Defective Error/Pointer Interactions in the Linux Kernel

Cindy Rubio-Gonzalez
Ben Liblit

Technical Report #1687

March 2011

Defective Error/Pointer Interactions in the Linux Kernel

Cindy Rubio-González
crubio@cs.wisc.edu

Ben Liblit
liblit@cs.wisc.edu

Computer Sciences Department, University of Wisconsin–Madison
1210 W Dayton St., Madison, Wisconsin, United States of America

ABSTRACT

Linux run-time errors are represented by integer values referred to as *error codes*. These values propagate across long function-call chains before being handled. As these error codes propagate, they are often temporarily or permanently encoded into pointer values. Error-valued pointers are not valid memory addresses, and therefore require special care by programmers. Misuse of pointer variables that store error codes can lead to serious problems such as system crashes, data corruption, unexpected results, etc. We use static program analysis to find three classes of bugs relating error-valued pointers: bad dereferences, bad pointer arithmetic, and bad overwrites. Our tool finds 57 true bugs among 52 different Linux file system implementations, the virtual file system (VFS), the memory management module (mm), and 4 drivers.

1. INTRODUCTION

Most Linux run-time errors are represented as simple integer codes. Each integer code corresponds to a different kind of error, and macros give these mnemonic names. For example, the integer code 12 represents the out-of-memory error, defined by the macro ENOMEM. Linux defines a set of 34 basic error codes, which are negated by convention. Because Linux is written in C, there is no mechanism to throw or raise error codes as exceptions. Instead, error codes are propagated through function return values and variable assignments. This is also known as the return-code idiom [9], and is widely used in large C programs, including other operating systems.

Due to the size and complexity of systems such as Linux, error propagation can quickly become very complex, effort-demanding and error-prone. Prior work [6, 20, 21] has described static analyses that determine how error codes are propagated. These analyses have identified hundreds of dropped/overwritten errors and documentation flaws in numerous Linux file systems.

As they propagate, error codes may be transformed into other error codes or cast into other types such as pointers. This makes it possible for error codes to be stored in pointer variables, bringing up additional problems particular to pointer values. For example, an error-valued pointer should never be dereferenced. Improper use of pointer values in systems code can have serious consequences such as system crashes, data corruption and unexpected results. Prior work [6, 20, 21] did not take error/pointer interaction into consideration, making it impossible to detect such pointer-related problems.

The contributions of this paper are as follows:

- We characterize error transformation in the Linux kernel (Section 2) and show how these transformations can lead to bugs due to defective error/pointer interaction (Section 3).
- We extend an error-propagation analysis to properly model the effects of error/pointer transformation on Linux error propagation (Section 4).
- We apply this analysis to find program points at which error-valued pointers are dereferenced, used in pointer arithmetic, or overwritten (Section 5).
- We report results for 52 different Linux file system implementations, the virtual file system (VFS), the memory management module (mm) and 4 drivers (Section 6).

2. ERROR TRANSFORMATION IN THE LINUX KERNEL

Error transformation refers to changes in error representation as errors propagate across software layers. Integer error codes may be cast into other types or be transformed into different error codes. In particular, integer error codes are often cast to pointer values. To be clear, these are not pointers that refer to the locations of error codes. Rather, the numeric value of the pointer itself is actually a small integer error code rather than a proper memory address. As offensive as this may seem from a type-system perspective, it is nevertheless a well-accepted practice found throughout the Linux kernel. Linux introduces two functions to convert (cast) error codes from integers to pointers and vice versa: ERR_PTR and PTR_ERR. Also, the Boolean function IS_ERR is used to determine whether a pointer variable contains an error code.

Figure 1 shows an example of integer-to-pointer error transformation. Function open_xa_dir returns a pointer value. Variable xaroot may receive an error-valued pointer from a function call on line 4. Function IS_ERR on line 5 tests the return value. If it is an error, the error-valued pointer is returned. Also, function ERR_PTR is called on lines 15 and 20 to transform an integer error code into a pointer.

Figure 2 illustrates the opposite transformation, from pointer to integer. Function reiserfs_listxattr returns an integer value. Error constants are returned on lines 6 and 9. Also, variable dir may receive an error-valued pointer from a call to function open_xa_dir (shown in Figure 1). If it is an error, then function PTR_ERR transforms the error from a pointer to an integer on line 14.

The preceding examples, though simplified for this paper, already illustrate how tricky it can be to follow error flows. Error codes are often propagated through long call chains and transformed several times before being handled. This makes error tracking quite challenging in large systems such as the Linux kernel. Thus, supporting error transformation is crucial to building a more complete understanding of error-code propagation and how the system recovers from run-time errors.

```
static struct dentry *open_xa_dir(...) {
  struct dentry *xaroot;
  ...
  xaroot = ...;   // may receive error
  if (IS_ERR(xaroot))
    return xaroot;

  if (...) {
    int err;
    if (...) {

      err = ...;
      if (err) {
        ...
        return ERR_PTR(err);
      }
    }
    if (...) {
      ...
      return ERR_PTR(-ENODATA);
    }
  }

  ...
  return ...;
}
```

Figure 1: Example of int-to-pointer transformation

```
int reiserfs_listxattr(...) {
  struct dentry *dir;
  int err = 0;

  if (...)
    return -EINVAL;

  if (...)
    return -EOPNOTSUPP;

  dir = open_xa_dir(...);  // may receive error
  ...
  if (IS_ERR(dir)) {
    err = PTR_ERR(dir);
    if (err == −ENODATA) {
      ...
      err = ...;
    }
    goto out;
  }
  ...

 out:
  ...
  return err;
}
```

Figure 2: Example of pointer-to-int transformation

```
1  static int fill_super(...) {
2    int error;
3    inode *root = ...;
4    ...
5    error = cnode_make(&root, ...);  // error and root receive error
6    if ( error || !root ) {
7      printk("... error %d", error);
8      goto error;
9    }
10   ...
11  error:
12   ...
13   if (root)  // root contains an error
14    iput(root);
15   ...
16 }
17
18 void iput(inode *inode) {
19   if (inode) {
20   BUG_ON(inode->i_state == ...);  // bad pointer dereference
21     if (...))
22       iput_final(inode);
23   }
24 }
```

Figure 3: Example of a bad pointer dereference. The Coda file system propagates an error-valued pointer which is dereferenced by the VFS (function iput).

3. ERROR-VALUED POINTER BUGS

We concentrate on finding bugs due to the improper use of error-holding pointers. The following subsections present three kinds of pointer-related bugs: bad dereferences, bad pointer arithmetic, and bad overwrites.

3.1 Bad Pointer Dereferences

We say that a bad pointer dereference occurs when an error-valued pointer is dereferenced, since an error value is not a valid memory address. Figure 3 shows an example. Function fill_super in the Coda file system calls function cnode_make on line 5, which may return the integer error code ENOMEM while storing the same error code in the pointer variable root. The error is logged on line 7. If root is not NULL (line 13), then function iput in the VFS is invoked with variable root as parameter. This function dereferences the error-valued pointer parameter inode on line 20.

Our goal is to find the program locations at which these bad pointer dereferences may occur. We identify the program points at which pointer variables are dereferenced, i.e., program points where the indirection ($*$) or arrow ($->$) operators are applied. Let us assume for now that we are able to retrieve the set of values each pointer variable may contain at any location l in the program. Thus, at each dereference of variable v, we retrieve the associated set of values $\mathscr{N}_l$, which corresponds to the set of values v may contain right before the dereference at l. Let $\mathscr{E}$ be the finite set of all error constants. Let OK be a single value not in $\mathscr{E}$ that represents all non-error constants. Let $\mathscr{C} = OK \cup \mathscr{E}$ be the set of all constants. Then $\mathscr{N}_l \subseteq \mathscr{C}$, and the set of error codes that variable v contains before the dereference is given by $\mathscr{N}_l \cap \mathscr{E}$. If $\mathscr{N}_l \cap \mathscr{E} \neq \emptyset$, then we report the bad pointer dereference.

```
#define virt_to_page(addr) (mem_map + (((unsigned long)(addr)-
PAGE_OFFSET) » PAGE_SHIFT))  // addr has error

void kfree(const void *x) {  // may be passed an error
  struct page *page;
  ...
  page = virt_to_head_page(x);  // passing error
  ...
}

struct page *virt_to_head_page(const void *x) {
  struct page *page = virt_to_page(x);  // macro defined in line 1
  return ...;
}
```

Figure 4: Bad pointer arithmetic found in the mm

3.2 Bad Pointer Arithmetic

Although error codes are stored in integer and pointer variables, these codes are conceptually atomic symbols, not numbers. Error-valued pointers should never be used to perform pointer arithmetic. For example, incrementing or decrementing a pointer variable that holds an error code will not result in a valid memory address. Similarly, subtracting two pointer variables that may contain error values will not yield the number of elements between both pointers as it would with valid addresses. Figure 4 shows an example of bad pointer arithmetic found in the mm. Callers of function kfree (line 3) may pass in a pointer variable that contains the error code ENOMEM, now in variable x. The variable is further passed to function virt_to_head_page when it is invoked on line 6. Finally, this function uses x to perform some pointer arithmetic on line 11, without first checking for any errors. Note that virt_to_page is defined on line 1.

We aim to identify the program points at which such bad pointer arithmetic instances occur. We find the program locations at which pointer arithmetic operators addition (+), subtraction (−), increment (++), or decrement (−−) are used. For each variable operand v in a given pointer arithmetic operation at program location l, we retrieve the set of values $\mathcal{N}_l$ that v may contain right before the operation. We report a problem if $\mathcal{N}_l \cap \mathcal{E} \neq \emptyset$ for any operand v.

3.3 Bad Overwrites

Bad overwrites occur when error values are overwritten before they have been properly noticed and handled by recovery/reporting code. Our goal is to find bad overwrites of error-valued pointers or error values stored in pointed-to variables. The later can occur either when the variable is assigned through a pointer dereference or when the pointer variable is assigned a different value, which may or may not be a valid address value.

In general, bad overwrites are more challenging to identify than those bugs described in previous sections. The reason is that most error-valued overwrites are safe, whereas (for example) error-valued pointer dereferences are always a serious problem. Also, the consequences of a bad overwrite may not be noticed immediately: the system may appear to continue running normally.

We do not attempt to identify or validate recovery code in general. Rather, we simply look for indications that the programmer is at least checking for the possibility of an error. If the check is clearly present, then presumably error handling or recovery follows. Section 4.3.4 discusses this aspect of the analysis in greater detail. As mentioned earlier, an error code may be safely overwritten after the error has been handled or checked. Figure 5 shows examples in which it is safe to overwrite errors that have been checked. In Figure 5a, err may receive one of several error codes on line 4. If this variable contains an error on line 6, then we continue to the next iteration of the loop, where the error is overwritten the next time line 4 is run. Overwriting an error code with the exact same error code is considered to be harmless, but the problem here is that different error codes might be returned by successive calls to function get_error. A similar pattern is illustrated in Figure 5b.

```
int *err;
...
while(...) {
  err = get_error();

  if (IS_ERR(err))
    continue;
  }
...
}
```

(a) Loop

```
int *err;
...

retry:
  ...
  err = get_error();

  if (err == ERR_PTR(−EIO)) {
    goto retry;
  }
```

(b) Goto

Figure 5: Two examples of safe error-overwrite patterns

In order to find bad overwrites, we identify the program points at which assignments are made to potentially-error-carrying storage locations. We recognize several patterns in which assignments cannot possibly be bad regardless the value contained by the receiver. Those assignments are not considered. The remaining assignments are potentially bad assignments. At each such assignment to pointer variable v at location l, we retrieve the set of error codes $\mathcal{N}_l$ that variable v may contain. If $\mathcal{N}_l \cap \mathcal{E} \neq \emptyset$, then we report the bad overwrite. A generalization of this strategy also allows us to check indirect assignments across pointers, as in "∗v = ..."; we give further details on this extension in Section 4.2.1.

4. ERROR PROPAGATION AND TRANSFORMATION

We assumed in Section 3 that we are able to retrieve the set of values that pointer variables may contain before being dereferenced, used in pointer arithmetic, or assigned. To provide this information, we adapt the error-propagation framework described by Rubio-González et al. [21], which performs an interprocedural, flow- and context-sensitive static program analysis to track errors until they are overwritten, dropped or handled. The goal is to find the set of values that each variable may contain at each program point. This problem resembles an over-approximating analogue of copy constant propagation [22].

Rubio-González et al. track how integer error codes propagate. However, their analysis does not support error transformation, which is necessary to find the bugs described in Section 3. For example, they assume that error propagation ends if the error is transformed into a pointer. In Figure 1, even though an error may be assigned on line 4, their analysis will not actually track error flow into variable xaroot because it is a pointer variable. Similarly, no pointer error value will be recognized as being returned at lines 15 and 20 because their analysis always clears the actual argument to any calls to function IS_ERR. Thus, no pointer error value is identified as returned by function open_xa_dir on line 11 in Figure 2.

We extend the error-propagation framework to support error transformation. The analysis is encoded as a path problem over weighted pushdown systems (WPDSs) [19]. A WPDS is a useful dataflow engine for problems that can be encoded with suitable weight domains, computing the meet-over-all-paths solution. The following subsections describe the WPDS components: (1) a pushdown system, (2) a bounded idempotent semiring, and (3) transfer functions. In order to support error transformation, we make an addition in one of the elements of the bounded idempotent semiring (see Section 4.2.1). We also replace the transfer functions of Rubio-González et al. with a new suite of functions that take into consideration pointer variables and error transformation (see Section 4.3). Section 4.4 explains how the dataflow problem is solved.

4.1 Pushdown System

A pushdown system (P, Γ, Δ) is used to model the control flow of the program, using the approach of Lal et al. [12]. Let P contain a single state $\{p\}$. Γ corresponds to program statements, and Δ is a set of stack-rewrite rules corresponding to edges of the interprocedural control flow graph (CFG). Control flow is encoded into these pushdown system (PDS) rules as follows:

$\langle p,a\rangle \hookrightarrow \langle p,b\rangle$	Intraprocedural flow from a to b
$\langle p,c\rangle \hookrightarrow \langle p,f_{enter}r\rangle$	Call from c to procedure entry f_{enter}, eventually returning to r
$\langle p,f_{exit}\rangle \hookrightarrow \langle p,\varepsilon\rangle$	Return from procedure exit f_{exit}

4.2 Bounded Idempotent Semiring

Let $\mathscr{S} = (D, \oplus, \otimes, \bar{0}, \bar{1})$ be a bounded idempotent semiring as defined by Reps et al. [19].

4.2.1 Set D

D is a set whose elements are drawn from $\mathscr{V} \to 2^{\mathscr{V} \cup \mathscr{C}}$, where $\mathscr{V}$ is the set of program variables, and $\mathscr{C}$ is the set of constant values. Constants include error codes, the special value *OK* (used to represent all non-error values) and the special value *uninitialized* (used to represent uninitialized variables). Each element in D is called a *weight* and is a mapping from variables to sets containing variables, error values, *OK* and/or *uninitialized*. This gives the possible values of a variable v following execution of a given program statement in terms of the values of constants and variables before that statement.

The error propagation analysis of Rubio-González et al. [21] does not allow for errors to be stored in pointer variables. Even in the special case of integer pointer parameters, error codes are stored in the integer variable pointed to, not in the pointer variable itself. We permit pointer variables of any type to store error values. This uncovers a new requirement: distinguishing between an error code stored in a pointer variable v and an error stored in $*v$. We introduce a *dereference variable* $*v$ for each pointer variable v. This allows us to distinguish and track error codes stored in either "level."

We replace dereference expressions with the corresponding dereference variables before performing the error-propagation analysis. Thus, the set $\mathscr{V}$ now also includes dereference variables. Even though the number of variables can increase considerably due to dereference variables, this does not represent a problem in practice. We apply a preliminary flow- and context-insensitive analysis introduced by Rubio-González and Liblit [20], which filters out irrelevant variables that cannot possibly contain error codes before applying the error-propagation analysis. Thus, we only keep those variables that are truly relevant to our analysis.

4.2.2 Operators Combine and Extend

The combine operator ($\oplus$) is used to summarize the weights of a set of paths that merge. It is applied component-wise, where a component is a variable. For all $w_1, w_2 \in D$ and $v \in \mathscr{V}$:

$$(w_1 \oplus w_2)(v) \equiv w_1(v) \cup w_2(v)$$

In other words, it is defined as the union of the sets of values a variable is mapped to in each of the paths being merged. This most-commonly happens when conditional branches join.

The extend operator ($\otimes$) is used to calculate the weight of a path. It is also applied component-wise. For all $w_1, w_2 \in D$ and $v \in \mathscr{V}$:

$$(w_1 \otimes w_2)(v) \equiv \begin{cases} (\mathscr{C} \cap w_2(v)) \cup \bigcup_{v' \in \mathscr{V} \cap w_2(v)} w_1(v') & \text{if } w_1(v) \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases}$$

The extend operator is essentially composition generalized to the power set of variables and constants rather than just single variables.

4.2.3 Weights $\bar{0}$ and $\bar{1}$

The weights $\bar{0}$ and $\bar{1}$ are both elements of the set D. The annihilator weight $\bar{0}$ maps each variable to the empty set and is the identity for the combine operator. The neutral weight $\bar{1}$ maps each variable to the set containing itself: a power-set generalization of the identity function. The weight $\bar{1}$ is the identity for the extend operator.

$$\bar{0} \equiv \{(v, \emptyset) \mid v \in \mathscr{V}\} \qquad \bar{1} \equiv \{(v, \{v\}) \mid v \in \mathscr{V}\}$$

4.3 Transfer Functions

Transfer functions define the new state of the program as a function of the old state. As discussed in Section 4.1, PDS rules correspond to edges in the CFG. Each PDS rule is associated with a *weight* or *transfer function*. Although here we describe weights as being associated with specific program statements, they are in fact associated with the edge from a statement to its unique successor. Conditionals are a special case: these are associated with multiple transfer functions since they have multiple outgoing edges.

The analysis has two chief modes of operation: *copy mode* and *transfer mode*. Consider an assignment $t = s$ where $t, s \in \mathscr{V}$ are distinct and s might contain an error code. In copy mode, the assignment $t = s$ copies the error value to t, leaving also an error value in s. In contrast, transfer mode leaves an error in t but removes the error from s, effectively transferring ownership of error values across assignments. The transfer functions described here correspond to copy mode.

All transfer functions share one key assumption: that pointer variables have no aliases inside a function. This makes our approach to pointers both unsound and incomplete, however it is simple and gives good results in practice.

4.3.1 Assignments

Table 1 shows the transfer functions for assignments. For the purpose of this discussion, we classify these into three groups. First consider assignments of the form $v = e$, where $e \in \mathscr{V} \cup \mathscr{C}$ and v is of type **int**. Let *Ident* be the function that maps each variable to the set containing itself, which is identical to $\bar{1}$. The transfer function for such an assignment is $Ident[v \mapsto \{e\}]$. In other words, v must have the value of e after this assignment, while all other variables retain whatever values they contained before the assignment, including e.

Next consider assignments that involve pointer or dereference variables. In either case, we need to update mappings at two levels. For example, for assignments of the form $*v = e$, where $*v$ is the dereference variable corresponding to pointer variable v and $e \in \mathscr{V} \cup \mathscr{C}$, the transfer function is $Ident[v \mapsto \{OK\}][*v \mapsto \{e\}]$. We map the dereference variable to any values e may contain. At the same time, we assume that the corresponding pointer variable contains a valid address, i.e. v is mapped to the *OK* value. The opposite occurs with assignments of the form $v = e$, where v is of some pointer type and $e \in \mathscr{V} \cup \mathscr{C}$ and not a pointer variable. In this case, variable v is mapped to whatever values e may contain, which must be non-address values. We assume that the corresponding dereference

Table 1: Transfer functions for assignments in copy mode

Pattern	Where	Transfer Function
$v = e$	$e \in \mathscr{V} \cup \mathscr{C}$ and v is of type int	$Ident[v \mapsto \{e\}]$
$v = e$	$e \in \mathscr{V} \cup \mathscr{C}$ and v is of pointer type but e is not	$Ident[v \mapsto \{e\}][*v \mapsto \{OK\}]$
$*v = e$	$*v \in \mathscr{V}$ and $e \in \mathscr{V} \cup \mathscr{C}$	$Ident[v \mapsto \{OK\}][*v \mapsto \{e\}]$
$v_1 = v_2$	$v_1, v_2 \in \mathscr{V}$ and v_1 and v_2 are of pointer type	$Ident[v_1 \mapsto \{v_2\}][*v_1 \mapsto \{*v_2\}]$
$v_1 = \&v_2$	$v_1, v_2 \in \mathscr{V}$ and v_1 is of pointer type	$Ident[v_1 \mapsto \{OK\}][*v_1 \mapsto \{v_2\}]$
$v = e_1 \ op \ e_2$	$e_1, e_2 \in \mathscr{V} \cup \mathscr{C}$ and op is a binary arithmetic, bitwise or logical operator	$Ident[v \mapsto \{OK\}]$
$v = op \ e$	$e \in \mathscr{V} \cup \mathscr{C}$ and op is a unary arithmetic, bitwise, or logical operator	$Ident[v \mapsto \{OK\}]$

variable $*v$ does not contain an error since v does not hold a valid address. Transfer functions for pointer-related assignments of the form $v_1 = v_2$ and $v_1 = \&v_2$ can also be found in Table 1.

Lastly, consider assignments of the form $v = e_1 \ op \ e_2$, where $e_1, e_2 \in \mathscr{V} \cup \mathscr{C}$ and op is a binary arithmetic, bitwise or logical operator. The program is converted into three-address form, with no more than one operator on the right side of each assignment. As noted earlier, error codes should be treated as atomic symbols, not numbers. Thus, we assume that the result of those operations is a non-error value. The transfer function is $Ident[v \mapsto \{OK\}]$, which maps the receiver variable v to the OK non-error value. The same transfer function applies for assignments of the form $v = op \ e$, where op is a unary arithmetic, bitwise or logical operator.

4.3.2 Function Calls

We primarily focus on parameter passing and value return for the case of non-void functions. Note that we transform the interprocedural CFG so that each function has a dummy entry node just before the first statement. We refer to the edge from the function call to this entry node as the call-to-enter edge. Each function also has a unique exit node. The edge from this node back to the call site is referred to as the exit-to-return edge.

Parameter Passing.

This is modeled as a two-step process: first the caller exports its arguments into global exchange variables, then the callee imports these exchange variables into its formal parameters. Exchange variables are global variables introduced for the sole purpose of value passing between callers and callees. There is one exchange variable for each function formal parameter.

Suppose function F has formal parameters $f_1, f_2, \ldots, f_n$, where some formal parameters may be of pointer type. Let $F(a_1, a_2, \ldots a_n)$ be a function call to F with actual parameters $a_i \in \mathscr{V} \cup \mathscr{C}$. We introduce a global exchange variable $F\$i$ for each formal parameter. We also introduce a global dereference exchange variable $F\$*i$ for each formal parameter of pointer type. The interprocedural call-to-enter edge is given the transfer function for a group of n simultaneous assignments $F\$i = a_i$, exporting each actual argument into the corresponding global exchange variable. Rules for assignment transfer functions apply. This means that, in the case of pointer arguments, we will be passing in the values of dereference variables when applicable.

The edge from the callee's entry node to the first actual statement in the callee is given the transfer function for a group of n simultaneous assignments $f_i = F\$i$. Note that since the transfer functions for assignments are applied, this group additionally includes an assignment of the form $*f_i = F\$*i$ for each parameter of pointer type. This step initializes each formal argument with a value from the corresponding exchange variable. For pointer variables, both the pointer and the corresponding dereference variable are initialized.

Figure 6 shows an example illustrating the idea behind pointer

```
void foo(int *a) {
  *a = −5;
  return;
}

int main() {
  int x = 0;

  foo(&x);

  x = 6;
  return 0;
}
```

(a) Original

```
1  int* foo$1;
2  int foo$*1;
3
4  void foo(int* a) {
5    int foo$*a;
6    a = foo$1; foo$*a = foo$*1;
7    foo$*a = −5;
8    foo$1 = a; foo$*1 = foo$*a;
9    return;
10 }
11
12 int main() {
13   int x = 0;
14
15   foo$1 = OK; foo$*1 = x;
16   foo(&x);
17   x = foo$*1;
18
19   x = 6;
20   return 0;
21 }
```

(b) Transformed

Figure 6: Example making parameter and return value passing explicit. Highlighted assignments emulate transfer functions.

parameter passing. Consider the code fragment in Figure 6a as though it is transformed into the code fragment in Figure 6b. The goal is to make parameter passing explicit. Function foo has one pointer parameter. We declare the corresponding pointer exchange and dereference exchange variables on lines 1 and 2, respectively. A dereference variable corresponding to the original pointer parameter is also declared on line 5. Exchange-variable assignments on lines 6 and 15 emulate the effects of the corresponding parameter-passing transfer functions.

Return Value Passing.

We also introduce a global return exchange variable $F\$\text{ret}$ for any non-void function F. This variable is used to pass the function result value from the callee to the caller. Thus, for non-void functions, the edges from the callee's last statements to the exit node are given the transfer function $Ident[F\$\text{ret} \mapsto \{e\}]$, where e is the return expression. The interprocedural exit-to-return edge is given the transfer function $Ident[r \mapsto F\$\text{ret}]$, where $r \in \mathscr{V}$ is the variable in which the caller stores the result of the call, if any.

In addition, we copy back certain other values upon function return. Many functions take a pointer to a caller-local variable where (at any of the two levels) an error code, if any, should be written. In particular, formal dereference variables are copied back into their corresponding dereference exchange variables. The edges from the callee's last statements to the exit node are additionally given the transfer function for a group of at most n simultaneous assignments $F\$*i = *f_i$. Finally, dereference exchange variable values are copied back to any actual variables at the caller's side. The interprocedural exit-to-return edge is given the transfer function for a group of at most n simultaneous assignments $*a_i = F\$*i$, where a_i is a pointer variable or $a_i = F\$*i$, where a_i is an *address of* expression. The idea is illustrated on lines 8 and 17 in Figure 6b.

4.3.3 Error Transformation Functions

We attribute a special meaning to calls to the function IS_ERR. As mentioned earlier, this Boolean function is used to test whether a variable contains a pointer error value. Typically, such calls are part of a conditional expression. Depending on the branch taken, we can deduce what the outcome is. For example, if the true branch is selected, then we know that the pointer definitely contained an error value. Conversely, when the false branch is chosen, the tested variable cannot possibly contain an error. Therefore, we map the variable passed as argument to IS_ERR to the value *OK* in the false branch.

Since our analysis supports error-valued pointers, calls to error-transformation functions ERR_PTR and PTR_ERR are treated as regular function calls, i.e., we apply the transfer functions for parameter passing and value return as discussed in Section 4.3.2. Previous work [20, 21] gives a special treatment to the function ERR_PTR (the source of most pointer error values) in order to truncate the propagation of any error passed through this function. As a result, the function PTR_ERR, which transforms pointer error values back to integer errors, had no real effect.

4.3.4 Error Handling

We preserve the special treatment of error-handling functions of Rubio-González et al. [21]. The analysis identifies a set of functions that are assumed to handle any errors contained in variables passed as their arguments. An example is the function printk: a variadic function that logs errors. While logging alone does not correct any problems, it clearly expresses programmer awareness that a problem has occurred; presumably it is being handled as well. In general, after errors are handled by these functions, they no longer need to be tracked. The transfer function for calls to such functions is $Ident[v \mapsto \{OK\}]$, where $v \in \mathscr{V}$ is an actual argument. We also include any corresponding dereference variables in the case of pointer actual arguments. The safe-overwrite patterns described in Section 3.3, are also considered as error-handling patterns here.

We can enable or disable error-handling pattern recognition in our analysis. It is enabled if the analysis results are used to find bugs for which error handling is relevant. For example, overwriting an unhandled error is considered bad while overwriting a handled error is harmless. On the other hand, error-handling pattern recognition is disabled if the analysis results are used to find bugs for which the distinction between handled and unhandled errors is unnecessary. For instance, dereferencing an error-valued pointer is still bad even after it has been passed to printk.

4.4 Solving the Dataflow Problem

We perform a poststar query [19] on the WPDS, with the beginning of the program as the starting configuration. We then read weights out from the resulting weighted automaton applying the path_summary algorithm of Lal et al. [11]. This algorithm allows to retrieve the weight representing execution from the beginning of the program to any particular point of interest. In other words, it finds the set of values that each variable may contain at each program point, including pointer and dereference variables. Moreover, we can retrieve the witness set associated with any given weight w. A witness set is a subset of the paths inspected, whose combine is w. This can be used to justify weight w. We use witness sets extensively to provide the programmer with useful and detailed diagnostic information (see Section 5).

Unlike prior work, our pointer-aware analysis tracks error codes even when transformed as discussed in Section 2. For example, error propagation no longer ends when an error is transformed into a pointer value. Error codes can now flow into and through pointer variables. In Figure 1, a pointer error value may be assigned to variable xaroot on line 4, which is checked for errors on line 5. If an error value, then the error is further propagated to any callers of this function, including reiserfs_listxattr (line 11 in Figure 2). The error is assigned to variable dir in Figure 2, which is checked for errors on line 13, transformed into an integer value on line 14 and further propagated to any callers if the error code is ENODATA.

5. FINDING AND REPORTING BUGS

We run the error-propagation and transformation analysis in two different configurations depending on the bugs to be found. The first configuration operates in copy mode with error-handling pattern recognition disabled; this finds bad pointer dereferences and bad pointer arithmetic. We use copy mode because dereferencing (or performing pointer arithmetic using) any copy of a pointer error value is equally bad. Thus, all copies of an error must be considered. Likewise, we disable error-handling pattern recognition because even after handling, an error code remains an invalid address which must not be dereferenced or used in pointer arithmetic.

The second configuration uses transfer mode with error-handling pattern recognition enabled. We use this configuration when finding bad overwrites. It is common for an error instance to be copied into several variables while only one copy is propagated and the rest can be safely overwritten. Rubio-González et al. [21] found that transfer mode leads to significantly fewer false positives when finding overwritten integer error codes. We find that this also holds for pointer error values. We enable error-handling pattern recognition because we are only interested in finding overwrites of unhandled error codes, thus handled errors must be identified.

We identify program locations and variables of interest as explained in Section 3 and use the analysis results to determine which of those represent error-valued pointer bugs. Each bug report consists of a sample trace that illustrates how a given error reaches a particular program location l at which the error is dereferenced, used in pointer arithmetic, or overwritten. We use WPDS witness sets to construct these sample paths.

Reports are marked as *may* or *must*. If $\mathscr{N}_l \cap \mathscr{E} \neq \emptyset$ and $OK \in \mathscr{N}_l$, then the variable may also contain the value *OK*, so we report that l *may* be the site of a bad pointer dereference. If $OK \notin \mathscr{N}_l$, then the variable can only contain error codes, so we report that the program *must* have a defect at l.

Figure 7 shows a more detailed version of the VFS bad pointer dereference from Figure 3. The error ENOMEM is first returned by function iget in Figure 7a and propagated through three other functions (cnode_make, fill_super and iput, in that order) across two other files (shown in Figure 7b and Figure 7c). The bad dereference occurs on line 1325 of file fs/inode.c in Figure 7c. The sample path produced by our tool is shown in Figure 7d. This path is automatically filtered to show only program points directly relevant to the propagation of the error. We also provide an unfiltered sample path, not shown here, showing every single step from the program point at which the error is generated (i.e., the error macro is used) to the program point at which the problem occurs. We list all other error

```
58  inode * iget(...) {
    ...
67    if (!inode)
68      return ERR_PTR(-ENOMEM);
    ...
81  }
    ...
89  int cnode_make(inode **inode, ...) {
    ...
101   *inode = iget(sb, fid, &attr);
102   if ( IS_ERR(*inode) ) {
103     printk("...");
104     return PTR_ERR(*inode);
105   }
```

(a) File coda/cnode.c

```
143 static int fill_super(...) {
    ...
194   error = cnode_make(&root, ...);
195   if (error || !root) {
196     printk("... error %d\n", error);
197     goto error;
198   }
    ...
207  error:
208   bdi_destroy(&vc−>bdi);
209  bdi_err:
210   if (root)
211     iput(root);
    ...
216 }
```

(b) File coda/inode.c

```
1322 void iput(inode *inode) {
1323
1324   if (inode) {
1325     BUG_ON(inode->i_state == ...);
1326
1327     if (...)
1328       iput_final(inode);
1329   }
1330 }
```

(c) File fs/inode.c

```
fs/coda/cnode.c:68: an unchecked error may be returned
fs/coda/cnode.c:101:"*inode" receives an error from function "iget"
fs/coda/cnode.c:104:"*inode" may have an unchecked error
fs/coda/inode.c:194:"root" may have an unchecked error
fs/coda/inode.c:211:"root" may have an unchecked error
fs/inode.c:1325: Dereferencing variable inode, which may contain error code ENOMEM
```

(d) Sample trace

Figure 7: Example of diagnostic output

codes, if any, that may also reach there.

6. EXPERIMENTAL EVALUATION

We use the CIL C front end [17] to apply preliminary source-to-source transformations on Linux kernel code, such as redefining error code macros to avoid mistaking regular constants for error codes. We also use CIL to traverse the CFG and emit a textual representation of the WPDS. Our analysis tool uses the WALi WPDS library [10] to perform the interprocedural dataflow analysis on this WPDS. We use *binary decision diagrams* (BDDs) [2] as implemented by the BuDDy BDD library [14] to encode weights.

We analyze 52 file systems (including widely-used implementations such as ext3 and ReiserFS), the VFS, the mm, and 4 heavily-used device drivers (SCSI, PCI, IDE, ATA) found in the Linux 2.6.35.4 kernel. We analyze each file system and driver separately along with both the VFS and mm. All bug reports for bad dereferences have been submitted to developers. We are in the process of submitting the rest of the reports.

6.1 Bad Pointer Dereferences

Our tool produces 41 error-valued pointer dereference reports, of which 36 are true bugs. We report only the first of multiple dereferences of each pointer variable within a function. In other words, as soon as a variable is dereferenced in a function, any subsequent dereferences made in this function or its callees are not reported by the tool. Similarly, we do not report duplicate bugs resulting from analyzing shared code (VFS and mm) multiple times.

Table 2 shows the number of error-valued pointer dereferences found per file system, module and driver. Note that the location of a bad dereference sometimes differs from the location where a missing error-check ought to be added. For example, the mm contains a dereference that is only reported when analyzing the Coda, NTFS and ReiserFS file systems. We count this as a single bad dereference located in the mm. So far, Coda developers have confirmed that this potential error-valued dereference is due to a missing error check in a Coda function. This is likely to be the case for the other two file systems. On the other hand, most of the other dereferences found in shared code are reported when analyzing any file system implementation. This suggests that the error checks

Table 2: Error-valued pointer dereferences. File systems, modules, and drivers producing no diagnostic reports are omitted.

	Number of Diagnostic Reports		
Dereference Location	True Bugs	False Positives	Total
AFFS	4	0	4
Coda	0	1	1
devpts	1	0	1
FAT	0	1	1
HFS+	1	0	1
mm	15	0	15
NTFS	3	0	3
PCI	1	0	1
ReiserFS	3	0	3
SCSI	1	0	1
VFS	7	3	10
Total	36	5	41

```
1  struct bnode ∗bnode_split(...) {
2   bnode ∗node = ...;
3   if (IS_ERR(node))
4     return node;
5   ...
6   if (node−>next) {
7     struct bnode *next = bnode_find(..., node->next);
8     next->prev = node->this;  // bad dereference
9     ...
10  }
11 }
```

Figure 8: Example of a bad pointer dereference due to a missing error check in the HFS+ file system

might be needed within the shared code itself.

We classify true dereference bugs into four categories depending on their source:

6.1.1 Missing Check

We refer to a missing error check when there is no check at all before dereferencing a potential error-valued pointer. 17 out of 36 (47%) true dereference bugs are due to a missing check. Figure 8 shows an example found in the HFS+ file system. Function and variable names have been shortened for simplicity. Function bnode_split calls function bnode_find on line 7, which is expected to return the *next* node. However, function bnode_find may also return one of two error codes: EIO or ENOMEM. Because of this, callers of function bnode_find must check the pointer result value for errors before any dereferences. Nonetheless, function bnode_split does dereference the result value immediately on line 8, without checking for any errors.

6.1.2 Wrong Check

We define a wrong check as any check other than an explicit error check (i.e., a call to function IS_ERR) involving the variable being dereferenced. This is the second-most-common scenario leading to error-valued pointer dereferences, accounting for 11 out of 36 true bugs (31%). We identify two variants of wrong checks. In the first case, the pointer dereference is preceded by a check for NULL but not for an error code (6 bugs). In the second case, there is an error check, but it involves an unrelated pointer variable (5 bugs).

Figure 9 shows an example of the first variant. The pointer variable p may receive the error code ENOMEM on line 4. If so, the **while** loop on line 5 is entered, then exits on line 8 since the condition on line 7 is true. Pointer p is passed as parameter to function r_stop on line 11, which checks it for NULL before calling function deactivate_super with variable v as a parameter. Since v contains an error code, the function deactivate_super is indeed called, which then dereferences the error-valued pointer on line 21.

6.1.3 Double Error Code

First identified by Gunawi et al. [6], double error code refers to cases in which there are two ways to report an error: by storing an error in a pointer parameter or passing it through the function return value. Action is often taken upon the function return value, which may or may not be checked for errors. At the same time, a copy of the error is left in the pointer argument and dereferenced later. This pointer is sometimes checked, but only for the NULL value. We find 5 (14%) true error-valued dereferences due to double error codes. An example of double error code can be found in Figure 3 (simplified version) or Figure 7 (extended version including diagnostics).

```
1  static int traverse(...) {
2    void ∗p;
3    ...
4    p = m->op->start(...);  // may receive error
5    while (p) {
6      ...
7      if (IS_ERR(p))
8        break;
9      ...
10   }
11 m->op->stop(..., p);  // passing error
12   ...
13 }
14
15 static void r_stop(..., void ∗v) {
16   if (v)
17     deactivate_super(v);  // passing error
18 }
19
20 void deactivate_super(struct super_block ∗s) {
21   if (!atomic_add_unless(&s->s_active, ...)) {  // bad dereference
22     ...
23   }
24 }
```

Figure 9: Example of wrong error check in the ReiserFS file system (function r_stop) leading to a bad pointer dereference in the VFS (function deactivate_super)

6.1.4 Global Variable

This category refers to the case in which an error code is stored in a global pointer variable. Only 3 error-valued dereferences fall into this group. In the first situation, the global pointer variable devpts_mnt (declared in the devpts file system) may be assigned one of two error codes: ENOMEM or ENODEV. This variable is dereferenced in a function eventually called from function devpts_kill_index, which is an entry-point function to our analysis, i.e. no function within the analyzed code invokes it. The second and third cases are similar and refer to the VFS global pointer variable pipe_mnt. This variable may be assigned one of six error codes, including ENOMEM and EIO. Variable pipe_mnt is dereferenced in a function eventually called from the system call pipe and also from entry-point function exit_pipe_fs.

6.1.5 False Positives

Finally, we identify 5 out of 41 reports (12%) to be false positives. Figure 10 illustrates an example. Pointer variable new_fl may receive an error code in line 5. There are two conditionals on lines 7 and 8 and on line 13. Variable new_fl is checked for errors in the first conditional, but the call to function IS_ERR is part of a compound conditional statement. Our tool correctly recognizes that even though there is an error, the whole expression may not evaluate to true. Nonetheless, the two conditionals are complementary: the conditional statement on line 13 evaluates to true if that on lines 7 and 8 was false, thereby covering all possibilities. The analysis does not detect this, so the dereference on line 17 is reported. This scenario is found twice.

Other false positives arise when (1) the error check is not exhaustive, but the missing error codes cannot possibly reach that program point; (2) there is a double error code and one is checked before dereferencing the other; and (3) a copy of the error is made and

```
int __break_lease(...) {
  struct file_lock ∗new_fl;
  int error = 0;
  ...
  new_fl = lease_alloc(...);  // may receive error
  ...
  if (IS_ERR(new_fl) && !i_have_this_lease
    && ((mode & O_NONBLOCK) == 0)) {
    error = PTR_ERR(new_fl);
    goto out;
  }
  ...
  if (i_have_this_lease || (mode & O_NONBLOCK)) {
    error = −EWOULDBLOCK;
    goto out;
  }
  error = wait_event_interrupt(new_fl->fl_wait, ...);
  ...
 out:
  ...
  return error;
}
```

Figure 10: Example false positive found in the VFS

Table 3: Bad pointer arithmetic

	Number of Diagnostic Reports		
Location	True Bugs	False Positives	Total
Coda	1	0	1
mm	15	0	15
ReiserFS	1	0	1
Total	17	0	17

checked before dereferencing the original variable. We can easily remove (1) since we have information regarding what error codes reach or not a given program point. Similarly, we can remove (3) by running the analysis in transfer mode. On the other hand, the false positives resulting from (2) and the example described in Figure 10 would require more effort to be removed.

6.2 Bad Pointer Arithmetic

Table 3 shows the results of our analysis of pointer arithmetic applied to pointers whose values are actually error codes, not addresses. Our tool reports 17 true instances of bad pointer arithmetic: 15 from the mm and the other 2 from the Coda and ReiserFS file systems. No false positives are identified. Note that we only report the first instance in which an error-valued pointer is used to perform pointer arithmetic. Subsequent bad uses, including bad dereferences, are not reported. Similarly, if the error-valued pointer is first dereferenced, subsequent uses in pointer arithmetic are not reported.

As with bad pointer dereferences in Section 6.1, most of the bad pointer-arithmetic instances are due to missing checks (70% or 12 out of 17 reports). The remaining bad pointer operations are surrounded by conditionals, but none of them include checks for errors in the operands. The majority of the reports involve pointer additions (65% or 11 out of 17 reports), while the rest involve subtraction. We find no bad increments or decrements.

In all cases but one, the error-valued pointer is assumed to contain a valid address that is used to calculate another address. The one exception is a calculation involving an error-valued pointer that determines the function return value. In all situations, the error-valued pointer may contain the error ENOMEM. There are two cases in which the pointer may additionally contain the EFAULT error code, which (ironically) denotes a bad address.

Most cases, including all those in the mm, are solely triggered by the SCSI driver. An example is shown in Figure 4. Callers of function kfree (line 3) may pass in a pointer variable that contains the error code ENOMEM, now in variable x. The variable is further passed to function virt_to_head_page when it is invoked in line 6. Finally, this function uses x to perform some pointer arithmetic in line 11, without first checking for any errors. Note that virt_to_page is defined in line 1.

```
1  struct buffer_head ∗ext3_getblk(..., int ∗errp) {
2    int err;
3    ...
4    err = ext3_get_blocks_handle(...);  // may receive error
5    ...
6    *errp = err;  // copy error
7    if (!err && ...) {
8      ...
9    }
10   return NULL;
11 }
12
13 struct buffer_head ∗ext3_bread(..., int ∗err) {
14   struct buffer_head ∗ bh;
15   bh = ext3_getblk(..., err);  // err has an error
16   if (!bh)
17     return bh;
18   ... // code leads to overwrites
19 }
```

Figure 11: Double error code in the UDF file system, leading to 12 overwrite false positives

6.3 Bad Overwrites

Our tool produces 7 reports describing the overwrite of error-valued pointer variables. As with other kinds of bugs, we eliminate duplicated reports that belong to shared code (VFS and mm). We identify 3 true bugs located in the mm. In 2 cases an error is stored in a global variable, which is overwritten later without first being checked for errors. In the remaining case, the error is stored in a static local variable. 3 out of the 4 false positives are found to be duplicates but located in file-system specific code. This is due to cloned (copied and pasted) code. We are not able to recognize this automatically, thus we count these as multiple reports. These overwrites are located in the ext2, System V, and UFS file systems and are due to complex loop conditions. The other false positive is found in the mm.

The tool reports 31 cases in which errors contained in dereference variables are overwritten, among which we only identify 1 true bug in the SCSI driver. The remaining false positives are associated with the ext3 (15 reports), UDF (12 reports) and UFS (2 reports) file systems and the SCSI (1 report) driver. There is a complete overlap between reports belonging to ext3 and UDF due to cloned code. Double error codes, as discussed in Section 6.1.3, cause most false positives (87%). Figure 11 shows an example. An error returned on line 4 is copied to the formal parameter ∗errp on line 6. Function ext3_getblk then returns NULL. The caller ext3_bread stores the

Table 4: Analysis performance for a subset of file systems and drivers. Sizes include 133 KLOC of shared VFS and mm code. Configuration 1 is used to find bad dereferences and bad pointer arithmetic while configuration 2 targets bad overwrites.

		Configuration 1		Configuration 2	
File System	KLOC	Time (min:sec)	Mem (GB)	Time (min:sec)	Mem (GB)
AFFS	137	2:48	0.86	3:17	0.87
Coda	136	2:54	0.83	3:15	0.84
devpts	134	2:36	0.81	3:06	0.82
FAT	140	3:06	0.88	3:21	0.90
HFS+	143	2:54	0.86	3:31	0.87
NTFS	162	4:12	1.37	4:39	1.39
PCI	191	3:24	1.00	3:55	1.02
ReiserFS	161	4:06	1.36	4:37	1.37
SCSI	703	11:00	2.42	13:04	2.52
Avg FS	-	2:54	0.87	3:16	0.89
Avg Drivers	-	5:24	1.44	6:18	1.50

returned value in bh, which is further returned on line 17. However, because we are tracking variable err and not variable bh, the analysis chooses the path that skips the conditional of line 16 and eventually leads to 12 overwrites. The same piece of code is found in file-system–specific code for both ext3 and UDF, accounting for every false positive in the later. Note that we do not find any overwrite of error-valued dereference variables due to assignments to pointer variables.

We find considerably fewer overwrites than Rubio-González et al. [21], which reported 25 true overwrites of integer error codes across five Linux file systems. One difference between integer and pointer error values is that there is an explicit error check function for the later (IS_ERR). The existence of such a function may influence developers into being more aware of error checking, thus contributing to fewer bugs. Another reason might be that although error-valued pointers are extensively used throughout the propagation of errors, these errors may ultimately end up back in **int** variables.

6.4 Performance

We use a dual 3.2 GHz Intel Pentium 4 processor workstation with 3 GB RAM to run our experiments. We analyze 1,538,082 lines of code, including white space and comments. Counting reanalysis of the VFS and mm as used by multiple file systems, we process 8,875,522 lines of code in total. Table 4 shows the size (in thousands of lines of code) for those file systems and drivers in which bugs are found. The table also includes running time and memory usage for the two different analysis configurations described in Section 5.

Running times corresponding to the first configuration range from 2 minutes 36 seconds (devpts) to 4 minutes 12 seconds (NTFS) for the subset of file systems shown in Table 4, with an average of 2 minutes 54 seconds for all 52 file systems. The average running time for drivers is 5 minutes 24 seconds. In particular, the analysis of the PCI and SCSI drivers take 3 minutes 24 seconds and 11 minutes, respectively. Regarding memory usage, our analysis takes from 0.81 to 2.42 GB. Thus, overall, we find that the analysis scales and performs quite well even with the added burden of tracking pointer-typed variables and their corresponding dereference variables.

Finally, we find that an average of 42% of the variables that hold errors at some point during execution are pointer variables. This shows that error transformation is not merely an anomaly; it is critical to understanding how error propagation really works.

7. RELATED WORK

Engler et al. [5] infer programmer beliefs from systems code and check for contradictions. They offer six checkers, including a NULL-consistency checker that reveals an error-valued pointer dereference. They also provide an IS_ERR-consistency checker, which reveals that NULL checks are often omitted when checking for errors. We do not infer beliefs. Instead, we track error codes to find what pointer variables may hold them and then report those that are used improperly, including but not limited to pointer dereferences.

Lawall et al. [13] use Coccinelle [18] to find bugs in Linux. Their case study identifies and classifies functions based on their known return values: a valid pointer, NULL, ERR_PTR, or both. The tool reports program points at which inappropriate or insufficient checks are detected. This can reveal some error-valued dereferences. However, dereferences made at functions that cannot be classified by the tool cannot possibly be found, and only 6% of the functions are classified as returning ERR_PTR or both ERR_PTR and NULL. Also, dereferences of error-valued pointers that are never returned by a function or further manipulated cannot be found. Our approach uses an interprocedural flow- and context-sensitive dataflow analysis that allows us to track error-pointer values regardless of their location and whether or not they are transformed.

Although identifying missing or inappropriate checks [5, 13] can lead to finding and fixing potential problems, our tool instead reports the exact program location at which problems might occur due to misuse of error-valued pointers. Note that our bug reports also help programmers find the program points at which error checks should be added in order to fix the problems reported. These tools aim to find a wider range of bugs; their discovery of missing or inappropriate error checks is only an example case study of a generic capability. Our tool is more specialized: it finds more specific kinds of bugs than Engler et al. [5] and Lawall et al. [13], and is more precise in finding these bugs.

Numerous efforts (e.g., [1, 3–5, 7, 8, 15, 16, 23]) have focused on finding NULL pointer dereferences using varied approaches. Our problem is a generalization of the NULL dereference problem, where instead of just one invalid pointer value, we are tracking 34 of them. However, our problem is also more complex. Error codes might transform during propagation, which does not occur with NULL pointers. In addition, while dereferencing and using NULL values in pointer arithmetic is as bad as using error values, overwriting NULL is perfectly benign. Overwriting unhandled error values, however, may have serious consequences.

The core of the error-propagation analysis we extend in this paper has been used for other purposes in the past. Rubio-González et al. [21] use error-propagation analysis to find dropped or overwritten integer error codes in Linux file systems. Rubio-González and Liblit [20] use a similar analysis to find the set of error codes returned by file-related Linux system calls and compare these against the Linux manual pages, finding hundreds of error-code mismatches. None of these support error transformation to find the kinds of bugs described in this paper. Also, they do not analyze the mm or any drivers. We use the new error-propagation and transformation analysis to find error-valued pointer-related bugs instead. Beyond revealing new types of bugs, the pointer-aware analysis described here could also be used to improve all previous error-propagation work by providing more complete tracking of errors across a variety of code.

8. CONCLUSIONS

In this paper we describe three kinds of bugs arising from defective interactions between error codes and pointers: bad dereferences, bad pointer arithmetic, and bad overwrites. We show how to extend an existing error-propagation analysis to account for error transformation as in the Linux kernel in order to find these bugs. We apply the analysis to 52 Linux file system implementations, the VFS, the mm and 4 drivers, finding a total of 57 true bugs. Hiding error codes in pointers may seem distasteful, but it is by no means uncommon: we find that 42% of the variables that may contain error codes are pointer variables. Thus, understanding the behavior of error-valued pointers is an important component to having a more complete understanding of how errors propagate in large systems such as the Linux kernel.

9. REFERENCES

[1] D. Babic and A. J. Hu. Calysto: scalable and precise extended static checking. In W. Schäfer, M. B. Dwyer, and V. Gruhn, editors, *ICSE*, pages 211–220. ACM, 2008.

[2] R. E. Bryant. Binary decision diagrams and beyond: enabling technologies for formal verification. In R. L. Rudell, editor, *ICCAD*, pages 236–243. IEEE Computer Society, 1995.

[3] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Softw., Pract. Exper.*, 30(7):775–802, 2000.

[4] I. Dillig, T. Dillig, and A. Aiken. Static error detection using semantic inconsistency inference. In J. Ferrante and K. S. McKinley, editors, *PLDI*, pages 435–445. ACM, 2007.

[5] D. R. Engler, D. Y. Chen, and A. Chou. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *SOSP*, pages 57–72, 2001.

[6] H. S. Gunawi, C. Rubio-González, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and B. Liblit. EIO: Error handling is occasionally correct. In *6th USENIX Conference on File and Storage Technologies (FAST '08)*, San Jose, California, Feb. 2008.

[7] D. Hovemeyer and W. Pugh. Finding more null pointer bugs, but not too many. In M. Das and D. Grossman, editors, *PASTE*, pages 9–14. ACM, 2007.

[8] S. Karthik and H. G. Jayakumar. Static analysis: C code error checking for reliable and secure programming. In C. Ardil, editor, *IEC (Prague)*, pages 434–439. Enformatika, Çanakkale, Turkey, 2005.

[9] A. Kelley and I. Pohl. *A book on C (4th ed.): programming in C*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[10] N. Kidd, T. Reps, and A. Lal. WALi: A C++ library for weighted pushdown systems. `http://www.cs.wisc.edu/wpis/wpds/`, 2009.

[11] A. Lal, N. Kidd, T. W. Reps, and T. Touili. Abstract error projection. In H. R. Nielson and G. Filé, editors, *SAS*, volume 4634 of *Lecture Notes in Computer Science*, pages 200–217. Springer, 2007.

[12] A. Lal, T. Touili, N. Kidd, and T. Reps. Interprocedural analysis of concurrent programs under a context bound. Technical Report 1598, University of Wisconsin–Madison, July 2007.

[13] J. L. Lawall, J. Brunel, N. Palix, R. R. Hansen, H. Stuart, and G. Muller. Wysiwib: A declarative approach to finding api protocols and bugs in linux code. In *DSN*, pages 43–52. IEEE, 2009.

[14] J. Lind-Nielsen. BuDDy - A Binary Decision Diagram Package. http://sourceforge.net/projects/buddy, 2004.

[15] A. Loginov, E. Yahav, S. Chandra, S. Fink, N. Rinetzky, and M. G. Nanda. Verifying dereference safety via expanding-scope analysis. In B. G. Ryder and A. Zeller, editors, *ISSTA*, pages 213–224. ACM, 2008.

[16] M. G. Nanda and S. Sinha. Accurate interprocedural null-dereference analysis for java. In *ICSE*, pages 133–143. IEEE, 2009.

[17] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In R. N. Horspool, editor, *CC*, volume 2304 of *Lecture Notes in Computer Science*, pages 213–228. Springer, 2002.

[18] Y. Padioleau, J. L. Lawall, R. R. Hansen, and G. Muller. Documenting and automating collateral evolutions in linux device drivers. In J. S. Sventek and S. Hand, editors, *EuroSys*, pages 247–260. ACM, 2008.

[19] T. W. Reps, S. Schwoon, S. Jha, and D. Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Sci. Comput. Program.*, 58(1-2):206–263, 2005.

[20] C. Rubio-González and B. Liblit. Expect the unexpected: Error code mismatches between documentation and the real world. In S. Lerner and A. Rountev, editors, *9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE 2010)*, Toronto, Canada, June 2010. ACM SIGPLAN and SIGSOFT.

[21] C. Rubio-González, H. S. Gunawi, B. Liblit, R. H. Arpaci-Dusseau, and A. C. Arpaci-Dusseau. Error Propagation Analysis for File Systems. In *Proceedings of the ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation*, Dublin, Ireland, June 15–20 2009.

[22] M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. In *POPL*, pages 291–299, 1985.

[23] Y. Xie, A. Chou, and D. R. Engler. Archer: using symbolic, path-sensitive analysis to detect memory access errors. In *ESEC / SIGSOFT FSE*, pages 327–336. ACM, 2003.