EIO: Error Handling is Occasionally Correct

Haryadi S. Gunawi, Cindy Rubio-González,
Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Ben Liblit
Computer Sciences Department, University of Wisconsin-Madison

Abstract

The reliability of file systems depends in part on how well they propagate errors. We develop a static analysis technique, EDP, that analyzes how file systems and storage device drivers propagate error codes. Running our EDP analysis on all file systems and 3 major storage device drivers in Linux 2.6, we find that errors are often incorrectly propagated; 1153 calls (13%) drop an error code without handling it.

We perform a set of analyses to rank the robustness of each subsystem based on the completeness of its error propagation; we find that many popular file systems are less robust than other available choices. We confirm that write errors are neglected more often than read errors. We also find that many violations are not corner-case mistakes, but perhaps intentional choices. Finally, we show that inter-module calls play a part in incorrect error propagation, but that chained propagations do not. In conclusion, error propagation appears complex and hard to perform correctly in modern systems.

1 Introduction

The robustness of file systems and storage systems is a major concern, and rightly so [32]. Recent work has shown that file systems are especially unreliable when the underlying disk system does not behave as expected [20]. Specifically, many modern commodity file systems, such as Linux ext3 [31], ReiserFS [23], IBM's JFS [1], and Windows NTFS [27], all have serious bugs and inconsistencies in how they handle errors from the storage system. However, the question remains unanswered as to why these fault-handling bugs are present.

In this paper, we investigate what we believe is one of the root causes of deficient fault handling: *incorrect error code propagation*. To be properly handled, a low-level error code (*e.g.*, an "I/O error" returned from a device driver) must be correctly propagated to the appropriate code in the file system. Further, if the file system is unable to recover from the fault, it may wish to pass the error up to the application, again requiring correct error propagation.

Without correct error propagation, any comprehensive failure policy is useless: recovery mechanisms and policies cannot be invoked if the error is not propagated. Incorrect error propagation has been a significant problem in many systems. For example, self-healing systems cannot heal themselves if error signals never reach the self-recovery modules [6, 26], components behind an interface do not receive error notifications [16], and distributed systems often obtain misleading error codes [15, 30], which turns into frustration for human debugging. In summary, if errors are not propagated, then the effort spent detecting and recovering from those errors [4, 5, 18, 21, 22, 28, 29] is worthless.

To analyze how errors are propagated in file and storage system code, we have developed a static source-code analysis technique. Our technique, named *Error Detection and Propagation (EDP)* analysis, shows how error codes flow through the file system and storage drivers. EDP performs a dataflow analysis by constructing a function-call graph showing how error codes propagate through return values and function parameters.

We have applied EDP analysis to all file systems and 3 major storage device drivers (SCSI, IDE, and Software RAID) implemented in Linux 2.6. We find that *error handling is occasionally correct*. Specifically, we see that low-level errors are sometimes lost as they travel through the many layers of the storage subsystem: out of the 9022 function calls through which the analyzed error codes propagate, we find that 1153 calls (13%) do not correctly save the propagated error codes.

Our detailed analysis enables us to make a number of conclusions. First, we find that the more complex the file system (in terms of both lines of code and number of function calls with error codes), the more likely it is to incorrectly propagate errors; thus, these more complex file systems are more likely to suffer from silent failures. Second, we observe that I/O write operations are more likely to neglect error codes than I/O read operations. Third, we find that many violations are not corner-case mistakes: the return codes of some functions are consistently ignored, which makes us suspect that the omissions are intentional. Finally, we show how inter-module calls play a major part in causing incorrect error propagation, but that chained propagations do not.

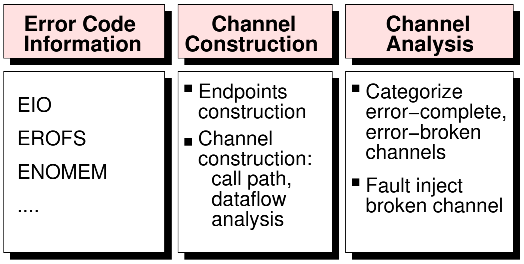


Figure 1: **EDP Architecture.** *The diagram shows the framework for Error Detection and Propagation (EDP) analysis of file and storage systems code.*

The rest of this paper is organized as follows. We describe our methodology and present our results in Section 2 and 3 respectively. To understand the root causes of the problem, we perform a set of deeper analyses in Section 4. Section 5 and 6 discuss future work and related work respectively. Finally, Section 7 concludes.

2 Methodology

To understand the propagation of error codes, we have developed a static analysis technique that we name *Error Detection and Propagation (EDP)*. In this section, we identify the components of Linux 2.6 that we will analyze and describe EDP.

2.1 Target Systems

In this paper, we analyze how errors are propagated through the file systems and storage device drivers in Linux 2.6.15.4. We examine all Linux implementations of file systems that are located in 51 directories. These file systems are of different types, including disk-based file systems, network file systems, file system protocols, and many others. Our analysis follows requests through the virtual file system and memory management layers as well. In addition to file systems, we also examine three major storage device drivers (SCSI, IDE, and software RAID), as well as all lower-level drivers. Beyond these subsystems, our tool can be used to analyze other Linux components as well.

2.2 EDP Analysis

The basic mechanism of EDP is a dataflow analysis: EDP constructs a function-call graph covering all cases in which error codes propagate through return values or function parameters. To build EDP, we harness C Intermediate Language (CIL) [19]. CIL performs source-to-source transformation of C programs and thus can be used in the analysis of large complex programs such as the Linux kernel. The EDP analysis is written as a CIL extension in 4000 lines of code in the OCaml language.

Subsystem	**Single (seconds)**	**Full (seconds)**	Subsystem Size (Kloc)
VFS	**4**	–	34
Mem. Mgmt.	**3**	–	20
XFS	**8**	**13**	71
ReiserFS	**3**	**8**	24
ext3	**2**	**7**	12
Apple HFS	**1**	**6**	5
VFAT	**1**	**5**	1
All File Systems Together		**47**	372

Table 1: **EDP Performance.** *The table shows the EDP runtime for different subsystems. "Single" runtime represents the time to analyze each subsystem in isolation without interaction with other subsystems (e.g., VFS and MM). "Full" runtime represents the time to analyze a file system along with the virtual file system and the memory management. The last row reports the time to analyze all of the file systems together.*

The abstraction that we introduce in EDP is that error codes flow along *channels*, where a channel is the set of function calls between where an error code is first generated and where it is terminated (*e.g.*, by being either handled or dropped). As shown in Figure 1, EDP contains three major components. The first component identifies the error codes that will be tracked. The second constructs the channels along which the error codes propagate. Finally, the third component analyzes the channels and classifies each as being either complete or broken.

Table 1 reports the EDP runtime for different subsystems, running on a machine with 2.4 GHz Intel Pentium 4 CPU and 512 MB of memory. Overall, EDP analysis is fast; analyzing all file systems together in a single run only takes 47 seconds. We now describe the three components of EDP in more detail.

2.2.1 Error Code Information

The first component of EDP identifies the error codes to track. One example is `EIO`, a generic error code that commonly indicates I/O failure and is used extensively throughout the file system; for example, in ext3, `EIO` touches 266 functions and propagates through 467 calls. Besides `EIO`, many kernel subsystems commonly use other error codes as defined in `include/asm-generic/errno.h`. In total, there are hundreds of error codes that are used for different purposes. We report our findings on the propagation of 34 basic error codes that are mostly used across all file systems and storage device drivers. These error codes can be found in `include/asm-generic/errno-base.h`.

2.2.2 Channel Construction

The second component of EDP constructs the *channel* in which the specified error codes propagate. A channel can be constructed from function calls and asynchronous wake-up paths; in our current analysis, we focus only on function calls and discuss asynchronous paths in Section 5.3.

We define a channel by its two endpoints: generation and termination. The *generation endpoint* is the function that exposes an error code, either directly through a return value (*e.g.*, the function contains a `return -EIO` statement) or indirectly through a function argument passed by reference. After finding all generation endpoints, EDP marks each function that propagates the error codes; *propagating functions* receive error codes from the functions that they call and then simply propagate them in a return value or function parameter. The *termination endpoint* is the function in which an error code is no longer propagated in the return value or a parameter of the function.

One of the major challenges we address when constructing error channels is handling function pointers. The typical approach for handling function pointers is to implement a points-to analysis [13] that identifies the set of real functions each function pointer might point at; however, field-sensitive points-to analyses can be expensive. Therefore, we customize our points-to analysis to exploit the systematic structure that these pointers exhibit.

First, we keep track of all structures that have function pointers. For example, the VFS read and write interfaces are defined as fields in the `file_ops` structure:

```
struct file_ops {
    int (*read)  ();
    int (*write) ();
};
```

Since each file system needs to define its own `file_ops`, we automatically find all global instances of such structures, look for the function pointer assignments within the instances, and map function-pointer implementations to the function pointer interfaces. For example, ext2 and ext3 define their file operations like this:

```
struct file_ops ext2_f_ops {
    .read  = ext2_read;
    .write = ext2_write;
};
struct file_ops ext3_f_ops {
    .read  = ext3_read;
    .write = ext3_write;
};
```

Given such global structure instances, we add the interface implementations (*e.g.*, `ext2_read`) to the implementation list of the corresponding interfaces (*e.g.*, `file_ops→read`). Although this technique connects most of the mappings, a function pointer assignment could still occur in an instruction rather than in a global structure instance. Thus, our tool also visits all functions and finds any assignment that maps an implementation to an interface. For example, if we find an assignment such as `f_op->read = ntfs_read`, then we add `ntfs_read` to the list of `file_ops→read` implementations.

In the last phase, we change function pointer calls to direct calls. For example, if VFS makes an interface call such as `(f_op->read)()`, then we automatically rewrite such an assignment to:

```
switch (...) {
    case ext2:  ext2_read();  break;
    case ext3:  ext3_read();  break;
    case ntfs:  ntfs_read();  break;
    ...
}
```

Across all Linux file systems and storage device drivers, there are 191 structural interfaces (*e.g.*, `file_ops`), 904 function pointer fields (*e.g.*, `read`), 5039 implementations (*e.g.*, `ext2_read`), and 2685 function pointer calls (*e.g.*, `(f_op->read)()`). Out of 2865 function pointer calls, we connect all except 564 calls (20%). The unconnected 20% of calls are due to indirect implementation assignment. For example, we cannot map assignment such as `f_op->read = f`, where `f` is either a local variable or a function parameter, and not a function name. While it is feasible to traceback such assignments using stronger and more expensive analysis, we assume that major interfaces linking modules together have already been connected as part of global instances. If all calls are connected, more error propagation chain can be analyzed, which means more violations are likely to be found.

2.2.3 Channel Analysis

The third component of EDP distinguishes two kinds of channels: error-complete and error-broken channels. An *error-complete* channel is a channel that minimally checks the occurrence of an error. An error-complete channel thus has this property at its termination endpoint:

$$\exists \; if \; (expr) \; \{ \; ... \; \}, \; where \\ errorCodeVariable \subseteq expr$$

which states that an error code is considered checked if there exist an `if` condition whose expression contains the variable that stores the error code. For example, the function `goodTerminationEndpoint` in the code segment below carries an error-complete channel because the function saves the returned error code (line 2) and checks the error code (line 3):

```
void goodTerminationEndpoint() {
    int err = generationEndpoint();
    if (err)
        ...
}
int generationEndpoint() {
    return -EIO;
}
```

Note that an error could be checked but not handled properly, *e.g.* no error handling in the `if` condition. Since error handling is usually specific to each file system, and hence there are many instances of it, we decided to be "generous" in the way we define how error is handled, *i.e.* by just checking it. More violations might be found when we incorporate all instances of error handling.

An *error-broken* channel is the inverse of an error-complete channel. In particular, the error code is either *unsaved*, *unchecked*, or *overwritten*. For example, the function `badTerminationEndpoint` below carries an error-broken channel of unchecked type because the function saves the returned error code (line 2) but it never checks the error before the function exits (line 3):

```
1 void badTerminationEndpoint() {
2     int err = generationEndpoint();
3     return;
4 }
```

An error-broken channel is a serious file system bug because it can lead to a silent failure. In a few cases, we inject faults in error-broken channels to confirm the existence of silent failures. We utilize our block-level fault injection technique [20] to exercise error-broken channels that relate to disk I/O. In a broken channel, we look for two pieces of information: which workload and which failure led us to that channel. After finding the necessary information, we run the workload, inject the specific block failure, and observe the I/O traces and the returned error codes received in upper layers (*e.g.*, the application layer) to confirm whether a broken channel leads to a silent failure. The reader will note that our fault-injection technique is limited to disk I/O related channels. To exercise all error-broken channels, techniques such as symbolic execution and directed testing [9, 10] that simulate the environment of the component in test would be of great utility.

2.2.4 Limitations

Error propagation has complex characteristics: correct error codes must be returned; each subsystem uses both generic and specific error codes; one error code could be mapped to another; error codes are stored not only in scalar variables but also in structures (*e.g.*, control blocks); and error codes flow not only through function calls but also asynchronously via interrupts and callbacks. In our static analysis, we have not modeled all these characteristics. Nevertheless, by just focusing on the propagation of basic error codes via function call, we have found numerous violations that need to be fixed. A more complete tool that covers the properties above would uncover even more incorrect error handling.

3 Results

We have performed EDP analysis on all file systems and storage device drivers in Linux 2.6.15.4. Our analysis studies how 34 basic error codes (*e.g.*, `EIO` and `ENOMEM`) defined in `include/asm-generic/errno-base.h` propagate through these subsystems. We examine these basic error codes because they involve thousands of functions and propagate across thousands of calls.

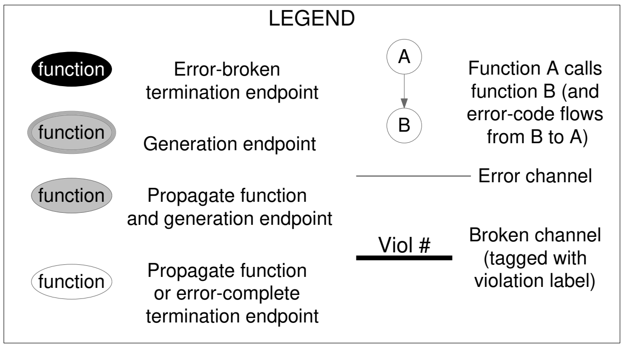
In these results, we distinguish two types of violations that make up an error-broken channel: unsaved and unchecked error codes (overwritten codes have been deferred to future work; see Section 5.1 for more information). An *unsaved error code* is found when a callee propagates an error code via the return value, but the caller does not save the return value (*i.e.*, it is treated as a void-returning call even though it actually returns an error code). Throughout the paper, we refer to this type of broken channel as a "*bad call*." An *unchecked error code* is found when a variable that may contain an error code is neither checked nor used in the future; we always refer to this case as an unchecked code.

3.1 Unsaved Error Codes

First, we report the number of error-broken channels due to a caller simply not saving the returned error code (*i.e.*, the number of bad calls). The simplified HFS code below shows an example of unsaved error code. The function `find_init` accepts a new uninitialized `find_data` structure (line 2), allocates a memory space for the `search_key` field (line 3), and returns `ENOMEM` error code when the memory allocation fails (line 5). However, one of its callers, `file_lookup`, does not save the returned error code (line 10) but tries to access the `search_key` field which still points to `NULL` (line 11). Hence, a null-pointer dereference takes place and the system could crash or corrupt data.

```
 1 // hfs/bfind.c
 2 int find_init(find_data *fd) {
 3     fd->search_key = kmalloc(..)
 4     if (!fd->search_key)
 5         return -ENOMEM;
 6     ...
 7 }
 8 // hfs/inode.c
 9 int file_lookup() {
10     find_init(fd); /* NOT-SAVED E.C */
11     fd->search_key->cat = ...; /* BAD!! */
12     ...
13 }
```

Viol#	Caller →	Callee	Filename	Line#
A	file_lookup	find_init	inode.c	493
B	fill_super	find_init	super.c	385
C	lookup	find_init	dir.c	30
D	brec_updt_prnt	__brec_find	brec.c	405
E	brec_updt_prnt	__brec_find	brec.c	345
F	cat_delete	free_fork	catalog.c	228
G	cat_delete	find_init	catalog.c	213
H	cat_create	find_init	catalog.c	95
I	file_trunc	free_exts	extent.c	507
J	file_trunc	free_exts	extent.c	497
K	file_trunc	find_init	extent.c	494
L	ext_write_ext	find_init	extent.c	135
M	ext_read_ext	find_init	extent.c	188
N	brec_rmv	__brec_find	brec.c	193
O	readdir	find_init	dir.c	68
P	cat_move	find_init	catalog.c	280
Q	brec_insert	__brec_find	brec.c	145
R	free_fork	free_exts	extent.c	307
S	free_fork	find_init	extent.c	301



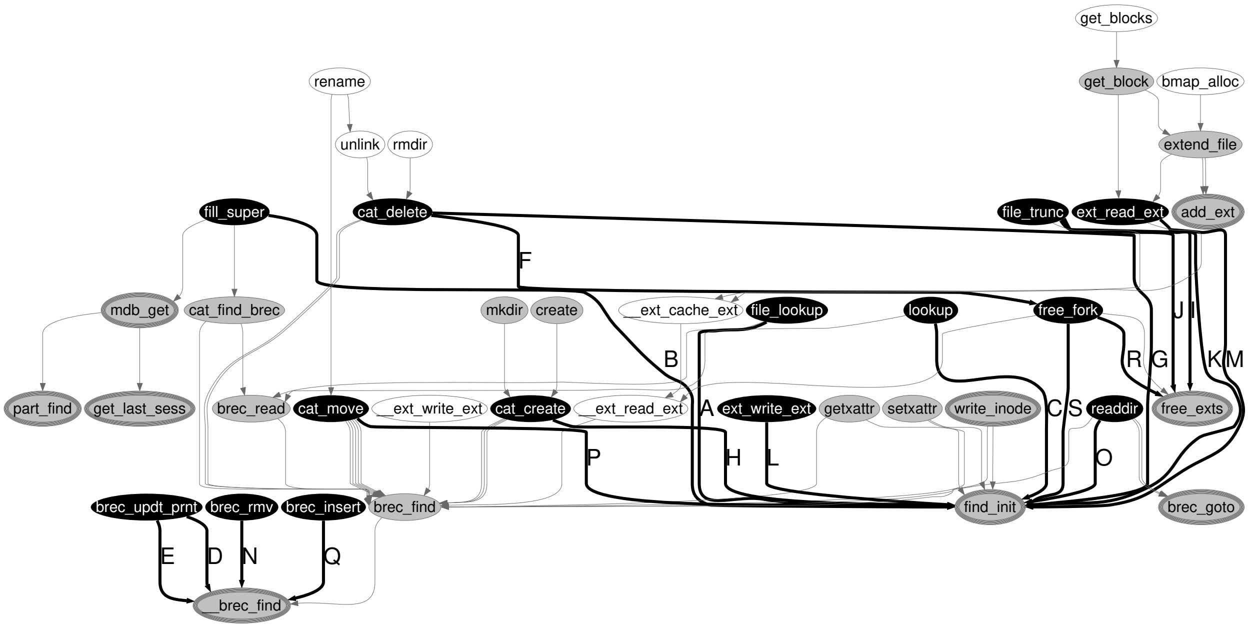
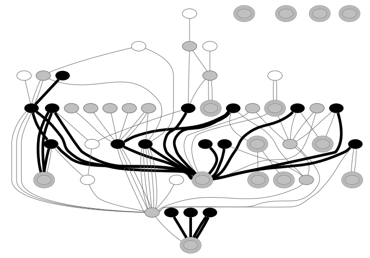
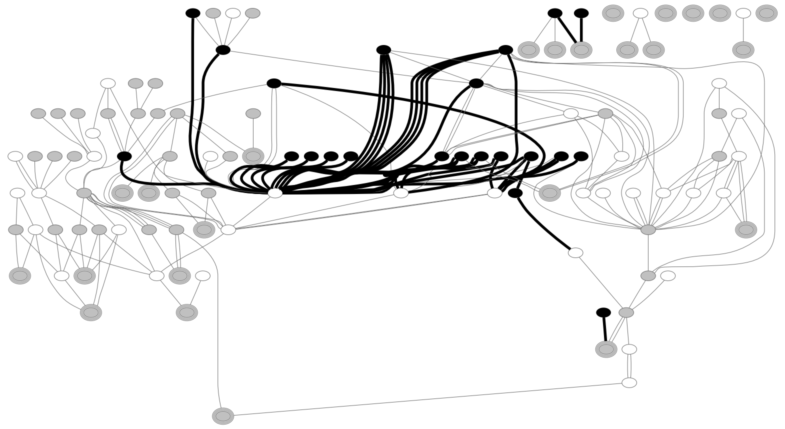


Figure 2: **A Sample of EDP Output.** *The lower figure depicts the EDP output for the HFS file system. Some function names have been shortened to improve readability. As summarized in the upper right legend, a gray node with a thicker border represents a function that generates an error code. The other gray node represents the same thing, but the function also propagates the error code received from its callee. A white node represents a good function, i.e. it either propagates the error code to its caller or if it does not propagate the error code it minimally checks the error code. A black node represents an error-broken termination endpoint, i.e. it is a function that commits the violation of unsaved error codes. The darker and thicker edge coming out from a black node implies a broken error channel (a bad call); an error code actually flows from its callee, but the caller drops the error code. For ease of debugging, each bad call is labeled with a violation number whose detailed information can be found in the upper left violation table. For example, violation #E found in the bottom left corner of the graph is a bad call made by* `brec_updt_prnt` *when calling* `__brec_find`*, which can be located in* `fs/hfs/brec.c` *line 345.*

HFS+ [22 bad / 84 calls, 26%]



ext3 [37 bad / 188 calls, 20%]



ReiserFS [35 bad / 218 calls, 16%]

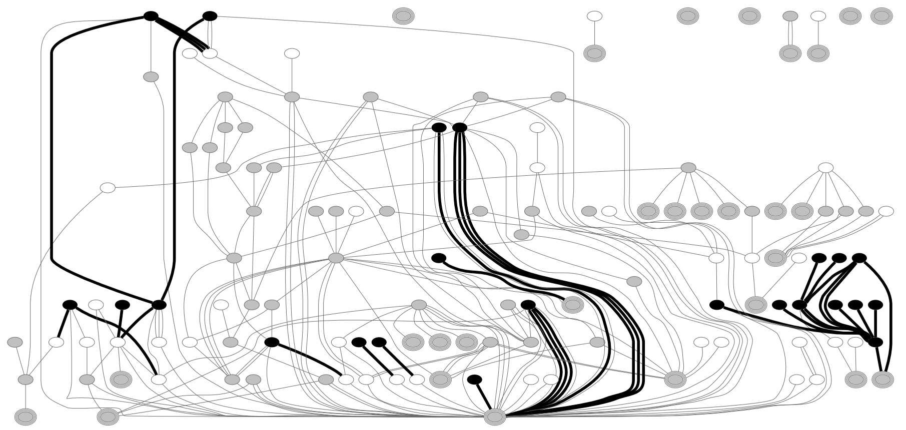
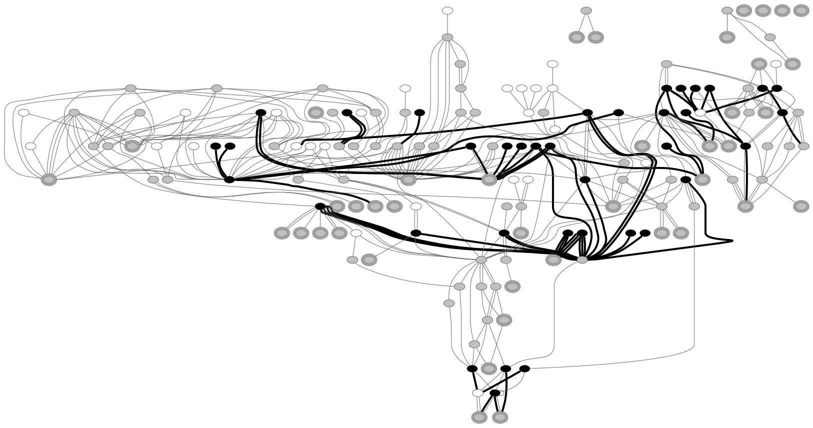
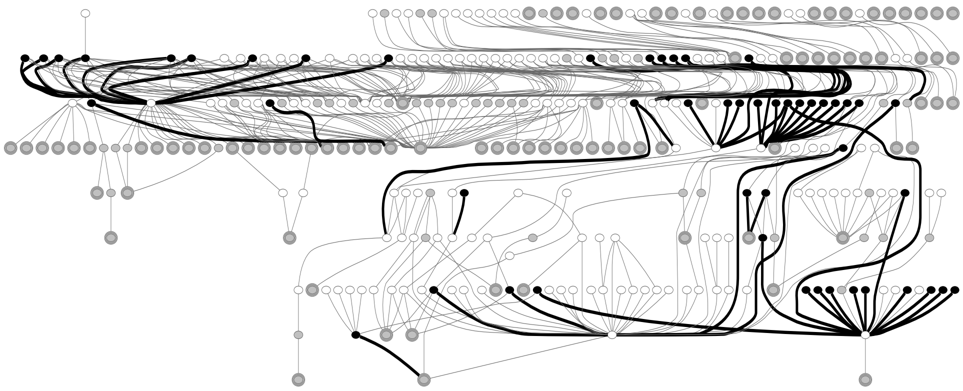


Figure 3: **More Samples of EDP Output.** *The figures illustrate the prevalent problem of incomplete error-propagation across different types of file systems. Details such as function names and violation numbers have been removed. Gray edges represent calls that propagate error codes. Black edges represent bad calls. The number of edges are reported in [X / Y , Z%] format where X and Y represent the number of black and all (gray and black) edges respectively, and Z represents the fraction of X and Y. For more information, please see the legend in Figure 2.*

IBM JFS [61 bad / 340 calls, 18%]



NFS Client [54 bad / 446 calls, 12%]



XFS [105 bad / 1453 calls, 7%]

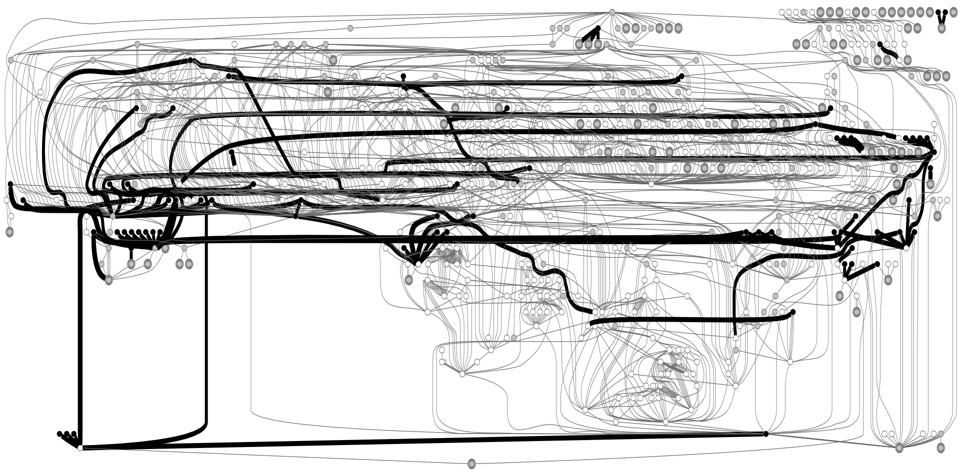


Figure 4: **More Samples of EDP Output (Cont'd).** *Please see caption in Figure 3.*

File Systems

	Bad Calls	EC Calls	Size (Kloc)	Frac (%)	Viol/ Kloc
XFS	**101**	1457	71	6.9	1.4
Virtual FS	**96**	1149	34	8.4	2.9
IBM JFS	**95**	390	17	24.4	5.6
ext3	**80**	362	12	22.1	7.2
NFS Client	**62**	482	18	12.9	3.6
CIFS	**43**	339	21	12.7	2.1
ReiserFS	**42**	399	24	10.5	1.8
Mem. Mgmt.	**40**	351	20	11.4	2.0
Apple HFS+	**25**	98	7	25.5	3.7
JFFS v2	**24**	153	11	15.7	2.2
Apple HFS	**20**	76	5	26.3	4.8
SMB	**19**	196	6	9.7	3.5
ext2	**18**	103	6	17.5	3.3
AFS	**16**	62	7	25.8	2.6
NTFS	**15**	186	18	8.1	0.9
NFS Server	**15**	265	14	5.7	1.2
NCP	**13**	169	5	7.7	2.6
UFS	**12**	44	5	27.3	2.6
JBD	**10**	43	4	23.3	2.6
FAT	**9**	81	4	11.1	2.9
Plan 9	**9**	80	4	11.2	2.4
System V	**7**	30	3	23.3	3.2
JFFS	**7**	56	5	12.5	1.4
UDF	**6**	50	9	12.0	0.7
MSDOS	**5**	39	1	12.8	9.3
VFAT	**4**	39	1	10.3	5.0
Minix	**4**	31	4	12.9	1.2

File Systems (Cont'd)

	Bad Calls	EC Calls	Size (Kloc)	Frac (%)	Viol/ Kloc
FUSE	**4**	48	3	8.3	1.5
Automounter4	**4**	53	2	7.5	2.7
NFS Lockd	**3**	21	4	14.3	0.8
Relayfs	**2**	5	1	40.0	2.7
Partitions	**2**	3	4	66.7	0.6
ISO	**2**	19	3	10.5	0.7
HugeTLB Sup	**2**	10	1	20.0	3.0
Compr. ROM	**2**	3	1	66.7	4.5
ADFS	**2**	30	2	6.7	1.3
sysfs sup.	**1**	29	2	3.4	0.8
romfs sup.	**1**	3	1	33.3	2.4
ramfs sup.	**1**	6	1	16.7	6.0
QNX 4	**1**	8	2	12.5	0.9
proc fs sup.	**1**	44	6	2.3	0.2
OS/2 HPFS	**1**	18	6	5.6	0.2
FreeVxFS	**1**	4	2	25.0	0.7
EFS	**1**	3	1	33.3	1.4
devpts	**1**	2	1	50.0	6.2
Boot FS	**1**	9	1	11.1	1.2
BeOS	**1**	5	3	20.0	0.5
Automounter	**1**	41	2	2.4	1.0
Amiga FFS	**1**	34	3	2.9	0.3
exportfs sup.	**0**	1	1	0.0	0.0
Coda	**0**	149	3	0.0	0.0
Total	**833**	7278	366	–	–
Average	**16.3**	142.7	7.2	**17.0**	**2.4**

Storage Drivers

	Bad Calls	EC Calls	Size (Kloc)	Frac (%)	Viol/ Kloc
SCSI (root)	**123**	628	198	19.6	0.6
IDE (root)	**53**	223	15	23.8	3.5
Block Dev (root)	**39**	195	36	20.0	1.1
Software RAID	**31**	290	32	10.7	1.0
SCSI (aacraid)	**30**	76	7	39.5	4.8
SCSI (lpfc)	**14**	30	16	46.7	0.9
Blk Dev (P-IDE)	**11**	17	8	64.7	1.5
SCSI aic7xxx	**8**	62	37	12.9	0.2
IDE (pci)	**5**	106	12	4.7	0.4

Storage Drivers (Cont'd)

	Bad Calls	EC Calls	Size (Kloc)	Frac (%)	Viol/ Kloc
IDE legacy	**2**	3	3	66.7	0.8
Blk Layer Core	**2**	65	8	3.1	0.3
SCSI megaraid	**1**	30	6	3.3	0.2
Blk Dev (Eth)	**1**	5	2	20.0	0.7
SCSI (sym53c8)	**0**	6	10	0.0	0.0
SCSI (qla2xxx)	**0**	8	49	0.0	0.0
Total	**320**	1744	430	–	–
Average	**21.3**	116.3	28.6	**22.4**	**1.1**

Table 2: **Error-broken channels due to unsaved error codes.** *These tables report the number of bad calls found across all file systems and storage device drivers in Linux 2.6.15.4. In each table, from left to right column we report the name of the subsystem, the number of bad calls, the number of error channels (i.e., the number of calls to functions that propagate error codes), the size of the subsystem, the fraction of bad calls over all error-related calls (ratio of 2nd and 3rd column), and finally the number of violations per Kloc (ratio of 2nd and 4th column). We categorize a directory as a subsystem. Thus, for storage drivers, since different SCSI device drivers exist in the first-level of the* `scsi/` *directory, we put all of them as one subsystem. SCSI device drivers that are located in different directories (e.g.,* `scsi/lpfc/`, `scsi/aacraid/`*) are categorized as different subsystems. The same principle is applied to IDE.*

To show how EDP is useful in finding error propagation bugs, we begin by showing a sample of EDP analysis for a simple file system, Apple HFS. Then, we present our findings on all subsystems that we analyze, and finally discuss false positives.

3.1.1 EDP on Apple HFS

Figure 2 depicts the EDP output when analyzing the propagation of the 34 basic error codes in the Apple HFS file system. There are two important elements that EDP produces in order to ease the debugging process. First, EDP generates an error propagation graph that only includes functions and function calls through which the analyzed error codes propagate. From the graph, one can easily catch all bad calls and functions that make the bad calls. Second, EDP provides a table that presents more detailed information for each bad call (*e.g.*, the location where the bad call is made).

Using the information that EDP provides, we found three major error-handling inconsistencies in HFS. First, 11 out of 14 calls to `find_init` drop the returned error codes. As described earlier in this section, this bug could cause the system to crash or corrupt data. Second, 4 out of 5 total calls to the function `__brec_find` are bad calls (as indicated by the four black edges, E, D, N, and Q, found in the lower left of the graph). The task of this function is to find a record in an HFS node that best matches the given key, and return `ENOENT` (no entry) error code if it fails. The only call that saves this error code is made by the wrapper, `brec_find`. Interestingly, all 18 calls to this wrapper propagate the error code properly (as indicated by all gray edges coming into the function).

Finally, 3 out of 4 calls to `free_exts` do not save the returned error code (labeled R, I, and J). This function traverses a list of extents and locates the extents to be freed. If the extents cannot be found, the function returns `EIO`. More interestingly, the developer wrote a comment "panic?" just before the return statement (maybe in the hope that in this failure case the callers will call panic, which will never happen if the error code is dropped). By and large, we found similar inconsistencies in all the subsystems we analyzed. The fact that the fraction of bad calls over all calls to a function is generally high is intriguing, and will be discussed further in Section 4.3.

3.1.2 EDP on All File Systems and Storage Drivers

Figure 3 and 4 show EDP outputs for six more file systems whose error-propagation graphs represent an interesting sample. EDP outputs for the rest of the file systems can be downloaded from our web site [11]. A small file system such as HFS+ has simple propagation chains, yet bad calls are still made. More complex error propagation can be seen in ext3, ReiserFS, and IBM JFS; within these file systems, error-codes propagate throughout 180 to 340 function calls. The error propagation in NFS is more structured compared to other file systems. Finally, among all file systems we analyze, XFS has the most complex error propagation chain; almost 1500 function calls propagate error-codes. Note that each graph in Figures 3 and 4 was produced by analyzing each file system in isolation (*i.e.*, the graph only shows intra-module but not inter-module calls), yet they already illustrate the complexity of error code propagation in each file system. Manual code inspection would require a tremendous amount of work to find error-propagation bugs.

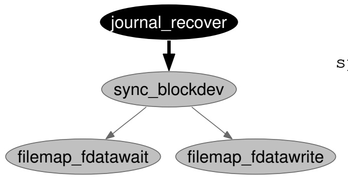
Next, we analyzed the propagation of error codes across all file systems and storage device drivers as a whole. All inter-module calls were connected by our EDP channel constructor, which connects all function pointer calls; hence, we were able to catch inter-module bad calls in addition to intra-module ones. Table 2 summarizes our findings. Note that the number of violations reported is higher than the ones reported in Figures 2, 3, and 4 because we catch more bugs when we analyze each file system in conjunction with other subsystems (*e.g.*, ext3 with the journaling layer, VFS, and the memory management).

Surprisingly, out of 9022 error channels, 1153 (or nearly 13%) constitute bad calls. This appears to be a long-standing problem. We ran a partial analysis in Linux 2.4 (not shown) and found that the magnitude of incomplete error code propagation is essentially the same. In Section 4, we try to dissect the root causes of this problem.

3.1.3 False Positives

It is important to note that while the number of bad calls is high, not all bad calls could cause damage to the system. The primary reason is what we call a *double error code*; some functions expose two or more error codes at the same time, and checking one of the error codes while ignoring the others can still be correct. For example, in the ReiserFS code below, the error code returned from `sync_dirty_buffer` does not have to be saved (line 8) *if and only if* the function performs the check on the second error code (line 9); the buffer must be checked whether it is is up-to-date.

```
 1 // fs/buffer.c
 2 int sync_dirty_buffer (buffer_head* bh) {
 3     ...
 4     return ret; // RETURN ERROR CODE
 5 }
 6 // reiserfs/journal.c
 7 int flush_commit_list() {
 8     sync_dirty_buffer(bh); // UNSAVED EC
 9     if (!buffer_uptodate(bh)) {
10         return -EIO;
11     }
12 }
```




```
journal_recover()
  /* BROKEN CHANNEL */
  sync_blockdev();

sync_blockdev()
  ret = fm_fdatawrite();
  err = fm_fdatawait();
  if(!ret) ret = err;
  /* PROPAGATE EIO */
  return ret;
```

Figure 5: **Silent error in journal recovery.** *In the figure on the left, EDP marks `journal_recover` as a termination endpoint of a broken channel. The code snippet on the right shows that `journal_recover` ignores the `EIO` propagated by `sync_blockdev`.*

To ensure that the number of false positives we report is not overly large, we manually analyze all of the code snippets to check whether a second error code is being checked. Note that this manual process can be automated if we incorporate all types of error codes into EDP. We have found only a total of 39 false positives, which have been excluded from the numbers we report in this paper. Thus, the high numbers in Table 2 provide a hint to a real and critical problem.

3.2 Silent Failures: Manifestations of Unsaved Error Codes

To show that unsaved error codes represent a serious problem that can lead to silent failures, we injected disk block failures in a few cases. As shown in Figure 5, one serious silent failure arises during file system recovery: the journaling block device layer (JBD) does not properly propagate any block write failures, including inode, directory, bitmap, superblock, and other block write failures. EDP unearths these silent failures by pinpointing the `journal_recover` function, which is responsible for file system recovery, as it calls `sync_blockdev` to flush the dirty buffer pages owned by the block device. Unfortunately, `journal_recover` does not save the error code propagated by `sync_blockdev` in the case of block write failures. This is an example where the error code is dropped in the middle of its propagation chain; `sync_blockdev` correctly propagates the `EIO` error codes received from the two function calls it makes.

A similar problem occurs in the NFS server code. From a similar failure injection experiment, we found that the NFS client is not informed when a write failure occurs during a `sync` operation. In the experiment, the client updates old data and then sends a `sync` operation with the data to the NFS server. The NFS server then invokes the `nfsd_dosync` operation, which mainly performs three operations similar to the `sync_blockdev` call above. First, the NFS server writes dirty pages to the disk; second, it writes dirty inodes and the superblock to disk; third, it waits until the ongoing I/O data transfer terminates. All these three operations could return error codes, but the implementation of `nfsd_dosync` does not save any return values. As a result, the NFS client will never notice any disk write failures occurring in the server. Thus, even a careful, error-robust client cannot trust the server to inform it of errors that occur.

In the NFS server code, we might expect that at least one return value would be saved and checked properly. However, no return values are saved, leading one to question whether the returned error codes from the `write` or `sync` operations are correctly handled in general. It could be the case that the developers are not concerned about write failures. We investigate this hypothesis in Section 4.2.

3.3 Unchecked Error Code

Lastly, we report the number of error-broken channels due to a variable that contains an error code not being checked or used in the future. For example, in the IBM JFS code below, `rc` carries an error code propagated from `txCommit` (line 4), but `rc` is never checked.

```
1 // jfs/jfs_txnmgr.c
2 int jfs_sync () {
3     int rc;
4     rc = txCommit(); // UNCHECKED 'rc'
5     // No usage or check of 'rc'
6     // after this line
7 }
```

This analysis can also report false positives due to the double error code problem described previously. In addition, we also find the problem of *overloaded variables* that contribute as false positives. We define a variable to be overloaded if the variable could contain an error code or a data value. For instance, `blknum` in the QNX4 code below is an example of an overloaded variable:

```
1  // qnx4/dir.c
2 int qnx4_readdir () {
3     int blknum;
4     struct buffer_head *bh;
5     blknum = qnx4_block_map();
6     bh = sb_bread (blknum);
7     if (bh == NULL)
8         // error
9 }
```

In this code, `qnx4_block_map` could return an error code (line 5), which is usually a negative value. `sb_bread` takes a block number and returns a buffer head that contains the data for that particular block (line 6). Since a negative block number will lead to a `NULL` buffer head (line 7), the error code stored in `blknum` does not have to be explicitly checked. The developer believes that the other part of the code will catch this error or eventually raise related errors. This practice reduces the accuracy of our static analysis.

	By % Broken		**By Viol/Kloc**	
Rank	FS	Frac.	FS	Viol/Kloc
1	IBM JFS	24.4	ext3	7.2
2	ext3	22.1	IBM JFS	5.6
3	JFFS v2	15.7	NFS Client	3.6
4	NFS Client	12.9	VFS	2.9
5	CIFS	12.7	JFFS v2	2.2
6	MemMgmt	11.4	CIFS	2.1
7	ReiserFS	10.5	MemMgmt	2.0
8	VFS	8.4	ReiserFS	1.8
9	NTFS	8.1	XFS	1.4
10	XFS	6.9	NFS Server	1.2

Table 3: **Least Robust File Systems.** *The table shows the ten least robust file systems using two ranking systems. In the first ranking system, file system robustness is ranked based on the fraction of broken channels over all error channels (the 5th column of Table 2). The second ranking system sorts file systems based on the number of broken channels found in every Kloc (the 6th column of Table 2).*

Since the number of unchecked error code reports is small, we were able to remove the false positives and find a total of 3 and 2 unchecked error codes in file systems and storage drivers, respectively, that could lead to silent failures.

4 Analysis of Results

In the following sections, we present five analyses whereby we try to uncover the root causes and impact of incomplete error propagation. Since the number of unchecked and overwritten error codes is small, we only consider unsaved error codes (bad calls) in our analyses; thus we use "bad calls" and "broken channels" interchangeably from now on. First, we made a correlation between robustness and complexity. Second, we analyzed whether file systems and storage device drivers give different treatment to errors occurring in I/O read vs. I/O write operations. From that analysis we find that many write errors are neglected; hence we perform the next study in which we try to answer whether ignored errors are corner-case mistakes or intentional choices. In the final two analyses, we analyze whether chained error propagation and inter-module calls play major parts in causing incorrect error propagation.

4.1 Complexity and Robustness

In our first analysis, we would like to correlate the number of mistakes in a subsystem with the complexity of that subsystem. For file systems, XFS with 71 Kloc has more mistakes than other, smaller file systems. However, it is not necessary that XFS is seen as the least robust file system. Table 3 sorts the robustness of each file system based on two rankings. In both rankings, we only account file systems that are at least 10 Kloc in size with at least 50 error-related calls, *i.e.* we only consider "complex" file systems.

A noteworthy observation is that ext3 and IBM JFS are ranked as the two least robust file systems. This fact affirms our earlier findings on the robustness of ext3 and IBM JFS [20]. In this prior work, we found that ext3 and IBM JFS are inconsistent in dealing with different kinds of disk failures. Thus, it might be the case that these inconsistent policies correlate with inconsistent error propagation.

Among storage device drivers, it is interesting to compare the robustness of the SCSI and IDE subsystems. If we compare SCSI and IDE subsystems using the first ranking system, SCSI and IDE are almost comparable (21% vs. 18%). However, if we compare them based on the second ranking system, then the SCSI subsystem is almost four times more robust than IDE (0.6 vs. 2.1 errors/Kloc). Nevertheless it seems the case that SCSI utilizes basic error codes much more than IDE does.

When the robustness of storage drivers and file systems is compared using the first ranking, on average storage drivers are less robust compared to file systems (22% vs. 17%, as reported in the last rows of Table 2). On the other hand, in the second ranking system, storage drivers are more robust compared to file systems (1.1 vs. 2.4 mistakes/Kloc). From our point of view, the first ranking system is more valid because a subsystem could be comprised of submodules that do not necessarily use error codes; what is more important is the number of bad calls in the population of all error-related calls.

4.2 Neglected Write Errors

As mentioned in Section 3.2, we have observed that error codes propagated in `write` or `sync` operations are often ignored. Thus, we investigate how many write errors are neglected compared to read errors. This study is motivated by our findings in that section as well as by our earlier findings where we found that at least for ext3, read failures are detected, but write errors are often ignored [20].

To perform this study, we filter out calls that do not relate to read and write operations. Since it is impractical to do that manually, we use a simple string comparison to mark calls that are relevant to our analysis. That is we only take a caller→callee pair where the callee contains the string `read`, `write`, `sync`, or `wait`. We include `wait`-type calls because in many cases `wait`-type callees (*e.g.*, `filemap_datawait`) represent waiting for one or more I/O operations and could return error information on the operation. Thus, in our study, `write`-, `sync`-, and `wait`-type calls are categorized as write operations.

Callee Type	Bad Calls	EC Calls	**Frac. (%)**
Read*	26	603	**4.3**
Sync	70	236	**29.7**
Wait	27	70	**38.6**
Write	80	598	**13.4**
Sync+Wait+Write	177	904	**19.6**
Specific Callee			
`filemap_fdatawait`	22	29	**75.9**
`filemap_fdatawrite`	30	47	**63.8**
`sync_blockdev`	15	21	**71.4**

Table 4: **Neglected write errors in file system code.** *The table shows that read errors are handled more correctly than write errors. The upper table shows the fraction of bad calls over four category of calls: read, sync, wait, and write. The later three can be categorized as a write operation. The lower table shows neglected write errors for three specific functions. The 29 (*) violated read calls are all related to readahead and asynchronous read; in other words, all error codes returned in synchronous reads are being saved and checked.*

The upper half of Table 4 reports our findings. The last column shows how often errors are ignored in the file system code. Interestingly, file systems have a tendency to correctly handle error codes propagated from `read`-type calls, but not those from `write`-type calls (4.3% vs. 19.6%). The 29 (4.3%) unsaved read error codes are all found in readahead operations in the memory management subsystem; it might be acceptable to ignore prefetch read errors because such reads can be reissued in the future whenever the page is actually read.

As discussed in Section 3.1, a function could return more than one error code at the same time, and checking only one of them suffices. However, if we know that a certain function only returns a single error code and yet the caller does not save the return value properly, then we would know that such call is really a flaw. To find real flaws in the file system code, we examined three important functions that we know only return single error codes: `sync_blockdev`, `filemap_fdatawrite`, and `filemap_fdatawait`. A file system that does not check the returned error codes from these functions would obviously let failures go unnoticed in the upper layers.

The lower half of Table 4 reports our findings. Many error codes returned from the three methods are simply not saved ($>$ 63% in all cases). Two conclusions might be drawn from this observation. First, this could suggest that higher-level recovery code does not exist (since if it exists, it will not be invoked due to the broken error channel), or it could be the case that errors are intentionally neglected. We consider this second possibility in greater detail in the next section.

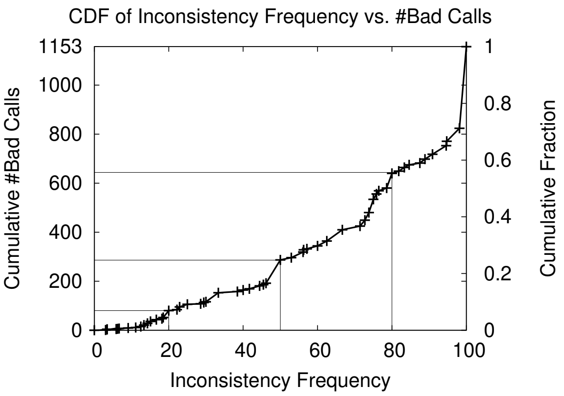


Figure 6: **Inconsistent calls frequency.** *The figure shows that inconsistent calls are not corner-case bugs. The x-axis represents the inconsistent-call frequency of a function. x=20% means that there is one bad call out of five total calls; x=80% means that there are four bad calls out of five total calls. The left y-axis counts the cumulative number of bad calls. For example, below the 20% mark, there are 80 bad calls that have an inconsistent-call frequency of less than 20%. As reported in Table 2, there exist a total of 1153 bad calls. The right y-axis shows the cumulative fraction of bad calls over the 1153 bad calls.*

4.3 Inconsistent Calls: Corner Case or Majority?

In this section, we consider the nature of *inconsistent* calls. For example, we found that 1 out of 33 calls to `ide_setup_pci_device` does not save the return value. One would probably consider this single call as an inconsistent implementation because the majority of the calls to that function save the return value. On the other hand, we also found that 53 out of 54 calls to `unregister_filesystem` do not save the return error codes. Assuming that most kernel developers are essentially competent, this suggests that it may actually be safe to not check the error code returned from this particular function.

To quantify inconsistent calls, we define the *inconsistent call frequency* of a function as the ratio of bad calls over all error-related calls to the function, and correlate this frequency with the number of bad calls to the function. For example, the inconsistent call frequencies for `ide_setup_pci_blockdev` and `unregister_filesystem` are 3% (1/33) and 98% (53/54) respectively and the numbers of bad calls are 1 and 53 respectively.

Figure 6 plots the cumulative distribution function of this behavior. The graph could be seen as a means to prioritize which bad calls to fix first. Bad calls that fall below the 20% mark could be treated as *corner cases*, *i.e.* we should be suspicious on one bad call in the midst of four good calls to the same function. On the other hand, bad calls that fall above the 80% mark could hint that either different developers make the same mistake and ignore it, or it is probably safe to make such a mistake.

One perplexing phenomenon visible in the graph is that around 871 bad calls fall above the 50% mark. In other words, they cannot be considered as corner-case bugs; the developers might be aware of these bad calls, but probably just ignore them. One thing we have learned from our recent work on file system code is that if a file system does not know how to recover from a failure, it has the tendency to just ignore the error code. For example, ext3 ignores write failures during checkpointing simply because it has no recovery mechanism (*e.g.*, chained transactions [12]) to deal with such failures. Thus, we suspect that there are deeper design shortcomings behind poor error code handling; error code mismanagement may be as much symptom as disease.

Our analysis is similar to the work of Engler *et al.* on findings bugs automatically [8]. In their work, they use existing implementation to imply beliefs and facts. Applying their analysis to our case, the bad calls that fall above the 80% mark might be considered as good calls. However, since we are analyzing the specific problem of error propagation, we use that semantic knowledge and demand a discipline that promotes checking an error code in all circumstances, rather than one that follows majority rules.

4.4 Characteristics of Error Channels

Finally, we study whether the characteristic of an error channel has an impact on the robustness of error code propagation in that channel. In particular, we explore two characteristics of error channels: one based on the error propagation distance and one based on the location distance (inter- vs. intra-file calls).

With the first characteristic, we would like to find out whether error codes are lost near the generation endpoint or somewhere in the middle of the propagation chain. We distinguish two calls: direct-error and propagate-error calls. In a *direct-error call*, the callee is an error-generation endpoint. In a *propagate-error call*, the callee is not a generation endpoint; rather it is a function that propagates an error code from one of the functions that it calls, *i.e.* it is a function in the middle of the propagation chain. Next, we define a *bad* direct-error (or propagate-error) call as a direct-error (or propagate-error) call that does not save the returned error code.

Initially, we assumed that the frequency of bad propagate-error calls would be higher than that of bad direct-error calls; we assumed error codes tend to be dropped in the middle of the chain rather than near the generation endpoint. It turns out that the number of bad direct-error and propagate-error calls are similar for file system code but the other way around for storage driver code. In particular, for file systems, the ratio of bad over all direct-error calls is 10%, and the ratio of bad over all propagate-error calls is 14%. For storage drivers, they are 20% and 15% respectively.

	Bad Calls	EC Calls	**Frac. (%)**
File Systems			
Inter-module	307	1944	**15.8**
Inter-file	367	2786	**13.2**
Intra-file	159	2548	**6.2**
Storage Drivers			
Inter-module	48	199	**24.1**
Inter-file	92	495	**18.6**
Intra-file	180	1050	**17.1**

Table 5: **Calls based on location distance.** *The table shows that the fraction of bad calls in inter-module calls is higher than the one in inter-file calls. Similarly, inter-file calls are less robust than intra-file calls. Note that "inter-file" refers to cross-file calls within the same module. Inter-file calls across different modules are categorized as inter-module.*

Lastly, in the second characteristic, we categorized calls based on the location distance between a caller and a callee. In particular, we distinguish three calls: inter-module, inter-file (but within the same module), and intra-file calls. Table 5 reports that intra-file calls are more robust than inter-file calls, and inter-file calls are more robust than intra-file calls. For example, out of 1944 inter-module calls in which error codes propagate in file system, 307 (16%) of them are bad calls. However, out of 2786 inter-file calls within the same module, there are only 367 (13%) bad calls. Intra-file calls only exhibit 6% bad calls. The same pattern occurs in storage device drivers. Thus, we conclude that the location distance between the caller and the callee plays a role in the robustness of the call.

5 Future Work

In this section, we discuss some of the issues we previously deferred regarding how to build complete and accurate static error propagation analysis. In general, we plan to refine our static analysis with the intention of uncovering more violations within the file and storage system stack.

5.1 Overwritten Error Codes

In this paper, we examined broken channels that are caused by unsaved and unchecked error codes; broken channels can also be caused by *overwritten error codes*, in which the container that holds the error code is overwritten with another value before the previous error is checked. For example, the CIFS code below overwrites (line 6) the previous error code received from another call (line 4).

```
1 // cifs/transport.c
2 int SendReceive () {
3     int rc;
4     rc = cifs_sign_smb(); // PROPAGATE E.C.
5     ... // No use of 'rc' here
6     rc = smb_send(); // OVERWRITTEN
7 }
```

Currently, EDP detects overwritten error codes, but reports too many false positives to be useful. We are in the process of fine-tuning EDP so that it provides more accurate output. The biggest problem we have encountered is due to the nature of the error hierarchy: in many cases, a less critical error code is overwritten with a more critical one. For example, in the memory management code below, when first encountering a page error, the error code is set to `EIO` (line 6). Later, the function checks whether the flags of a `map` structure carry a no-space error code (line 8). If so, the `EIO` error code is overwritten (line 9) with a new error code `ENOSPC`.

```
 1 // mm/filemap.c
 2 int wait_on_page_writeback_range (pg, map) {
 3     int ret = 0;
 4     ...
 5     if (PageError(pg))
 6         ret = -EIO;
 7     ...
 8     if (test_bit(AS_ENOSPC, &map->flags))
 9         ret = -ENOSPC;
10     if (test_bit (AS_EIO, &map->flags))
11         ret = -EIO;
12     return ret;
13 }
```

Manually inspecting the results obtained from EDP, we have identified five real cases of overwritten error codes: one each in AFS and FAT, and three in CIFS. We believe we will find more cases as we fine-tune our analysis of overwritten error codes.

5.2 Error Transformation

Our current EDP analysis focuses on the basic error codes that are stored and propagated mainly in integer containers. However, file and storage systems also use other specific error codes stored in complex structures that can be mapped to other error codes in new error containers; we call this issue *error transformation*. For example, the block layer clears the `uptodate` bit stored in a buffer structure to signal I/O failure, while the VFS layer simply uses generic error codes such as `EIO` and `EROFS`. We have observed a path where an error container changes five times, involving four different types of containers. A complete EDP analysis must recognize all transformations. With a more complete analysis, we expect to see even more violations.

5.3 Asynchronous Error Channels

Finally, we plan to expand our definition of error channels to include *asynchronous paths*. We briefly describe two examples of asynchronous paths and their complexities. First, when a lower layer interrupts an upper one to notify it of the completion of an I/O, the low-level I/O error code is usually stored in a structure located in the heap; the receiver of the interrupt should grab the structure and check the error it carries, but tracking this propagation through the heap is not straightforward. Another example occurs during journaling: a journal daemon is woken up somewhere in the `fsync()` path and propagates a journal error code via a global journal state. When we consider asynchronous error channels, we also expect the number of violations to increase.

6 Related Work

Previous work has used static techniques to understand variety of problems in software systems. For example, Meta-level compilation (MC) [7, 8] enables a programmer to write simple, system-specific compiler extensions to automatically check software for rule violations. With their work, one can find broken channels by specifying a rule such as "a returned variable must be checked." Compared to their work, ours presents more information on how error propagates and convert it into graphical output for ease of analysis and debugging.

Another related project is FiSC [32], which uses the model-checking tool CMC [17] to find file system errors in the Linux kernel. Every time the file system under test transitions to a new state, FiSC runs a series of invariant checkers looking for file system errors. If an error is found, one can trace back the states and diagnose the sequence of actions that lead to the error. One aspect of our work that is similar to FiSC is that we unearth silent failures. For example, FiSC detects a bug where a system call returns success after it calls a resource allocation routine that fails, *e.g.* due to memory failures.

In recent work, Johansson analyzes run-time error propagation based on interface observations [14]. Specifically, an error is injected at the OS-driver interface by changing the value of a data parameter. By observing the application-OS interface after the error injection, they reveal whether errors occurring in the OS environment (device drivers) will propagate through the OS and affect applications. This run-time technique is complementary to our work, especially to uncover the eventual bad effects of error-broken channels.

Solving the error propagation problem is also similar to solving the problem of unchecked exceptions. Sacramento *et al.* found too many unchecked exceptions, thus doubting programmers' assurances in documenting exceptions [25]. Nevertheless, since using exceptions is not a kernel programming style, at least at the current state, solutions to the problem of unchecked exceptions might not be applicable to kernel code. Only recently is there an effort in employing exceptions in OS code [3].

Our tool is also similar to Jex [24]. While Jex is a static analysis tool that determines exception flow information in Java programs, our tool determines the error code flow information within the Linux kernel.

To fix the incomplete error propagation problem, developers could simply adopt a simple set-check-use methodology [2]. However, it is interesting to see that this simple practice has not been applied thoroughly in file systems and storage device drivers. As mentioned in Section 4.3, we suspect that there are deeper design shortcomings behind poor error code handling.

7 Conclusion

In this paper, we have analyzed the file and storage systems in Linux 2.6 and found that error codes are not consistently propagated. We conclude by reprinting some developer comments we found near some problematic cases:

> CIFS – *"Not much we can do if it fails anyway, ignore rc."*
>
> CIFS – *"Should we pass any errors back?"*
>
> ext3 – *"Error, skip block and hope for the best."*
>
> ext3 – *"There's no way of reporting error returned from ext3_mark_inode_dirty() to userspace. So ignore it."*
>
> IBM JFS – *"Note: todo: log error handler."*
>
> ReiserFS – *"We can't do anything about an error here."*
>
> XFS – *"Just ignore errors at this point. There is nothing we can do except to try to keep going."*
>
> SCSI – *"Retval ignored?"*
>
> SCSI – *"Todo: handle failure."*

These comments from developers indicate part of the problem: even when the developers are aware they are not properly propagating an error, they do not know how to implement the correct response. Given static analysis tools to identify the source of bugs (such as EDP), developers may still not be able to fix all bugs in a straightforward manner.

Due to these observations, we believe it is thus time to rethink how failures are managed in large systems. Preaching that developers follow error handling conventions and hoping the resulting systems work as desired seems naive at best. New approaches to error detection, propagation, and recovery are needed; in the future, we plan to explore a range of error architectures, hoping to find methods that increase the level of robustness in the storage systems upon which we all rely.

Acknowledgments

We thank the members of the ADSL research group for their insightful comments. We would also like to thank Geoff Kuenning (our shepherd) and the anonymous reviewers for their excellent feedback and comments, many of which have greatly improved this paper. The second author wishes to thank the National Council on Science and Technology of Mexico and the Secretariat of Public Education for their financial support.

This work is supported by the National Science Foundation under the following grants: CCF-0621487, CNS-0509474, CCR-0133456, as well as by generous donations from Network Appliance and Sun Microsystems.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or other institutions.

References

[1] Steve Best. JFS Overview. www.ibm.com/developer works/library/l-jfs.html, 2000.

[2] Michael W. Bigrigg and Jacob J. Vos. The Set-Check-Use Methodology for Detecting Error Propagation Failures in I/O Routines. In *WDB '02*, Washington, DC, June 2002.

[3] Bruno Cabral and Paulo Marques. Making Exception Handling Work. In *HotDep II*, Seattle, Washington, Nov 2006.

[4] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. Microreboot – A Technique for Cheap Recovery. In *OSDI '04*, pages 31–44, San Francisco, CA, December 2004.

[5] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX '98 Security*, San Antonio, TX, January 1998.

[6] Daniel Ellard and James Megquier. DISP: Practical, Efficient, Secure, and Faul-Tolerant Distributed Data Storage. *ACM Transactions on Storage (TOS)*, 1(1):71–94, Feb 2005.

[7] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions . In *OSDI '00*, San Diego, CA, October 2000.

[8] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *SOSP '01*, pages 57–72, Banff, Canada, October 2001.

[9] Dawson R. Engler and Daniel Dunbar. Under-constrained execution: making automatic code destruction easy and scalable. In *ISSTA '07*, London, United Kingdom, July 2007.

[10] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed Automated Random Testing. In *PLDI '05*, Chicago, IL, June 2005.

[11] Haryadi S. Gunawi. EDP Output for All File Systems. www.cs.wisc.edu/adsl/Publications/eio-fast08/readme.html.

[12] Haryadi S. Gunawi, Vijayan Prabhakaran, Swetha Krishnan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Improving File System Reliability with I/O Shepherding. In *SOSP '07*, pages 283–296, Stevenson, Washington, October 2007.

[13] Michael Hind. Pointer Analysis: Haven't We Solved This Problem Yet? In *PASTE '01*, Snowbird, Utah, June 2001.

[14] Andreas Johansson and Neeraj Suri. Error Propagation Profiling of Operating Systems . In *DSN '05*, Yokohoma, Japan, June 2005.

[15] George Kola, Tevfik Kosar, and Miron Livny. Faults in Large Distributed Systems and What We Can Do About Them. In *Euro-Par*, August 2005.

[16] Philip Koopman and John DeVale. Comparing the Robustness of POSIX Operating Systems. In *FTCS-29*, Madison, Wisconsin, June 1999.

[17] Madanlal Musuvathi, David Y.W. Park, Andy Chou, Dawson R. Engler, and David L. Dill. CMC: A Pragmatic Approach to Model Checking Real Code. In *OSDI '02*, Boston, MA, December 2002.

[18] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. CCured: Type-Safe Retrofitting of Legacy Software. *ACM Transactions on Programming Languages and Systems*, 27(3), May 2005.

[19] George C. Necula, Scott McPeak, S. P. Rahul, and Westley Weimer. Cil: An infrastructure for c program analysis and transformation. In *CC '02*, pages 213–228, April 2002.

[20] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. IRON File Systems. In *SOSP '05*, pages 206–220, Brighton, UK, October 2005.

[21] Feng Qin, Shan Lu, and Yuanyuan Zhou. Exploiting ECC-memory for detecting memory leaks and memory corruption during production runs. In *HPCA-11*, San Francisco, California, February 2005.

[22] Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. Rx: Treating Bugs As Allergies. In *SOSP '05*, Brighton, UK, October 2005.

[23] Hans Reiser. ReiserFS. www.namesys.com, 2004.

[24] Martin P. Robillard and Gail C. Murphy. Designing Robust Java Programs with Exceptions. In *FSE '00*, San Diego, CA, November 2000.

[25] Paulo Sacramento, Bruno Cabral, and Paulo Marques. Unchecked Exceptions: Can the Programmer be Trusted to Document Exceptions? In *IVNET '06*, Florianopolis, Brazil, October 2006.

[26] Stelios Sidiroglou, Michael E. Locasto, Stephen W. Boyd, and Angelos D. Keromytis. Building a Reactive Immune System for Software Services. In *USENIX '05*, Anaheim, CA, April 2005.

[27] David A. Solomon. *Inside Windows NT*. Microsoft Programming Series. Microsoft Press, 2nd edition, May 1998.

[28] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the Reliability of Commodity Operating Systems. In *SOSP '03*, Bolton Landing, NY, October 2003.

[29] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Recovering device drivers. In *OSDI '04*, pages 1–16, San Francisco, CA, December 2004.

[30] Douglas Thain and Miron Livny. Error Scope on a Computational Grid: Theory and Practice. In *HPDC 11*, Edinburgh, Scotland, July 2002.

[31] Stephen C. Tweedie. Journaling the Linux ext2fs File System. In *The Fourth Annual Linux Expo*, Durham, North Carolina, May 1998.

[32] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using Model Checking to Find Serious File System Errors. In *OSDI '04*, San Francisco, CA, December 2004.