALLAS took 1-2 minutes to check one fast path on average. We manually examined all of them and identified 155 fast-path bugs (see Table 1). More specifically, cross-checking with mainline Linux kernels helps us confirm that 27 bugs of them are valid in recent Android OS, as Android kernel is derived from Linux, and some Android bugs reported by PALLAS have been fixed in the latest Linux kernel but not been fixed in Android OS yet. For the remaining bugs, we have reported them to the corresponding developer community for confirmation and fix.

It is noted that a fast path may have multiple bugs. The average latent period of these bugs is 3.1 years. We list 34 bugs for each software in Table 7 to illustrate the examples of the fast-path bugs detected by PALLAS.

Specifically, PALLAS finds 12 bugs related to unexpected output. For example, in *Chromium*, developers expected a flag from a handler with the *OpenNaClExecutable* function to ensure a file handle is available for downloading in a fast path. However, the function never returned a value, causing that the fast path is never executed. 14 bugs are reported and related to the incomplete implementation of trigger conditions. For instance, a fast path was implemented for fragmenting TCP packages in Open vSwitch, its trigger conditions should include the checking of the *CHECKSUM_PARTIAL* flag. However, the buggy code missed that checking before entering the fast path, causing a performance bug. PALLAS detects 10 bugs caused by overwriting immutable variables. A bug caused by the modification of the immutable variable *gfp_mark* as described in Section 2.1 is also detected in the Android kernel.

5.2 Completeness

To evaluate the completeness of PALLAS, we collected 62 known fast-path bugs from the bugs covered in our study and synthesized them into the Linux kernel. PALLAS was executed to identify them. Table 8 shows that only one bug (marked *) was missed by PALLAS due to a semantic exception. Specifically, the buggy code returned a page state set as clean, which was incorrect and should be set as dirty. PALLAS did not report a warning because no runtime data (e.g., the value of the page state) is available for the bug analysis.

5.3 False Positives

As discussed in Section 5.1, we use PALLAS to test 90 new fast paths committed in various software, PALLAS reported 224 warnings. We carefully study all the warnings, and manually reproduce them to validate that 155 warnings are bugs: 31% of warnings reported by PALLAS are false positives.

We investigate them and present main sources of false positives for each category of fast-path bugs. (1) Path state: a false alarm can be caused for changing a local immutable state after saving its snapshot in a global data structure, with which developers can restore the state afterward. (2) Trigger condition: some data structures (e.g., dirty bit in page table) which have implicit semantic on path states could also be used as trigger conditions, causing false alarms although specified trigger condition is not detected. (3) Path output: semantic exceptional cases can cause false positives. For example, the output of a fast path can be checked internally in the fast path and completely skipped in its caller function. (4) Fault handler: most false positives are caused by the faults being handled by low-level functions. (5) Assistant data structure: for some data structures which are asyn-

Table 7. List of new bugs discovered by PALLAS. The first three columns show software systems, file names, and fast-path functions in which the bugs are identified. *MM*: virtual memory manager; *FS*: file systems; *NET*: network; *DEV*: device drivers; *WB*: Chromium web browser; *MOB*: Android kernel; *SDN*: Open vSwitch. The fourth column lists error types, including path state (*S*), fault handling (*F*), path conditions (*C*), path output (*O*), and assistant data structures (*D*). The fifth column describes the potential consequences of the bugs. And the last column shows the latent period[2] of the bugs.

Software	File	Fast path operation	Error	Consequence	Years
MM	slab.c	Allocate w/ local pages	[F] missing handler	System crash	6.5
FS	uptodate.c	Insert metadata buffer to cache w/o resizing	[O] missing log output	Inconsistency	2.2
	uptodate.c	Insert new buffer to cache w/o resizing	[F] missing handler	System crash	6.1
	xfs_ialloc.c	Allocate an inode using the free inode btree	[O] wrong output	Inconsistency	2.2
NET	af_unix.c	Send page data w/ socket	[C] incorrect order	Regression	1.1
	tcp_ipv4.c	Get first established socket w/o a lock	[O] wrong lock state	Deadlock	8.4
	udp.c	Send msgs w/o a lock for non-corking case	[O] wrong output	Wrong result	5.4
DEV	cl_page.c	Find Luster page in cache	[O] unexpected output	System crash	3.2
	hvc_console.c	Open w/ an existing port	[F] skipping handler	System crash	5.5
	lov_io.c	I/O initialization when file is striped	[C] missing condition	Regression	3.2
	mpt3sas_base.c	Send fast-path requests to firmware	[D] suboptimal layout	Regression	3.7
	mpt3sas_scsih.c	Turn on fast path for IR physdisk	[F] skipping handler	System crash	2.9
WB	ppb_nacl_private_impl.cc	Download a file w/ PNaCl support	[F] missing handler	System crash	N/A
	ppb_nacl_private_impl.cc	Download a Nexe file w/ PNaCl support	[F] unexpected output	System crash	N/A
	task_queue_impl.cc	Post delayed tasks w/o a lock	[O] wrong return	Wrong result	N/A
	task_queue_impl.cc	Post delayed tasks w/o a lock	[S] suboptimal layout	Regression	N/A
	web_url_loader_impl.cc	Load URL w/ local data	[F] missing handler	System crash	N/A
	wts_terminal_monitor.cc	Get session id w/ physical console	[O] wrong return	Wrong result	N/A
	ScriptValueSerializer.cpp	Write ASCII strings	[F] missing handler	Inconsistency	N/A
	GraphicsContext.cpp	Draw w/ Shader	[F] missing handler	System crash	N/A
	PartitionAlloc.cpp	Allocate pages in the active-page list	[F] wrong handler	Wrong result	N/A
MOB	cpufreq-set.c	Modify only one value of a policy	[O] wrong output	Wrong result	4.6
	macvtap.c	Pin user pages in memory	[F] missing handler	System crash	4.7
	mempolicy.c	Allocate a page w/ a default policy	[S] wrong state	Memory leak	2.1
	mempolicy.c	Allocate a page w/ a default policy	[C] incorrect order	Regression	2.1
	namei.c	Lookup inode w/o a lock	[O] unexpected state	Inconsistency	0.8
	namespace.c	Unmount file systems w/o a lock	[C] skipping slow path	System crash	2.7
	page_alloc.c	Get a page from freelist	[S] immutable state	Wrong result	0.8
	skbuff.c	Reallocate when a skb has a single reference	[C] wrong condition	Memory leak	1.9
	xfs_mount.c	Modify a counter if it is in use	[F] missing handler	Inconsistency	2.3
SDN	dpif-netdev.c	Process in defined fast path	[C] incorrect order	Regression	2.8
	ip6_output.c	Create fragments for not cloned skb	[C] incomplete	Regression	0.5
	netdevice.c	Calculate header offset in fast path	[F] missing handler	System crash	0.5
	vxlan.c	Calculate header offset in fast path	[F] missing handler	System crash	0.5

chronously or lazily updated for better performance, the uncoordinated updates are allowed. To further reduce the false positive number, users can specify more semantic information as they check their developed fast paths.

6. Discussion and Limitations

Although PALLAS can detect many fast-path bugs across various software systems with simple and generic approaches, the toolkit still has limitations that require care .

[2] The latent period for bugs in Chromium is not available in its patch tracking system.

Fast path bug study: In this paper, we studied 172 fast-path bugs from four subsystems in the Linux kernel. As discussed in [48], the size of our study set would be large enough to represent the entire population, according to Central Limit Theorem. The distribution of fast-path bugs in the Linux kernel may not represent the distribution patterns in other software systems, however investigating the bug distribution is not our main focus. The goal of our study is to identify the plausible root causes of fast-path bugs. We built the toolkit PALLAS based on these identified causes, and applied it to detect fast-path bugs in other software systems such as web browsers and mobile operating systems. The re-

Table 8. Completeness of PALLAS' results. The first two columns show bug source and causes, respectively. The third column shows the ratio of detected bugs (D) and the total number of bugs for verification (T).

Bug Source	Bug Causes	D/T
Path State	Overwriting immutable variables	4/4
	Correlated variables	6/6
	Uninitialized immutable variables	2/2
Trigger Condition	Missing condition checking	8/8
	Incomplete implementation	8/8
	Incorrect order of checking	2/2
Path Output	Unexpected output	*5/6
	Mismatching output	8/8
	Missing output checking	2/2
Fault Handling	Missing fault handler	8/8
Assistant Data Structure	Suboptimal organization	6/6
	Stale value	2/2

sults testify that our findings on fast-path bugs are generic. As more causes are identified, more bugs would be found with our approach.

Static analysis rules: The rules used in PALLAS are extracted from our fast-path bug study. We only extracted the rules that can be executed with static analysis, as we believe fast-path bugs exist in a variety of software systems, and a generic approach for discovering bugs is desirable. Our study is also useful for building other bug-finding tools such as model checkers, though how to apply these findings in building model checkers is outside of the scope of this paper.

Simplicity and generality: We would like to emphasize that the main goal of building PALLAS in this paper is to demonstrate the power of detecting fast-path bugs with simple and generic approaches. When a fast path is developed, developers and testers only need to write a few lines of code to specify the simple semantic knowledge (e.g., the variables that should be immutable) through the protocols in PALLAS. During our evaluation, the time we spent on fulfilling the protocols is little (although it is hard to quantify) after understanding each fast path. We believe the actual developers will take less effort because they have a deeper understanding of the code as they implement the fast path.

7. Related Work

Fast path and code specialization: Code specialization [28, 41] which was proposed decades ago, is sharing the same spirit of fast path: optimizing parts of a program for common cases with specialized code. It required associated guards (e.g., type guard and memory guard) to check and verify whether the specialized code is executed following the predefined logics. These guards still suffer from bugs as the target system becomes complicated (e.g., memory manager). In practice, these guards are only implemented in special systems such as seL4 [16], Ironclad [11], Recon [7], COGENT [1] which have end-to-end proof and verification of implementation correctness. In general, for systems such as the Linux kernel, web browser and mobile OS which care about performance and ease of development, there is no verification framework to validate the implementation correctness of their fast paths and specialized code.

Bug and patch characterization study: Bug and patch characterization studies provide patterns and insights for developing effective bug-finding tools. The general approaches used in our study follow the methods discussed in prior bug and patch studies on the Linux memory manager [13], file systems [24], distributed systems [10], concurrency bugs [20, 26], etc. Our study focuses on fast-path bugs in a variety of software systems, including memory manager, file systems, network and device drivers. Moreover, a code comparison tool is built in our study to help us quickly narrow down the search scope for bugs' root causes, based on the observation that a fast path is usually derived from a slow path. Our findings inspire us to build a semantics-aware static analysis tool, and we believe our study is also useful for building other relevant tools.

Model Checking To detect semantic bugs, model checking has been proven to be an effective approach in prior work [19, 46, 47]. Markey et al. [27] specifically discussed the model checking techniques for a program path, however, a naive model checking could traverse a great number of possible paths [18], and a model checker requires specific models for specific systems [5], making the model checking approach less attractive for detecting fast-path bugs. In our paper, we extract basic and simple rules from our findings and apply them into static analysis to build a simple, generic and systematic toolkit to detect fast-path bugs for many kinds of software systems.

8. Conclusions

In this paper, we present the fast-path bugs and conduct a thorough study on them across a variety of representative subsystems in the Linux kernel. We find that many of these deep semantic bugs can be pinpointed with static analysis. We extract a set of rules based on our findings, and build a semantic-aware checking tool PALLAS. PALLAS is simple and powerful and can be applied to different kinds of software systems beyond the Linux kernel. After applying PALLAS to popular software like web browser, mobile OS and software-defined networking system, we detect 155 new fast-path bugs.

Acknowledgments

We would like to thank the anonymous reviewers for their helpful comments and feedback. We also thank Moinuddin K. Qureshi for the discussions on this work. This research was supported in part by NSF ACI-1565338 and WSUV Research Grant.

References

[1] S. Amani, A. Hixon, Z. Chen, C. Rizkallah, P. Chubb, L. O'Connor, J. Beeren, Y. Nagashima, J. Lim, T. Sewell, J. Tuong, G. Keller, T. Murray, G. Klein, and G. Heiser. COGENT: Verifying High-Assurance File System Implementations. In *ASPLOS'16*, Atlanta, GA, Apr. 2016.

[2] Android Open Source Project.
`https://source.android.com/index.html`.

[3] Chromium: An Open-Source Browser Project.
`https://www.chromium.org/Home`.

[4] clang: a C language family frontend for LLVM.
`http://clang.llvm.org/`.

[5] D. Engler and M. Musuvathi. Static Analysis Versus Software Model Checking for Bug Finding. In *VMCAI'04*, 2004.

[6] Fast Path.
`https://en.wikipedia.org/wiki/Fast_path`.

[7] D. Fryer, K. Sun, R. Mahmood, T. Cheng, S. Benjamin, A. Goel, and A. D. Brown. Recon: Verifying file system consistency at runtime. *Trans. Storage*, 8(4), Dec. 2012.

[8] fs: Remove i_cindex from struct inode.
`https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/drivers?id=9fd5746fd3d7838bf6ff991d50f1257057d1156f`.

[9] H. S. Gunawi, C. Rubio-Gonzalez, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and B. Liblit. EIO: Error Handling is Occasionally Correct. In *FAST'08*, 2008.

[10] H. S. Gunawi, M. Hao, T. Leesatapornwongsa, T. Patana-anake, T. Do, J. Adityatama, K. J. Eliazar, A. Laksono, J. F. Lukman, V. Martin, and A. D. Satria. What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems. In *SOCC'14*, Seattle, WA, Nov. 2014.

[11] C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill. Ironclad Apps: End-to-End Security via Automated Full-System Verification. In *OSDI'14*, Broomfield, CO, Oct. 2014.

[12] J. Huang, X. Zhang, and K. Schwan. Understanding Issue Correlations: A Case Study of the Hadoop System. In *SOCC'15*, Kohala Coast, HI, Aug. 2015.

[13] J. Huang, M. K. Qureshi, and K. Schwan. An Evolutionary Study of Linux Memory Management for Fun and Profit. In *USENIX ATC'16*, Denver, CO, June 2016.

[14] A. Hunter. A Brief Introduction to the Design of UBIFS. *Technical Report*.

[15] K. Kelsey, T. Bai, C. Ding, and C. Zhang. Fast Track: A Software System for Speculative Program Optimization. In *CGO'09*, Seattle, WA, Mar. 2009.

[16] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, M. Norrish, R. Kolanski, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal Verification of an OS Kernel. In *SOSP'09*, Big Sky, Montana, Oct. 2009.

[17] A. Kogan and E. Petrank. A Methodology for Creating Fast Wait-Free Data Structures. In *PPoPP'12*, New Orleans, Louisiana, USA, Feb. 2012.

[18] L. Kuhtz. Model Checking Finite Paths and Trees. *PhD thesis, Saarland University*, 2010.

[19] T. Leesatapornwongsa, M. Hao, P. Joshi, J. F. Lukman, and H. S. Gunawi. SAMC: Semantic-Aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems. In *OSDI'14*, Broomfield, CO, Oct. 2014.

[20] T. Leesatapornwongsa, J. F. Lukman, S. Lu, and H. S. Gunawi. TaxDC: A Taxonomy of Non-Deterministic Concurrency Bugs in Datacenter Distributed Systems. In *ASPLOS'16*, Atlanta, GA, Apr. 2016.

[21] D. Lie, A. Chou, D. Engler, and D. L. Dill. A Simple Method for Extracting Models from Protocol Code. In *ISCA'01*, 2001.

[22] T. A. Limoncelli and D. Hughe. LISA'11 Theme – DevOps: New Challenges, Proven Values. *USENIX; login:*, 36(4), Aug. 2011.

[23] X. Liu, C. Kreitz, R. van Renesse, J. Hickey, M. Hayden, K. Birman, and R. Constable. Building Reliable, High-Performance Communication Systems from Components. In *SOSP'99*, Kiawah Island, SC, Dec. 1999.

[24] L. Lu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and S. Lu. A Study of Linux File System Evolution. In *FAST'13*, Feb. 2013.

[25] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou. MUVI: Automatically Inferring Multi-Variable Access Correlations and Detecting Related Semantic and Concurrency Bugs. In *SOSP'07*, stevenson, Washington, Oct. 2007.

[26] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from Mistakes - A Comprehensive Study on Real World Concurrency Bug Characteristics. In *ASPLOS'08*, Seattle, WA, Mar. 2008.

[27] N. Markey and P. Schnoebelen. Model Checking a Path. *Technical Report*, 2003.

[28] D. McNamee, J. Walpole, C. Pu, C. Cowan, C. Krasic, A. Goel, and P. Wagle. Specialization Tools and Techniques for Systematic Optimization of System Software. *ACM Transactions on Computer Systems*, 19(2).

[29] C. Min, S. Kashyap, B. Lee, C. Song, and T. Kim. Cross-checking Semantic Correctness: The Case of Finding File System Bugs. In *SOSP'15*, Monterey, CA, Oct. 2015.

[30] mm: page_alloc: spill to remote nodes before waking kswapd.
`https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/mm/page_alloc.c?id=3a025760fc158b3726eac89ee95d7f29599e9dfa`.

[31] mm:fix deferred congestion timeout if preferred zone is not allowed.
`https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=f33261d75b88f55a08e6a9648cef73509979bfba`.

[32] mm/memcontrol.c: fix uninitialized variable use in mem_cgroup_move_parent().
`https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/mm/memcontrol.c?id=8dba474f034c322d96ada39cb20cac711d80dcb2`.

[33] D. Mosberger and L. L. Peterson. Making Paths Explicit in the Scout Operating System. In *OSDI'96*, Oct. 1996.

[34] net: Check rps_flow_table when RPS map length is 1. `https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/net?id=8587523640441a9ff2564ebc6efeb39497ad6709`.

[35] net: tcp: add key management to congestion control. `https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/net?id=c5c6a8ab45ec0f18733afb4aaade0d4a139d80b3`.

[36] nfsd/create race fixes, infrastructure. `http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/fs/inode.c?id=261bca86ed4f7f391d1938167624e78da61dcc6b`.

[37] OCFS2 - Oracle Cluster File System for Linux. `http://www.oracle.com/us/technologies/linux/025995.htm`.

[38] ocfs2: fix disk file size and memory file size mismatch. `https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/fs?id=ce170828e24959c69e7a40364731edc0535c550f`.

[39] P. Olivier, J. Boukhobza, and E. Senn. On Benchmarking Embedded Linux Flash File Systems. *Technical Report*.

[40] Production Quality, Multilayer Open Virtual Switch. `http://openvswitch.org/`.

[41] C. Pu, T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, and K. Zhang. Optimistic Incremental Specialization: Streamlining a Commercial Operating System. In *SOSP'95*, CO, USA, Dec. 1995.

[42] slub: Add frozen check in __slab_alloc. `http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/mm/slub.c?id=507effeaba29bf724dfe38317fbd11d0fe25fa40`.

[43] tcp: Fix slab corruption with ipv6 and tcp6fuzz. `http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/net/ipv4/tcp_input.c?id=9ae27e0adbf471c7a6b80102e38e1d5a346b3b38`.

[44] Transmission Control Protocol. `https://tools.ietf.org/html/rfc793`.

[45] W. Xu, S. Kumar, and K. Li. Fast Paths in Concurrent Programs. In *PACT'04*, 2004.

[46] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using Model Checking to Find Serious File System Errors. In *OSDI'04*, San Francisco, CA, Dec. 2004.

[47] J. Yang, C. Sar, and D. Engler. EXPLODE: A Lightweight, General System for Finding Serious Storage System Errors. In *OSDI'06*, Seattle, WA, Nov. 2006.

[48] D. Yuan, Y. Luo, X. Zhuang, G. R. Rodrigues, X. Zhao, Y. Zhang, P. U. Jain, and M. Stumm. Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems. In *OSDI'14*, Broomfield, CO, Oct. 2014.