# Computer Sciences Department

**Verifying File System Properties with Type Inference**

Haryadi S. Gunawi
Shweta Krishnan

Technical Report #1695

August 2011

UNIVERSITY OF WISCONSIN MADISON

# Verifying File System Properties with Type Inference

Haryadi S. Gunawi and Swetha Krishnan
{*haryadi, swetha*}*@cs.wisc.edu*

## Abstract

*The storage stack is not trustworthy due to errors that arise from a variety of sources: unreliable hardware, malicious errors and file system bugs. Today, software errors play a dominant role due to their inherent complexity. In the first part of our project, we look towards verifying a specific file system property: on-disk pointer manipulation. We utilize CQUAL, a framework for adding type qualifiers with type inference support, and apply our analysis to the Linux ext2 file system. We find that adding qualifiers serves the valuable purpose of ensuring that on-disk pointers are accessed and manipulated correctly by the file system. Thus, we believe that the qualifiers we introduce would decrease the probability of bugs being introduced by file system programmers. We also describe our experience in using CQUAL and discuss its limitations. Based on our experience with CQUAL, we come up with a second analysis, a buffer management verifier, that fits better with the power of CQUAL by being simpler, yet more widely applicable to different file systems than the first analysis.*

## 1 Introduction

The storage stack is not trustworthy due to errors that arise from a variety of sources: unreliable hardware, transport, firmware, malicious errors and file system bugs. While traditionally hardware has been seen as the main source of errors, today, software errors play a more dominant part due to their inherent complexity. Alongside other ongoing file system bug-finding projects, we intend to verify two specific file system properties. First, we attempt to verify that file system manipulates on-disk pointers correctly. This means that we verify whether the pointers in the file system actually point to the disk blocks that they are assumed to point to. Without such verification, a programmer can introduce on-disk pointer bugs that will lead to a corrupt and unusable file system in the worst case. Moreover, on-disk pointer analysis is important for two reasons: First, even recent commodity file systems do not sanity-check their on-disk pointers [16, 15], e.g. the file system does not check if a file's data block pointer points to a wrong location. Second, the file system has the highest responsibility for the disk's contents. Hence, until pointer guards have been installed into the file system, it is necessary to check and prove that the file system moves on-disk pointers around correctly.

As each file system has a different internal representation of on-disk structures, we realize that one specification can only used by one file system. Hence, such analysis might be useful as a *verifier* rather than as a *bug finding* tool. We come up with a second analysis that is lightweight and reusable across many file systems. In particular, we attempt to verify a simple rule that relates to a reliability bug: "After disk read failure, the data in the buffer should not be used." We name this analysis as *buffer uptodate analysis*. Although this is a simple rule, recent work by Prabhakaran *et al.* shows that this violation still occurs in recent file systems [15]. Unfortunately, their work uses a black-box approach and hence could not pinpoint the locations of the bugs. We hope that with source code analysis we shall be able to locate these bugs.

In this project, we adopt an approach based on type qualifiers and type-inferencing [6, 10]. In our first analysis, we use CQUAL to add qualifiers to pointer types that point to on-disk structures. Simple qualifiers such as `$ibitmap` and `$bbitmap` could be added to the bitmap block pointers in order to distinguish between an inode bitmap and a block bitmap respectively. With such qualifiers, we can verify several basic structural properties of the ext2 file system [2]. Furthermore, with type qualifiers, we can distinguish different kinds of on-disk pointers besides the ones implied by existing structures. For instance, ext2 uses the `inode` structure to describe both file and directory inodes. Hence, to explicitly differentiate file and directory inodes, `$reginode` and `$dirinode` type qualifiers can be added to the pointer type `inode`.

The second analysis, the buffer uptodate analysis, requires flow-sensitivity; a buffer can have different states (and hence different qualifiers) at different locations in the program (similar to locking analysis). To apply flow-sensitive analysis to the buffer uptodate analysis, we could introduce qualifiers such as `$Null` and `$Uptodate` to annotate the buffer implying whether the buffer contains invalid data or invalid data respectively. With such qualifiers, the flow-sensitive buffer uptodate analysis will catch

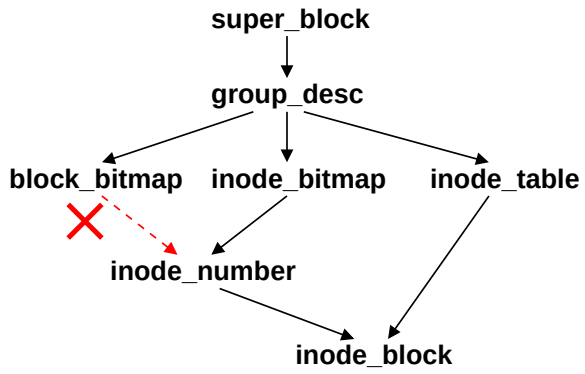


Figure 1: **Logical Dependencies:** *The graph above shows a chain of logical dependencies between various on-disk structures in ext2.*

any erroneous `$Null` buffer access.

The rest of this paper is structured as follows. We describe the disk pointer and the buffer uptodate analyses in Sections 2 and 3 respectively. We discuss future work in Section 4. Related work is discussed in Section 5. Finally, we conclude in Section 6.

# 2 Disk Pointer Analysis

With disk pointer analysis, we aim to capture and verify semantic properties relating to the ways in which various on-disk data and metadata are accessed and manipulated. For example, in the ext2 file system, the pointer to the group descriptor block must originate from the superblock, and the pointer to the inode bitmap block must originate from the group descriptor block. We have analyzed disk pointers for the Second Extended File System [2] since it is one of the simplest and oldest file systems implemented in the Linux kernel, and so a good candidate to start with.

We find that our disk pointer analysis is largely flow-insensitive (there are few cases where we require flow-sensitivity, but as illustrated in Section 2.4.2, we find that they can easily be transformed to be made flow-insensitive). Of course, given that our analysis is restricted to ext2, we cannot say this with confidence for other file systems.

It is important to note that this analysis is specific to a particular file system since the type qualifiers added are dependent on the metadata representations used by the file system, and the analysis applied is dependent on the control flow from one metadata type to another.

## 2.1 Background on Ext2 Types

The ext2 file system code uses various structure types to represent data and metadata stored on the disk. As seen in Figure 1, each structure is derived from one or more other structures and integers. First, the superblock is read from the disk, and since the superblock contains information such as the number of groups and the number of group descriptors per block, this structure is required to obtain the group descriptor block for a particular block group. Similarly, the group descriptor contains the information needed for retrieving the block bitmap, the inode bitmap and the inode table blocks from the disk. The inode bitmap is in turn used to fetch the inode number of a file being created. To retrieve the inode block number, both the inode number and the inode table are required. Thus, we can easily observe that there is a chain of logical dependencies between the various on-disk structures, and if there are violations anywhere in this chain, we could be manipulating data in incorrect ways. For instance, if one accidentally uses the block bitmap block's contents to try and retrieve an inode number, one would end up with an erroneous value.

### 2.1.1 Implicit Types

In ext2, there are two important elements that have implicit types: block number which is represented as `unsigned long` (`sector_t`) and the buffer head structure with its data field. A block number could assume types `block_bitmap` or `inode_bitmap` and so on. Ext2 uses the generic in-memory data structure `buffer_head` to read the contents of a disk block (be it a data or metadata block of any sort) into memory. This structure contains a `char* b_data` field which is a pointer to the data of that block. This data field is cast to various ext2 specific structures for metadata, so as to extract the necessary information. Thus, a buffer head could take on implicit types corresponding to the inode structure, the group descriptor structure and so on.

These implicit types come into play in ways "assumed to be safe." For instance, the function used in ext2 to read any disk block is:

```
struct buffer_head*
sb_bread (unsigned long block);
```

A dangerous use of this function could be as follows:

```
1. struct buffer_head *bh;
2. bh = ext2_read_inode_bitmap(...);
     // read an inode_bitmap block by a call
     // to sb_bread() within the function.
     // bh will now refer to an
     // inode_bitmap block
3. ext2_get_group_desc(..., bh);
     // within this function, bh->b_data is
     // cast to (struct ext2_group_desc *), so
```

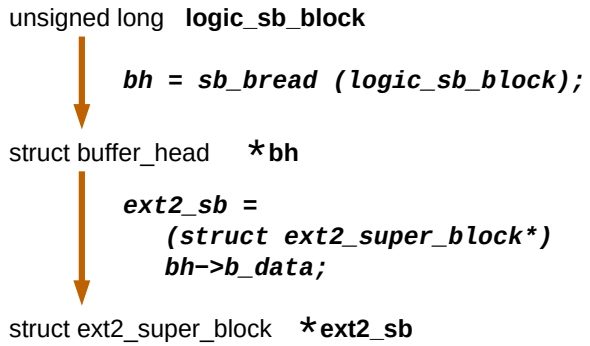


Figure 2: **Logical Connection:** *The diagram above shows how three different entities, the block number, the buffer head pointer and the* `ext2_super_block` *pointer, are logically connected by the control flow. This connection can be captured by attaching a type qualifier such as* `$superblock`.

```
// it now points to a group descriptor
// block.
```

Thus, here the inode bitmap block has been "tainted" with a group descriptor block's contents. We could trap this error using CQUAL by adding qualifiers to tag the buffer head instances with the block type they represent. For example, we would qualify the buffer head pointer returned by `ext2_read_inode_bitmap()` with `$ibitmap`, to signify an inode bitmap, and qualify the buffer head type passed to `ext2_get_group_desc()` with `$groupdesc` to signify a group descriptor. Then CQUAL would flag an error since the buffer head `bh` is used both as an `$ibitmap` in line 2 and as a `$groupdesc` in line 3.

### 2.1.2 Logical Connections

It is also interesting to observe that in retrieving metadata from blocks, a series of actions are taken that relate to the same logical entity. For example, as Figure 2 shows, the superblock number is passed to a function that reads the block corresponding to the given block number, and returns a buffer head pointer. To retrieve metadata information from the superblock, the data field of the buffer head is then cast to a pointer to the appropriate structure (`ext2_super_block` in this case). Thus, three different entities - block number, buffer head pointer and superblock pointer - are logically connected by the control flow, and this connection can be well captured by attaching a single type qualifier (say `$superblock`) to all of them.

## 2.2 Implementation

We outline our approaches of adding qualifiers to ensure correct usage of a single structure that assumes various types (such as buffer head), and to impose dependency checks.

•**Qualifying implicit types associated with buffer head:** The first naive approach we take is simple, that is, annotate every function declaration that receives or uses a buffer head pointer with the appropriate qualified type according to correctness requirements. For example, we add in our CQUAL prelude file [5], declarations of the form:

```
static $bbitmap struct buffer_head*
read_block_bitmap(unsigned int block_group);
```

•**Using qualifier subtyping:** In our lattice configuration, we define `$bbitmap` and `$ibitmap` to be qualifiers with level=value and sign=eq. We also introduce a new qualifier `$bitmap` (level=value, sign=neg) and add qualifier subtyping relationships: `$ibitmap` $\prec$ `$bitmap` and `$bbitmap` $\prec$ `$bitmap`. This allows us to take into account functions requiring a bitmap of any type as argument such as `ext2_set_bit` function that sets a bit in a bitmap:

```
int ext2_set_bit (int bit, $bitmap void *addr);
```

•**Leveraging qualifier polymorphism:** A key observation is that the block number and buffer head types, even though used in different places as multiple types, have a relationship: A function that takes a block number corresponding to say, an `$ibitmap`, must always return a buffer head pointer that semantically refers to an inode bitmap (`$ibitmap`). That is, any qualifier `Q` on a block number must also be reflected in the returned buffer head. This is a clear case for using qualifier polymorphism, which CQUAL provides. We add annotations of the following form to our prelude file:

```
$_1 struct buffer_head*
sb_bread ($_1 sector_t block);
```

This enables the flow of qualifiers from the block number of a block to the corresponding data. We also add a polymorphic function `getBData(bh)` to qualify the `b_data` pointer of a `struct buffer_head` with the same qualifier as the buffer head pointer passed to it. We have to do this since we disabled `fieldptrflow` for the qualifiers in general, hence preventing qualifiers from flowing from a structure to all its fields.

•**Adding interfaces:** To enforce properties, we need to attach a qualifier to the point of origin of a chain of dependencies, for instance, the block number of the block bitmap. Initially, we could not get CQUAL to work with

type qualifiers added directly within the kernel files. So, we had to find a workaround such that our qualifier annotations would be limited to just the CQUAL prelude file and would not need to touch the kernel files. For this, we introduce interface functions, such as the one below, in our prelude file:

```
$bbitmap tagBlkBitmap (sector_t blknum)
{
  return ($bbitmap sector_t) blknum;
}
```

We then instrument the points within the kernel where an unqualified variable (such as `blknum`) is being used directly and replace it with a call to the corresponding interface function that would qualify it.

## 2.3 Enhancements

We conducted the second phase of the above analysis, where we made the following improvements.

•**Minimizing the number of manual annotations:** We found that the interface functions approach required taking a close look at the kernel code to find points at which to insert calls to these functions. To avoid this, we made a second attempt in trying to add qualifiers directly in the kernel code, and eventually succeeded in getting CQUAL to work with in-kernel qualifiers.

There are two things here that are of benefit. First, by adding qualifiers at variable and structure field declaration points, and in function declarations within the file system code, we could reduce the total manual annotations. For example, adding the qualifier `$bbitmap` ahead of `bg_block_bitmap` field of the `ext2_group_desc` structure avoids adding annotations at each place where this field is being used.

Second, by adding qualifiers ahead of structure types instead of annotating each instance of the structure, we again save on the number of annotations. Since CQUAL does not regard qualifier annotations appropriately when the qualifier is added directly to the structure type definition, we instead explicitly typedef each structure type to a new name, and attach a qualifier to that new name. Of course, this means that we have to replace the type of each structure instance declaration with the new name, but that is fairly automatic and can be done easily with a find-and-replace. For example, we replace struct `ext2_group_desc*` with `EXT2_GD` everywhere, where `EXT2_GD` is defined as:

```
typedef
    $groupdesc struct ext2_group_desc*
    EXT2_GD;
```

•**Merging derivations from multiple qualifiers:** We find that in certain cases, a single qualified type may have a dependency on more than one qualified type. This arises in cases where there is a *computational dependency* within the file system's manipulation of pointers to disk blocks. Consider the simplified code snippet below:

```
struct ext2_inode *ext2_get_inode
    (int inum, ..)
{
    offset = ((inum - 1) % ...);
    iblock = gd->inode_table +
             (offset >> BLOCK_SIZE);
    struct buffer_head *bh =
        sb_bread (iblock);
    return
        (struct ext2_inode*) bh->b_data;
}
```

The question here is whether we can use type qualifiers to verify that the computation of `iblock` from two different integers is correct. Checking the absolute correctness of such a computation might be impossible, however, a cheap way to check partial correctness could be achieved by using qualifiers and qualifier propagation provided by CQUAL. The code above shows that in order to obtain an inode block (via `iblock` number), the file system must know the inode number (`inum`) and also the starting address of the inode table. The inode table pointer can only be derived by an access to the group descriptor block and an inode number can be acquired by finding the first free bit in the inode bitmap. In order to ensure that a programmer does not erroneously use say a number derived in some other way in place of the inode number obtained from the inode bitmap, we introduce qualifiers for such entities (that is, anything that is derived directly or indirectly from the disk). We also add interface functions, as shown below, to our prelude file, to capture such computational dependencies between a qualifier and two other qualifiers.

```
$inode sector_t
computeInode ($itable it, $inumber inum);
```

Note that CQUAL cannot do this automatically since the addition operation between two different qualified types would confuse CQUAL as to which qualifier (`$itable` or `$inumber`) it should propagate for the l-value of the assignment, hence creating a conflict of qualifiers. Current semantics of arithmetic addition in CQUAL require that the qualifiers on the operands match. However, since there are not many places where such computational dependencies with multiple entities are seen, we feel that adding manual annotations in such places can still be considered low overhead.

## 2.4 Case Studies

To test the effectiveness of our type-qualifiers approach to file system verification, we ran CQUAL on the annotated kernel ext2 file system code. We set up our own

CQUAL lattice file and prelude file prior to this experiment. While we could not uncover any bugs (logical or computational dependency violations) in the ext2 code, we come up with the following interesting case studies where small programming errors or missed code can lead to violations that our analysis with CQUAL can catch.

### 2.4.1 Case Study 1

In this first case study, we mix up use of buffer_heads in the same function. Consider the code fragment below:

```
void ext2_free_blocks(...)
{
    struct buffer_head *bitmap_bh = NULL;
    struct buffer_head *bh2;
    ...
    bitmap_bh = read_block_bitmap(...);
    get_group_desc(&bitmap_bh);
    // get_group_desc(&bh2);
}
```

Here, the intent (as observed from the commented line) is to use two different buffer head pointers for the block bitmap returned and the argument passed to `get_group_descriptor()`. However, a small programming error wherein the `bitmap_bh` returned by `read_block_bitmap()` is passed as argument to `get_group_desc()` can lead to incorrect results when `bitmap_bh`'s referred data is later used. Our framework, with qualifiers `$bbitmap` and `$groupdesc`, can catch this error, as can be seen in the CQUAL output [1] shown below:

```
44 WARNING: **sbi->s_group_desc treated as
            $bbitmap and $groupdesc
**sbi->s_group_desc: $bbitmap $groupdesc

139 $bbitmap == desc->bg_block_bitmap
 91          == cast
 91          == *sb_bread_ret@91
 91          == *bh
102          == *read_block_bitmap_ret
229          == *read_block_bitmap_ret@229
229          == *bitmap_bh
234          == **bh@234
234          == **bh
 69          == **sbi->s_group_desc
 44          == $groupdesc
```

Note that the only annotations added explicitly are to the `bg_block_bitmap` field of the group descriptor structure, and to the group descriptor structure itself. The rest of the lines in the output result from automatic qualifier inference and propagation that CQUAL provides.

[1] Original output has been modified (file paths removed) for ease of representation

### 2.4.2 Case Study 2

In the second case study, we ensure that "parent" inode must always be a directory inode. Consider the code fragment below:

```
struct find_group
    (..., $dirinode struct inode* parent);

struct inode* ext2_new_inode
    ($dirinode struct inode* dir, ...);
{
  struct inode* inode = new_inode();
  if(S_ISDIR(..)) {
    inode_dir =
        ($dirinode struct inode*) inode;
    find_group_dir(..., dir);
  }
  else {
    // NOT  a directory inode
    inode_gen =
        ($nondirinode struct inode*)inode;
    find_group(inode_gen,...);
    // find_group(dir,..);
  }
}
```

The function `find_group()` finds a block group for the new inode, implementing ext2's policy of trying to place the new inode in the same block group as its parent directory. However, notice that the directory assumption of the argument to this function is very loosely captured in the name `parent` given to the argument. We know that any inode that is treated as a parent of some other inode should be a directory inode. However, this function (like many other such functions) does not check whether the inode passed is indeed a directory inode. Rather, this check happens somewhere deep inside the function call hierarchy using `S_ISDIR()` on the `mode` field , and is assumed elsewhere. If there happens to be a call to this function, as shown above, with a non-directory inode as argument instead of a directory inode, then we would end up with allocations very different from that of the policy, since the regular inode is no parent. Again, CQUAl can catch this error by having qualifiers such as `$dirinode` and `$nondirinode` at the appropriate places. For example, every call to `S_ISDIR()` could be followed with an annotation attaching `$dirinode` to that inode instance.

Notice that we actually find the need for a flow-sensitive analysis here since the same inode instance would need to be qualified differently depending on the result of `S_ISDIR()`. However, in many places, we could easily transform the flow-sensitive analysis to a flow-insensitive analysis by using different names for the qualified inode instances on the two paths of the branch. There are other places where such transformation is not straightforward, and we ignore those for our present analysis. In any case, properties like the one shown above can be checked even in the absence of flow-sensitivity.

### 2.4.3 Case Study 3

This final case study shows an erroneous computation of inode block number. Consider the code fragment below:

```
void ext2_preread_inode(struct inode *inode)
{
  EXT2_GD gdp;
  ...
  block_group =
      (inode->i_ino - 1) /
      INODES_PER_GROUP(inode->i_sb);

  gdp = ext2_get_group_desc
      (inode->i_sb, block_group, &bh);

  offset =
      ((inode->i_ino - 1) %
      INODES_PER_GROUP(inode->i_sb)) *
      INODE_SIZE(inode->i_sb);

  block = computeInode(
       le32_to_cpu(gdp->bg_block_bitmap),
       offset >> BLOCK_SIZE_BITS(inode->i_sb));

  // block = computeInode(
  //   le32_to_cpu(gdp->bg_inode_table),
  //   offset >> BLOCK_SIZE_BITS(inode->i_sb));
}
```

Here, the computation of inode block number is done by an addition involving an offset derived from an `inode->i_ino` (qualified with `$inumber` in the inode structure), and the `bg_inode_table` (qualified with `$itable` in the group descriptor structure). To enforce a correctness check on the computation, we substitute the direct arithmetic with an interface function `computeInode()`, as explained in Section 2.3. With this, we can catch an erroneous computation like the one shown above that attempts to compute the inode block number from the block bitmap rather than from its `bg_inode_table` field.

## 2.5 Evaluation

We use the latest version (0.991) of CQUAL for our disk pointer analysis. We run our analysis on the ext2 files of the Linux kernel version 2.6.7. We also include in this directory the ext2-relevant header files and .c files such as buffer.c that are present elsewhere in the kernel directory structure. Table 1 shows the qualifiers we add to distinguish different entities (that live on the disk) in the ext2 file system. We add a total of 14 qualifiers. Notice that a few qualifiers like `$error` and `$bitmap` do not actually correspond directly to entities that live on the disk, but are used to qualify the error number field and to enforce subtyping relationships respectively. We add these qualifiers to make our analysis more complete. The subtyping relationships introduced are as follows:

| Qualifier | Sign |
|---|---|
| $superblock | eq |
| $groupdesc | eq |
| $bbitmap | eq |
| $ibitmap | eq |
| $bitmap | neg |
| $itable | eq |
| $inumber | eq |
| $indirect | eq |
| $dirinode | eq |
| $nondirinode | neg |
| $reginode | eq |
| $linkinode | eq |
| $inode | neg |
| $error | eq |

Table 1: **Type qualifiers:** *The table shows the type qualifiers added to check ext2.*

```
$bbitmap     < $bitmap
$ibitmap     < $bitmap
$dirinode    < $inode
$reginode    < $inode
$linkinode   < $inode
$nondirinode < $inode
$reginode    < $nondirinode
$linkinode   < $nondirinode
```

We add a total of 32 qualifier annotations in the ext2 code. We also make changes at a total of 15 places in the code to introduce calls to interface functions such as `getBData()`. Given that ext2 has around 7000+ lines of code, the number of changes we introduce is trivial.

The total number of files in ext2 that we touch is 12 (including the extra files such as header files). Since our focus was not on verification of special features, we do not touch certain files (such as acl.c and xattr_security.c) dealing with permissions and security in the ext2 code.

## 2.6 Limitations of CQUAL

In the course of our analysis using CQUAL (version 0.991), we have come across several limitations in CQUAL's current implementation, and have also found work arounds for some of them.

•**No polymorphism for structure fields:** CQUAL does not support polymorphism when dealing with fields of structures. For example, the following is not supported:

```
$_1 struct A {
    $_1 int a;
    char b;
};
```

Such support would be have useful for us with respect to the structure `buffer_head` and its `b_data` field, since this field should be qualified depending on the qualifier attached to the structure instance. We work around this by introducing the `getBData()` interface function as explained in Section 2.2.

•**No support for variable properties for the same qualifier with respect to different structures:** CQUAL allows us to specify "fieldptrflow" for each qualifier introduced, which refers to the direction in which the qualifier flows between pointers to aggregates (structures and unions) and pointers to their fields. However, there is no way to specify a different "fieldptrflow" direction for the same qualifier with respect to different structures. For example, qualifier `$q` should propagate from `struct buffer_head` to its `b_data` field but the same qualifier `$q` should not propagate from `struct ext2_group_desc` to its `bg_inode_bitmap` or `bg_block_bitmap` field. We work around this by disabling "fieldptrflow" and instead qualifying structure fields separately or introducing interface functions for qualifier propagation as the case may be.

•**Definition of polymorphic function is ignored:** CQUAL ignores definition of polymorphic functions. Even though commenting certain lines of code causes CQUAL to analyze the body of the function, but calls to that function still go to the polymorphic declaration. As a result, though we can catch a type-qualifier error internal to the function, we cannot catch one that requires analyzing the function and connecting those results to the results at a call-site.

•**False positives due to qualifier semantics in arithmetic operations:** Qualifiers flow from one operand to the other in case of arithmetic operations, causing false positives in some cases. For example, the current semantics for arithmetic "+" in CQUAL require that the qualifiers on the operands match. Hence, the following code fragement causes CQUAL to complain with an error *"incompatible operands of +,"* since the variable `c` gets two qualifiers `$q_t1` and `$q_t2` that flow to it through the addition. But as we can see, there is no real incorrect use of qualified types here; this is a false alarm. The creators of CQUAL suggest that in future, the semantics of arithmetic operations can alternately require the qualifiers on the operands to lower-bound the qualifier on the result.

```
$q_t1 int a;
$q_t2 int b;
int c;
int z;
int v;
z = a + c;
v = b + c;
```

# 3 Buffer Uptodate Analysis

Since each file system has its own internal representation of on-disk structures, our first analysis described in the previous section is not applicable to other file systems besides ext2 and ext3. Hence, such analysis might be useful as a *verifier* rather than a *bug finding* tool. To understand how CQUAL can be utilized as a bug finding tool we study how CQUAL has been used in previous work. Table 2 shows the code coverage and number of bugs found in three analyses that uses CQUAL. The two take-away points from Table 2 are: all analyses are lightweight and the corresponding rules are reusable across many source codes. For example, in the user/kernel pointer analysis, the rule that programmers must transform a user pointer to a kernel pointer appropriately can be applied to any part of the kernel code. Also logically, having a large code base as the target code gives a higher probability in finding bugs.

In the file system world, one analysis that is reusable across many file systems is an analysis that relates to buffer management. Almost all file systems use a buffer cache as the mechanism to store file system data in the memory and thus must use the buffer head abstraction. This provides the ground for our second analysis that we name *buffer management verifier*. The other property of the buffer head besides being used by many file systems is that the buffer head contains a lot of states representing the current status of the buffer as depicted in Figure 3.

Having a lot of states implies that there are many rules that can be checked upon buffer usage. For example, the traditional lock/unlock analysis can also be applied here; to find a reliability bug, we could enforce a rule such as "a dirty buffer should not be released" because it should be submitted to the disk first, otherwise the file system could reach an inconsistent state; to find a performance bug, we could also catch redundant operations such as an unnecessary disk read of an uptodate buffer. Although it is possible to impose such rules by deploying runtime checks throughout the kernel code, such deployment is not a common practice as it could lead to excessive checking and performance degradation [13]. Instead, when an operation is reached, a corresponding state is usually assumed to be true. For example, when accessing the data field of the buffer, it is assumed that the buffer is indeed uptodate. Thus, type qualifiers might be an appropriate approach to unearth and assert the assumed states.

## 3.1 Buffer Access Rule

For this project, we attempt to verify a simple rule that relates to a reliability bug: *After a disk read failure, the data in the buffer should not be used*. Although this is a simple rule, recent work by Prabhakaran *et al.* shows that

| Conf. | Analyses | Target | Bugs |
|---|---|---|---|
| PLDI'02 | Locking Analysis [7] | 892 files | 11 |
| SEC'01 | Format String Vulnerabilities [17] | 10 programs (20 KLOC/prog) | 2 |
| SEC'01 | User/Kernel Pointer Bugs [10] | Whole kernel | 11 |

Table 2: **CQUAL as a Bug Finding Tool:** *The table shows how CQUAL has been used as a bug finding tool. The analyses shown in the second column are generally lightweight and applicable to large code base.*

```
struct buffer_head
{
    unsigned long b_state   // buffer state bitmap
    atomic_t      b_count   // users using this block
    u32           b_size    // block size (usually 4KB)
    sector_t      b_blocknr // disk block number
    char*         b_data    // pointer to data block
    ...
}

enum bh_state_bits
{
    BH_Uptodate,      // Contains valid data
    BH_Dirty,         // Is dirty
    BH_Lock,          // Is locked
    ...
}
```

Figure 3: **Buffer Head Structure and Buffer States:** *The code above shows important fields of the buffer head and bitmaps that represent the state of the buffer (there are 14 state bits in total). Buffer head structure is the abstraction that will hold a disk block. For our analysis, we only track* `b_state` *and* `b_data` *fields of the buffer head and the* `BH_Uptodate` *state bit.*

this violation still occurs in recent file systems [15]. In their work, they construct approximately 50 test cases encompassing different block types and POSIX APIs, and run test cases to three different file systems in Linux 2.6. They find that there are two cases where disk read failure is ignored, one in ReiserFS and one in IBM JFS. Unfortunately, their work uses a black-box approach and hence could not pinpoint the locations of the bugs. We hope that with source code analysis we can locate these bugs.

In order to verify such a property, we need to observe the buffer uptodate bit in the buffer state field of the buffer head (See Figure 3). If the uptodate bit is set, it means the data field of the buffer head points to a location that contains a *valid* data, otherwise it points to an invalid data. Note that this is not a null/non-null pointer analysis; in both cases, uptodate and not uptodate, the buffer data pointer points to an already allocated region of memory. However, the data in the region could only be valid if the uptodate bit is asserted, thus the need to check the uptodate flag in correspondence with buffer data usage.

Figure 4 shows an example of a function that performs disk read (line 15) and correctly checks the buffer uptodate flag (line 18) before using the data field of the buffer (line 26). The function `read_inode` in particular reads an inode buffer from the disk and casts the buffer data to an inode structure. Meanwhile, please ignore the instruction that is marked as "CQUAL instr"; these instructions will be described in the next section. The check on the buffer uptodate bit of the buffer state is done through a wrapper function `buffer_uptodate`. Since the buffer uptodate flag is only set and cleared by the disk driver, buffer uptodate check must be performed every time after a synchronous disk read. The file system should not make any assumption that the read succeeds.

Generally, if the buffer is not uptodate, control flows to a failure handling block, but if the buffer is uptodate the data field is allowed to be accessed. For example, the code segment in Figure 4 flows to the `fail:` block which sets the inode to NULL if the buffer is not uptodate. But if the read succeeds, the buffer can be accessed and cast to an inode structure.

```
13  void read_inode (struct inode* inode, struct buffer_head *bh)
14  {
15      submit_bh(bh);                                          // Submit a disk read
16      change_type (*bh, $Null struct buffer_head);            // CQUAL instr
17
18      if (!buffer_uptodate(bh)) {                             // is buffer uptodate?
19          goto fail;                                          // if not, goto fail
20      }
21      else {
22          change_type (*bh, $Uptodate struct buffer_head);    // CQUAL instr
23      }
24
25      assert_type(*bh, $Uptodate struct buffer_head);         // CQUAL instr
26      inode = (struct inode*) bh->b_data;                     // read data and cast
27      return;                                                 //   to inode struct
28
29    fail:
30      inode = NULL;                                           // if fail, set inode
31  }                                                           //   to NULL
```

Figure 4: **An Example of Disk Read Operation.** *The function* `read_inode` *illustrates an example of how disk read is performed. The function* `read_inode` *accepts a buffer head* `bh` *whose data field (*`b_data`*) has been allocated, and then it primarily performs three things: submits a disk read (line 15), checks if the disk read succeeds or fails by checking the buffer uptodate bit of the buffer state field (line 18), and casts the data field of the buffer to an inode structure if the disk read succeeds (line 26). All instructions that are marked with "CQUAL instr" are the CQUAL instructions that are added automatically in order to run CQUAL flow-sensitive analysis on the code. In particular,* `change_type` *changes the qualifier of the buffer head and* `assert_type` *ensures that the buffer head has the same given qualifier.*

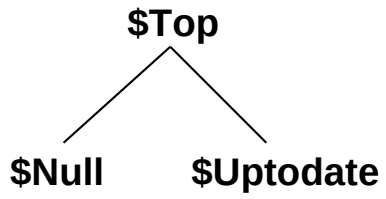


Figure 5: **Lattice for Buffer Uptodate Analysis.** *We introduce three qualifiers* `$Null`, `$Uptodate`, *and* `$Top` *in order to perform flow-sensitive buffer uptodate analysis.*

## 3.2 Qualifiers

To apply a flow-sensitive analysis to the buffer uptodate analysis, we introduce 3 qualifiers to annotate the buffer head as depicted in Figure 5: `$Null`, `$Uptodate`, and `$Top`. A `$Null` qualifier implies that the buffer has not been verified uptodate hence could contain invalid data. On the other hand, an `$Uptodate` data is valid to be used.

The `$Top` qualifier, and the subtyping `$Null<$Top` and `$Uptodate<$Top`, are needed because at every join point in the program (e.g. `fail:`), CQUAL joins qualifiers together. In particular, in one predecessor to `fail:`, `*bh` has qualifier `$Null`, and in the other predecessor, `*bh` has qualifier `$Uptodate`. Since CQUAL does not know which path was taken to reach that point, it decides that at `fail:`, `*bh` has qualifier `$Null| $Uptodate`, which is not allowed in the partial order in the absence of `$Top`. Thus, adding the `$Top` qualifier would allow CQUAL to use the qualifier `$Top` at the join point.

Each time after a disk read is performed, we add a `change_type($Null)` CQUAL instruction that changes the type of the buffer to `$Null` (line 16 of Figure 4). Subsequently, we have the buffer uptodate check where the corresponding `if` block describes what should happen if the buffer is not uptodate. The idea here is that we need to introduce an `else` block whose body contains a CQUAL instruction that changes the buffer type to `$Uptodate` (line 21-23). Finally, before any uses of the buffer data field we invoke the `assert_type` CQUAL instruction to assert that the data buffer is indeed valid (line 25).

The flow-sensitive buffer analysis will catch any erroneous `$Null` buffer access. For example, imagine if the `goto fail;` in line 19 is mistakenly removed. CQUAL will flag an error, with output as below since there is a flow from `change_type($Null)` at line 16 to `assert_type($Uptodate)` at line 25.

```
CQUAL error output:
 16 incompatible types in change_type
```

```
*bh@16: $Uptodate
 16     $Null <= __change
 16           <= *bh@16
 18           <= *bh
 18           <= *bh
 25           <= assert
 25           <= $Uptodate
```

## 3.3 Limitations

Unfortunately, unlike the flow-insensitive disk pointer analysis described in earlier section, the flow-sensitive buffer analysis cannot be directly applied to the kernel code, mostly due to the limitations of the flow-sensitive implementation of CQUAL, which we briefly describe here. More description on these limitations can be found in the CQUAL paper and manuals [4, 5, 7]. We are not sure whether the features below are theoretically impossible with a flow-sensitive analysis or they are just not yet implemented.

First, CQUAL's flow-sensitive implementation does not have field-sensitivity; all instances of a structure share the same field qualifiers. For example, given `struct foo { struct buffer_head a; } x, y;`, `x.a` and `y.a` will share qualifiers, which is a property that is not desirable for the kind of analysis we do. Second, polymorphism is not supported; as described by Foster *et al.*, analysis without context-sensitivity will not be accurate due to excessive false positives [10]. Third, there is no "cast preserve"; in our buffer analysis, cast preserve is needed because buffers are often stored in generic lists, where `buffer_head` is casted to `list_head`. In summary, without the features above, running flow-sensitive CQUAL directly on the kernel gives way too many errors (hundreds) that are impossible to analyze. In order to continue with the analysis, this limitation only leaves us with an option to reduce the kernel code base.

## 3.4 Code Reducer in CIL

We write a program using CIL [14] to reduce the kernel code base into a much smaller code base that only covers functions and instructions related to buffer manipulation. The program consists of approximately 600 lines of code. Besides reducing the code, the program will also automatically insert qualifier annotations as described in Section 3.2, including adding the "`else`" block to change the qualifier to `$Uptodate`. This automated process is actually a big advantage of the code reducer. In addition, control flow such as while, if, etc. must be preserved correctly in order to have the same flow as in the original code.

To significantly remove unrelated code, we only preserve functions and instructions that relate to buffer head. Moreover, since doing that still gives many false positives, we specifically only preserve functions that perform disk read and access the buffer data. One slight problem here is that buffer heads are stored within other structures, and further these structures are also nested in others (e.g. `super_block->super_block_info->buffer_head`). Hence, functions and instructions that touch those related structures must also be preserved. Although the process of finding those related structures can be automated, right now we manually specify them. Furthermore, due to lack of support for polymorphism, we must remove many insignificant calls that are unrelated to the analysis. For example, functions such as `brelease()` and `lock_buffer()`, that also manipulate the buffer, do not affect our buffer uptodate analysis in any way and hence can be omitted.

## 3.5 Results

We targeted three file systems for our analysis: ext2, ext3, and ReiserFS which consist of approximately 300 KLOCs. We have not analyzed IBM JFS because IBM JFS uses another buffer abstraction, `struct bio`, instead of the generic buffer head interface `struct buffer_head`. We believe it is straightforward to extend our analysis to include the `bio` interface. The CIL output consists of approximately 6 KLOCs, which is only 2% of the original code base. We know from previous findings that ReiserFS and IBM JFS each has one bug of non-uptodate buffer access. Yet, with our current approach, we still could not find the bugs, which we believe is because of the very low code coverage of the analyzed code.

Nevertheless, since we want to test whether our analysis is useful or not, we inject bugs into the source code and check whether the bug is detected as expected or not. In fault-injection experiments, the first question that comes up is where to inject the bugs. We write another program with CIL to help pinpoint interesting places to inject the bugs. We define and locate three types of interesting functions where we inject the bugs: *R-A* is a function that performs disk read (calls `submit_bh` and accesses the data buffer within the same function); *R-FA* is a function that performs disk read and then calls another function that accesses the buffer; *FR-A* is a function that calls another function to perform the disk read, and then accesses the buffer.

In order to introduce bugs that are realistic, we define our bug as removal of one important line. This line usually contains a statement that defines the control flow of the program such as the `goto fail;` statement in line 19 of Figure 4. So far, we have only analyzed *R-A* type of functions. We find three related functions (`__ext3_get_inode_loc`, `journal_read_transaction`, and `search_by_key`)

and introduce the bugs in these three functions. All the injected bugs are detected as expected.

# 4 Future Work

We realize that our buffer uptodate analysis is far from complete, largely due to the low coverage of the reduced kernel code, which is again due to the limitations of the flow-sensitive CQUAL. However, we feel that even with low code coverage, we still could come up with interesting case studies where we could inject our own bugs, hence suggesting that this might be a good path to proceed along. We feel that we have little hope in having a fully supported flow-sensitive CQUAL in the near future. Hence, this leaves us with couple of options described below.

From our buffer uptodate analysis, we learn that not many existing tools work perfectly on the Linux kernel due to its massive size. Hence, we have constructed a two phase analysis. The first one is about reducing the code base. Currently, we write our own code reducer using CIL. We plan to study other tools that have similar functionality, such as the Chen's MOPS infrastructure [3]. As mentioned in Section 3, there are many rules that can be checked within the buffer management world. Therefore, an infrastructure that produces much smaller code whose contents are only relevant with buffer management will be very useful as many buffer analyses can be performed on top of it.

In the second phase, which is the analysis itself, we have experienced that although the flow-sensitive CQUAL is theoretically appropriate to our analysis, it does not have many features that are present in the flow-insensitive CQUAL. Hence, we need to investigate other tools that are appropriate and fully supported for this kind of analysis. One example might be the GrammaTech's CodeSonar [8].

For the on-disk pointer analysis, we plan to conduct verification of further file system correctness properties such as "no hard links to a directory inode". We also intend to explore other interesting buffer management analyses, such as for ones to detect performance hitches. We would also need to expand our analyses to more file systems in order to comprehensively demonstrate the effectiveness of such analyses.

# 5 Related Work

Being the primary guardian of the disk contents, verifying file system properties can be considered a good research path. As far as we know, after two decades since the first "intelligible" file system was published [12], not many formal file system verifiers exist until today. Some file system specifications have been formally introduced in some previous work [1, 9]. Recently, Joshi and Holzmann realized that testing the file system is not enough, hence they took a "mini" challenge to a formal specification for a file system and verify it [11]. They intend to solve the challenge within three years.

The framework for adding type qualifiers to a language, which serves as a means to define polymorphic types was proposed by Foster *et al.* in [6]. In subsequent work, Johnson *et al.* use type qualifiers to find user/kernel pointer bugs [10]. In our project, we take the same approach but for solving file system specific problems. Another tool that might be useful is Sparse, which is a static type-checking program written by Linus Torvalds specifically for the Linux kernel [19]. However, CQUAL seems to be more powerful than Sparse.

Our motivation to analyze file system code stems from the work of Yang *et al.* [20] and Bairavasundaram *et al.* [16]. Yang *et al.* use model checking to show that the file system code is buggy and find serious errors in all the file systems explored. Bairavasundaram *et al.* emphasize how on-disk pointers form a critical piece of the file system code that is likely to fail under erroneous conditions. Surprisingly, they find that Windows NTFS, one of the most widely used file systems, does not verify on-disk pointers thoroughly before using them, causing the system to crash and rendering the file system unusable.

A recent approach to ensuring safe accesses to disk blocks through pointers is described in the ACCESS prototype built over Type-Safe Disks [18], which is intended to be implemented on disk processors. TSD only distinguishes between normal data blocks that do not have any outgoing pointers and reference blocks (such as inode blocks) that have incoming and outgoing pointers. Thus, the only access constraints that TSD can enforce are:"a block cannot be accessed until a valid reference block pointing to it is accessed." and and "No pointer creations/deletions can be made to root blocks." The two major differences between TSD and our disk pointer analysis are: First, our approach is more lightweight (with type inference support) and requires less changes to existing systems; with TSD, existing file systems must be modified in order to support new TSD APIs. Second, TSD ensures that the final product (the collection of on-disk pointers) is always correct, but it does not solve the problem of file system bugginess; we take into account the various ways in which the file system uses on-disk pointers.

# 6 Conclusions

We have shown that checking type qualifiers with automatic type inference is a powerful means of enforcing file system correctness properties. Our on-disk pointer

analysis with qualifier annotations is a useful and simple way for kernel programmers to trap the errors they might make while modifying the file system. With the bare minimal overhead of a just few annotations, identifying disk-pointer mismatches in the code with a simple tool can be some relief for coding in the complex kernel domain. Our buffer management verifier which performs flow-sensitive analysis using qualifiers, can be used to prevent reliability bugs in the file system code. Since qualifiers are statically checked, our analysis does not impact the performance of a file system in action. Nevertheless, our analyses are not complete. We hope that with further extension and improvement of our analyses, we shall be able to uncover known as well as unknown bugs in the file systems of today.

## Acknowledgments

We would like to thank Ben Liblit for advising this project. We also extend particular thanks to Jeff Foster and Rob Johnson for answering our questions promptly.

## References

[1] K. Arkoudas, K. Zee, V. Kuncak, and M. Rinard. Verifying a file system implementation. In *Proceedings of the 6th International Conference on Formal Engineering Methods (ICFEM '04)*, Seattle, Washington, November 2004.

[2] R. Card, T. Ts'o, and S. Tweedie. Design and Implementation of the Second Extended Filesystem. In Proceedings of the First Dutch International Symposium on Linux, 1994.

[3] H. Chen and D. Wagner. MOPS: an infrastructure for examining security properties of software. In *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS '02)*, Washington, DC, November 2002.

[4] J. S. Foster. "CQUAL User's Guide Version 0.98", Feb. 2004.

[5] J. S. Foster. "CQUAL User's Guide Version 0.991", Feb. 2004.

[6] J. S. Foster, M. Fahndrich, and A. Aiken. A Theory of Type Qualifiers. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '99)*, Atlanta, Georgia, May 1999.

[7] J. S. Foster, T. Terauchi, and A. Aiken. Flow-Sensitive Type Qualifiers. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '02)*, Berlin, Germany, June 2002.

[8] GrammaTech. "CodeSonar". www.grammatech.com.

[9] M. Heisel. Specification of the Unix file system: A comparative case study. In *4th International Conference on Algebraic Methodology and Software Technology (AMAST '95)*, Montreal, Canada, July 1995.

[10] R. Johnson and D. Wagner. Finding User/Kernel Pointer Bugs With Type Inference. In *Proceedings of the 13th USENIX Security Symposium (Sec '04)*, San Diego, California, August 2004.

[11] R. Joshi and G. J. Holzmann. A Mini Challange: Build a Verifiable Filesystem. In *Workshop on Verified Software: Theories, Tools, Experiments (VSTTE '05)*, Zurich, Switzerland, October 2005.

[12] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[13] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. CCured: Type-Safe Retrofitting of Legacy Software. *ACM Transactions on Programming Languages and Systems*, 27(3), May 2005.

[14] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. Cil: An infrastructure for c program analysis and transformation. In *International Conference on Compiler Construction (CC '02)*, pages 213–228, April 2002.

[15] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. IRON File Systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, pages 206–220, Brighton, United Kingdom, October 2005.

[16] M. Rungta, L. N. Bairavasundaram, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Limiting Trust in the Storage Stack. In *The 2nd International Workshop on Storage Security and Survivability (StorageSS '06)*, Alexandria, Virginia, November 2006.

[17] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting Format-String Vulnerabilities with Type Qualifiers. In *Proceedings of the 10th USENIX Security Symposium (Sec '01)*, Washington, D.C, August 2001.

[18] G. Sivathanu, S. Sundararaman, and E. Zadok. Type-Safe Disks. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, Seattle, Washington, November 2006.

[19] L. Torvalds. "Sparse". tree.celinuxforum.org/pubwiki/moin.cgi/Sparse.

[20] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using Model Checking to Find Serious File System Errors. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, San Francisco, California, December 2004.