Listening to Programmers
— Taxonomies and Characteristics of Comments in Operating System Code

Yoann Padioleau, Lin Tan and Yuanyuan Zhou
University of Illinois, Urbana Champaign
{*pad, lintan2, yyzhou*}@*cs.uiuc.edu*

Abstract

Innovations from multiple directions have been proposed to improve software reliability. Unfortunately, many of the innovations are not fully exploited by programmers. To bridge the gap, this paper proposes a new approach to "listen" to thousands of programmers: studying their programming comments. Since comments express programmers' assumptions and intentions, comments can reveal programmers' needs, which can provide guidance (1) for language/-tool designers on where they should develop new techniques or enhance the usability of existing ones, and (2) for programmers on what problems are most pervasive and important so that they should take initiatives to adopt some existing tools or language extensions.

We studied 1050 comments randomly sampled from the latest versions of Linux, FreeBSD, and OpenSolaris. We found that 52.6% of these comments could be leveraged by existing or to-be-proposed tools for improving reliability. Our findings include: (1) many comments describe code relationships, code evolutions, or the usage and meaning of integers and integer macros, (2) a significant amount of comments could be expressed by existing annotation languages, and (3) many comments express synchronization related concerns but are not well supported by annotation languages.

1 Introduction

1.1 Motivation

Software bugs hurt software reliability. Therefore, much effort from multiple directions has been devoted to improve software reliability by (1) designing and promoting programming language extensions [7, 27], better programming languages, or new development tools [8, 30] to prevent bug introduction in the first place; (2) building bug detection tools [9, 20] to find bugs before they cause damage; or (3) designing annotation languages [2, 11, 23, 34] to provide special information, such as buffer lengths, to facilitate and improve the accuracy of bug detection.

Unfortunately, many of the above innovations are not fully leveraged by programmers. This is mainly due to the existing gap between programmers and tool/language designers. In particular, what new language features or tools should be developed or improved with enhanced usability in order to address programmers' most important needs? Additionally, what problems are pervasive and critical so that programmers should adopt some annotation languages, code editor features or new bug detection tools in their software development practice?

To bridge the gap, researchers conduct user studies [1, 3] to understand programmers' needs as well as tool usability. While this is a promising direction, it raises many challenges such as how to ensure the representativeness of user studies, how to be objective, or how to avoid interfering with programmers' activities. Additionally, developers often do not realize what support/tools they need or need most. While we definitely should promote and encourage programmer user studies, we should also seek alternatives in the mean time.

Code comments, while significantly under-exploited, can provide a great data source for understanding programmers' needs. As comments are more flexible and expressive than source code, developers use them to complement their source code for various purposes, such as explanations, assumptions, specifications, etc. A significant amount of comments is already available in most modern software. For example, 23.1-29.7% (0.6-1.2 million lines) of the three popular operating systems (Linux, FreeBSD, and OpenSolaris), are comments, excluding blank lines.

While many comments are simply explanations of the code, we found that 52.6$\pm$2.9%[1] of the comments from Linux, FreeBSD, and OpenSolaris are not merely explanations (Section 4). These comments could (1) motivate new tools and language techniques or justify existing ones to address programmers' real needs, and (2) show where programmers need help and should adopt new or existing tools and language extensions to address these problems.

[1]The statistical margin of error is shown with 95% confidence level.

Specifically, studying programmers' comments could help in the following aspects:

(1) Programming language: Comments could motivate the design of new programming language extensions. The comments in OpenSolaris shown below specify the field name to which a value is assigned, e.g., assigning `15` to the `length` field. Specifying such information in comments is not only inconvenient but also error-prone when the `struct` definition changes. To address these limitations, the GCC *designator* extension was proposed to specify such field names in the code, e.g., `.length = 15`. This example shows that some needs indicated by comments have *already* been addressed by language extensions.

```
const struct st_drivetype st_drivetypes[] = { ...
    ``Unisys ...'', /* .name ... */
    15,              /* .length ... */ ...      };
```

(2) Software bug detection: Comments can be checked against source code for bug detection. An example comment from the Linux kernel is `/* This function must not be called from interrupt context */`. Our previous work, iComment [33], leveraged around 1% of the Linux kernel comments to automatically detect lock-related and call-related bugs. Although iComment showed the potential of using comments for software reliability, we wonder what information we could extract for bug detection from the remaining 99% of 1.2 million lines of the Linux kernel comments.

(3) Annotation language: To broaden the impact of annotation languages, we would benefit from studying: (1) How often programmers use comments instead of annotations for concerns that are already covered by existing annotation languages. If there are a significant amount of such comments, it may be possible to convert comments into existing annotation languages for automated bug detection. (2) What important concerns expressed in comments are not covered by existing annotation languages and could thus motivate new types of annotations.

(4) Code editor features: Comments can be used by editors to increase programmer productivity. For example, to make it easier to find relevant code, developers put cross reference information in comments, such as `/* See comment in struct sock definition to understand ... */`. It would be beneficial for an editor to utilize such comments and display correlated code and comments in the same window to reduce code navigation time. Such a feature may greatly increase programmer productivity as it has been found that programmers spend on average 35% of their programming time on code navigation [18].

State-of-the-art and Challenges: Although a large amount of comments exist, comment characteristics are poorly studied. In contrast, there is a flurry of work on bug characteristics [5, 24, 32] and characteristics of other program artifacts [25] which provided insights on improving software reliability.

Studying comments has two major challenges. First, it is difficult to understand comments. As comments are written in natural language, they are ambiguous. Therefore, the exact meaning and scope of comments may not be easy to determine by just reading the comments, and may require a thorough understanding of the semantics of the relevant code and comments. Second, unlike software bugs, which have a relatively well accepted classification, there is no comment taxonomy based on comment *content* yet. It is difficult to discover the underlying recurring patterns among comments, as many comments appear distinct.

1.2 Our Contributions

We take the first initiative, to the best of our knowledge, in studying comment *content* characteristics comprehensively. We started from comments in operating system code due to their overwhelming complexity and critical importance of reliability. More specifically, we manually examined 1050 comments randomly selected from three popular open source operating systems written in C: Linux, FreeBSD, and OpenSolaris (started as closed software).

We study the comments from several dimensions including (1) *what* is in comments, (2) *whom* the comments are written for or written by, (3) *where* the comments are, and (4) *when* the comments were written. We have made our comment database publically available at `http://opera.cs.uiuc.edu/CComment`.

Our comment study reveals that 52.6±2.9% or roughly 736,109[2] comments in the three operating systems are not merely explanations and could be leveraged by various techniques, either existing or to-be-proposed. We call these comments *exploitable* comments.

Our major findings that can benefit language/tool designers and system programmers include (§ denotes in which section the finding and its *implications* are discussed):

- **Finding 1 (§4.1):** 22.1% of the exploitable comments clarify the usage and meaning of integers and integer macros that are used by programmers to represent various restricted data types. Each of these restricted data types has its constraint, e.g., one should not confuse read-only port macros with read-write port macros. Since such constraints are not automatically assured by current compilers, bugs can be introduced.
- **Finding 2 (§4.2):** 16.8% of the exploitable comments specify or emphasize particular code relationships such as which function is responsible for filling a specific variable. These data flow, control flow, or other correlation

[2]Estimated based on the percentage (52.6%) and the total number of comments (1,399,447). The breakdown of statistics in each OS is shown in Section 6.

based comments are neither fully leveraged to check for bugs nor used to help programmers navigate code easily.

- **Finding 3 (§4.3):** 5.6% of the exploitable comments describe code evolution aspects such as cloned code (copy-pasted), deprecated code, and TODOs, which could be leveraged to check for various consistency issues. For example, a bug fix on one location should be propagated to its cloned copies [20].
- **Finding 4 (§4.4):** There are a significant number (4.7%) of comments related to properties of locks and synchronization, which almost no existing annotation languages except the proprietary tool Lock_Lint [23] can express.
- **Finding 5 (§5):** 10.7% of the exploitable comments can be expressed by existing annotation languages (ignoring the easy-to-use aspect) to help bug detection.
- **Finding 6 (§6):** OpenSolaris, which started as closed software, exhibits essentially the same comment characteristics as its open source counterpart. Our result complements the previous study that shows *code* characteristics are similar in the three operating systems [28].

To further understand how tied to OS and C code our results are, we have performed a preliminary study on 1050 comments from open source software written in C++ and Java (MySQL, Firefox and Eclipse), whose results are presented in Section 7.

2 Methodology

This section presents our comment classification process, the evaluated operating systems, the tools we developed that made this study easier, and threats to validity. The unit of comments we use is the comment block, not line.

2.1 Process

It is challenging to design a taxonomy classifying completely around 1.4 million comments using a limited number of mutually exclusive categories. Our main approach is a combination of random sampling, iterative refinement, and double verification by 2 independent opinions about the taxonomies and comment labels.

In addition to manually reading comments and code, we studied the last patch that modified a comment to better understand the comment and the motivation for the modification. Furthermore, we automatically extracted many properties for all comments, such as where a comment is located, the time a comment was written, and its author.

2.2 Evaluated Operating Systems

We randomly sampled and examined 1050 comments from three open source operating system kernels (350 per OS): Linux, FreeBSD, and OpenSolaris (an open source OS branched from the *closed* Solaris OS since June 2005) (Table 1). The three OSs are written in C and we studied the versions retrieved on February 12, 2008. We chose to study OSs because they are large, complex and their reliability is critically important. Specifically, as OS code is developed by many programmers, contains many components, uses a variety of algorithms, data structures, and software architectures, we hope it may exhibit a wide range of programmers' needs. Additionally, by studying a completely open-source code base (Linux), a code base with a strong academic component (FreeBSD), and a code base with a commercial lineage (OpenSolaris), we hope to find general software laws or interesting specificities.

Software	LOC	LOM	%	# of Comments
Linux	5.2M	1.2M	23.1%	729,923
FreeBSD	2.4M	0.6M	25.0%	289,413
OpenSolaris	3.7M	1.1M	29.7%	380, 111

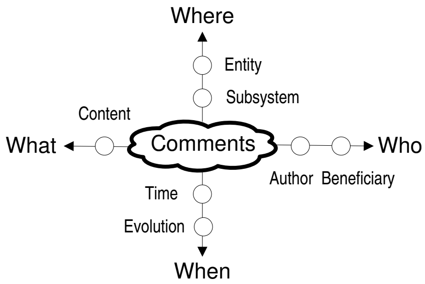
Table 1. Evaluated Operating Systems. LOC is lines of code and LOM is lines of comments. % denotes the percentage of comments in the entire code base. Blank lines are excluded.

2.3 Our Navigation and Analysis Tool

We developed a compiler front-end and an interface to version control systems to extract a rich amount of information about comments, such as to which C constructs they are attached, the time of their creation, and their authors. Unlike classical C front-ends (including gcc), which remove comments and C preprocessor constructs by first calling `cpp`, we, in contrast, are precisely interested in such information. Our compiler front-end thus understands code before it is transformed by C preprocessor and retains comments. We also developed a GUI to conveniently navigate, annotate, and analyze comments, using an interface inspired by iTunes and faceted information retrieval. With the help of simple heuristics, our tool can extract the location information of a comment, e.g., next to a loop, next to a field in a structure, etc. Therefore, our tool can group and display comments next to the same type of source code constructs/entities together, enabling us to focus on one group of comments at a time. This functionality made it easier for us to find recurring patterns in comments.

2.4 Threats to Validity

While we believe that the comments from the 3 examined OSs well represent comments in systems software, we do not intend to draw any general conclusions about comments in all software. Similar to any characteristic study, our findings should be considered together with our evaluation methodology. Section 7 presents a preliminary study of comments in other large software written in C++ and Java. Since we examined comments manually, subjectivity is inevitable. However, we tried our best to minimize such subjectivity by using double verification. While we sampled a fixed amount (350) of comments from each OS,



Content (§4,§5)	Beneficiary (§4,§5)	Entity (§6.1)	Subsystem (§6.1)
Type	Programming Language	Macro	Core
Interface	Bug Detection Tool	Function Definition	Architecture
Code Relationship	Annotation Language	Field	File System
PastFuture	Code/Text Editor	Variable	Drivers
Meta	Version Control System	If	Network
Explanation	Other	Function Call	Other
		Loop	
		Assignment	
		...	

Figure 1. Comment taxonomies. Comments that belong to the 4 categories above the dotted line are *exploitable* comments.

an alternative is to sample comments proportional to the total number of comments in the software. However, as these 3 OSs have similar comment distribution (Section 6), the two approaches should produce similar results. Infrequently appearing comments may not show up in our sample, but could still be very important. In the future, it would be interesting to give comments different weight, e.g., more weight for comments that take more effort to write, and less weight for automatically generated comments.

3 Taxonomies of Comments

We classify comments from different angles, according to the questions (inspired by Buckley et al. [4]) we are interested in (Figure 1):

- **What?** *Content:* what is in a comment? Does it contain useful information?
- **Who?** (1) *Beneficiary:* who (e.g., testers) or which tool (e.g., a bug detection tool) can benefit from a comment? (2) *Author:* who is the author of a comment, an expert, a beginner, or was it generated by a tool?
- **Where?** (1) *Code entity:* where in a file is a comment located (e.g., header, function, before a loop, in a macro)? (2) *Subsystem:* in which subsystem is a comment (e.g., drivers, file system)?
- **When?** (1) *Time:* when was a comment written? (2) *Evolution:* how do comments evolve over time?

In this paper we are mostly interested in the *Content* and *Beneficiary* dimensions, considering that our main goal is to listen to programmers through comments. As studying these 2 dimensions requires manual examination of comments, it is performed on the 1050 randomly sampled comments. The other dimensions are studied automatically to some extent using our tools on all comments.

The *Content* dimension contains multiple levels of categories, but only the top level categories are shown in Figure 1. Due to space constraints, we only discuss some subcategories of each top level category in this paper. *Type* encompasses many subcategories including *Unit*, *IntRange*, and *BitsBytes*, whose definitions and comment examples will be given later in §4.1. *Interface* contains subcategories such as *ErrorReturn* (§4.1) and *ContextLock* (§4.4). *Code Relationship* consists of subcategories including *DataFlow* and *ControlFlow* (§4.2). *PastFuture* encompasses subcategories such as *TODO* (§4.3). *Meta* comments describe copyright notices, authors, etc. *Explanation* comments are all other comments that cannot be classified into the 5 top level categories mentioned above. A detailed description of our content taxonomy and many comment examples are available on our website `http://opera.cs.uiuc.edu/CComment/`.

4 Comment Content (What)

In this section, we will discuss our findings based on comment content. Although addressing the concerns reflected by programmers' comments is out of the scope of this paper, we try to suggest how a tool/language designer or a system programmer can benefit from these comments at the end of each subsection. While some of the concerns may be known, our results confirm and reinforce them.

We found that 52.6±2.9% or roughly 736,109 of the comments in the three OSs belong to the four top level content categories, *Type*, *Interface*, *Code Relationship*, and *PastFuture*. We call them *exploitable* comments because they can potentially be used by existing work (i.e., annotation languages, bug detection tools, editor features, and programming languages), or inspire new work. We will use examples from these categories to illustrate this point later in this section.

As we are mainly motivated to improve software reliability, this study may be biased towards identifying comments that could help build more reliable software. For example, although *Explanation* and *Meta* comments can be useful for documentation and software maintenance, we do not consider them as exploitable comments, as they are less useful for avoiding or detecting bugs.

All percentages in this section are relative to the exploitable comments unless specified otherwise. Note that a small 1% of the exploitable comments represents roughly 7,400 comments in the three OSs altogether.

4.1 Integers and Integer Macros

Finding 1: 22.1% of the exploitable comments describe the usage and meaning of integers and integer macros, which are used to represent restricted data types such as bitsets, IO ports, and units. Each of such restricted data types has its constraint, for example, one should not write to read-only ports and values in different units should not be added directly. Since such constraints are not automatically assured by current compilers, bugs can be introduced.

We present a few important categories of comments related to integers and macros:

- **BitsBytes (7.1%):**

```
1. #define E1000_MCC   0x0401C  /* Multiple Collision
                                    Count - R/clr */
   #define E1000_RCTL 0x00100  /* Rx Control - RW */
2. #define IXGB_GPTCL  0x02108 /* Good Packets
                           Transmitted Count (Low) */
   #define IXGB_GPTCH 0x0210C /* Good Packets
                           Transmitted Count (High) */
3. #define EMU_DOCK_MINOR_REV  0x26 /* 0000xxx
                    3 bit Audio Dock FPGA Minor rev */
   #define EMU_DOCK_BOARD_ID   0x27 /* 00000xx
                           2 bits Audio Dock ID pins */
```

  An OS must perform low-level operations at the bit and byte level, e.g., when interacting with devices through IO ports. In the example above, comments are used to explain the usage of a port and the value that can be retrieved from that port. The first set of comments specify whether the port is read-only (R) or not (RW). These read-only ports can be an argument to register read functions as in `E1000_READ_REG(hw, E1000_MCC)` but not to register write functions such as `E1000_WRITE_REG-(E1000_RCTL)`. The second and third sets of comments explain respectively how to combine the result from multiple ports (low bits and high bits) to construct a complete value and which bits retrieved from a port should be used.

  Programmers must pay special attention to using these macros in the right context as the C type-checker cannot detect, for instance, code using the wrong bits. The need to specify and type-check low-level interactions has already been recognized by Merillon et al. [22] with the domain specific languages Devil. However, such domain specific languages are not adopted by programmers.

- **ErrorReturn (4.7%):**

```
/* return 1 if ACK, 0 if NAK, -1 if error. */
static int slhci_transaction(...) { ... }
```

  Many comments describe certain return values that indicate errors that a function can encounter. As different functions use different conventions (an error can be indicated by a negative value, zero, or a positive value), there is no assurance that the call-sites follow the same conventions. Explicit exception handling has already been incorporated in higher level languages such as C++ and Java, which can help to address this problem. Nevertheless, our results show that C++ and Java code still contain such error return related comments (Section 7).

- **Unit (4.7%):**

```
1. xge_os_mdelay(50); // wait 50 milliseconds
2. int mem; /* memory in 128 MB units */
```

  Another kind of recurring comments specifies the unit of a type. For example, the comments above indicate that the time is in milliseconds, and `mem` is in a unit of 128 MB. Confusing different units (second, millisecond, byte, kilobyte, etc.) can cause software bugs. As C and many other programming languages (except the latest version of Microsoft F#) do not provide different types for different units, programmers can only use more general types such as `int` for these different units.

- **Bitset (1.3%):**

```
1. /* Per process flags */
   #define PF_ALIGNWARN 0x00000001
   #define PF_STARTING  0x00000002
   #define PF_EXITING   0x00000004
2. /* Clock flags */
   #define RATE_CKCTL     (1 << 0)
   #define RATE_FIXED     (1 << 1)
   #define RATE_PROPAGATES     (1 << 2)
```

  A common idiom used in C, the bitset, allows encoding a set via an integer. Each bit of this integer represents a different element where different elements are usually represented via different macros as shown above. The comments introduce the name of a new bitset, which in some way introduce a new type. As OS code uses many such bitsets, it introduces many such macros. Unfortunately the C type checker can not detect, for instance, if a macro from one bitset is misused, e.g., combined with a macro from a different bitset.

- **IntRange (1.3%):**

```
short charheight; /* lines per char (1-32) */
```

  Surprisingly, although buffer ranges are the focus of many existing annotation languages, a similar concern about integer ranges, which appears as often in comments, is *not* covered by existing annotation languages. One possible reason is that buffer overflows are considered more important as they have been a dominant cause of security violations. The ADA programming language directly supports the integer range constraints.

Implication 1: Bug detection tools and annotation languages, which currently focus on pointers and memory bugs, should also pay attention to integers and macros.

4.2 Particular Code Relationships

Finding 2: 16.8% of the exploitable comments specify or emphasize some particular code relationships such as which function is responsible for filling a specific variable or in what order a group of functions should be invoked.

- **DataFlow & Reserved (4.3%):**

```
1. timeout_id_t msd_timeout_id; /*id returned by
                                    timeout() */
2. bool vdev_nowritecache; /* true if flushwritecache
                                    failed */
3. /* The argument to the periodic handler is not
    currently used, but is reserved for future */
```

  Some comments express data flow relationships, such as how data or events can affect the value of other data, or the lack of such relationships. In the last comment above, if the programmer tries to read the argument, the expression evaluates to junk which can cause unexpected behavior. Two annotation languages for C, SAL [2] and Splint [11], can express some of those concerns and perform source code checking against these concerns.

- **ControlFlow (5.4%):**

```
1. default: /* not reached */
2. case 10: /* FALLTHROUGH */;
2. /* Called from mmioctl_page_retire ... */
3. /* Call scsi_free before mem_free ... */
```

Some comments express control flow related concerns, such as (1) a piece of code is not reachable; (2) a missing `break` statement is intended; (3) which function should be the caller of another function; and (4) in what order a group of functions should be invoked.

As it is a common mistake to forget to add a `break` statement after a `case` statement, Splint and Lint [14] by default report any missing `break` statement after `case`. Developers can add the `FALLTHROUGH` annotation to suppress the warnings when a `break` is indeed not wanted.

Implication 2: These data flow, control flow, or other correlation based comments could be leveraged to check for bugs and also to help system programmers navigate code more easily. For example, while programmers should pay attention to such relationship comments, IDEs and tools designed to facilitate code navigation [15, 30] may want to exploit such information in comments to provide better navigation capabilities. They could even eliminate the need for such navigations by pro-actively anticipating what correlated code a programmer would like to read together.

4.3 Software Evolution

Finding 3: 5.6% of the exploitable comments describe code evolution aspects such as TODOs, deprecated code, or cloned code (copy-pasted), which can be leveraged to check for various consistency issues.

- **Clone (1.1%):**

```
/* the logic of the fast path is duplicated from this
   function. */
```

Many comments such as the example above are about code segments that achieve similar functionality but are scattered throughout program. If a bug is identified in a code segment, all of its clones should also be examined for similar bugs.

- **TODO and FIXME (3.6%):**

```
/* XXX - most fields in ki_rusage_ch are not (yet)
   filled in */
```

Typically, programmers use special names, such as `XXX`, `TODO`, and `FIXME` to indicate to-do tasks. While developers can search for the special tags to collect and complete the tasks listed in comments, they can easily forget to do so, which can introduce bugs.

Implication 3: Tool/language designers should provide better support for software evolution, either to ensure copy-pasted code is updated consistently, deprecated code is not used, or some TODO tasks are completed before software release. On the other hand, OS developers may want to use existing tools [20] to check for some of the cloned-code related inconsistencies.

4.4 Locking Specification

Finding 4: There is a significant number of comments (4.7%) related to locks and synchronizations, which almost no existing annotation languages except the closed proprietary tool Lock_Lint [23] can express and check.

- **ContextLock (3.3%):**

```
/* ... Assumes: tq->tq_lock is held. */
static void taskq_ent_free(...)
```

Many comments about the lock related context of a function are similar to the one shown above. These comments can be expressed by Lock_Lint annotations, e.g., the example above can be expressed by placing annotation `MUTEX_HELD(&tq->tq_lock)` in function `taskq_ent_free()`.

- **LockVarProtection (1.4%):**

```
/* Locking key to struct socket:
 * (a) constant after allocation, no locking required.
 ...
 * (h) locked by global mutex so_global_mtx. ... */
struct socket { ...
  short so_type; /* (a) generic type, see socket.h */
  ...
  so_gen_t so_gencnt; /* (h) generation count */ ... }
```

Some other lock-related comments indicate which lock is used to protect which shared variable. To express comment `(h)` above, one can use the Lock_Lint annotation `MUTEX_PROTECTS_DATA(so_global_mtx, so_gencnt)` after the struct definition.

Although many lock-related comments can be expressed by Lock_Lint, OpenSolaris still contain as many lock-related comments as Linux and FreeBSD, showing that Lock_Lint is not even fully utilized by its own creator (Sun).

Implication 4: As concurrent code is error-prone and difficult to debug and maintain, and as concurrency bugs may become more severe in the future due to the popularity of multicore machines, it might be beneficial for language designers to design some easy-to-use annotation languages for synchronization related concerns. To speed up adoption, annotation language designers may want to work together with system programmers to design annotation languages that handle most of their programming needs.

4.5 Other Categories

- **Memory (2.2%):**

```
1. /* param wrch (may be NULL) */
2. /* AmlBufferLength - Size of AmlBuffer */
3. /* The caller will free mp */
```

Many comments express memory related concerns. (1) a pointer may be null; (2) a buffer must be within a certain bound; or (3) which pointer/function is responsible for deallocating the buffer storage. These issues are addressed by existing annotation languages, including SAL and Splint. For example, one can add the Splint tag `owned` to a pointer to indicate that the pointed storage should be freed through this pointer. Such comments can help detect double free bugs and memory leaks. Similar concerns apply to other resources such as file handles that must also be released.

- **Context (3.4%):**

```
1. /* Assumes that SGE is stopped and all interrupts
      are disabled. */
2. /* permission checks will be done by the caller */
```

The context of a function call is important for program correctness. Besides lock-related context, other context-related concerns are also commonly expressed in comments such as (1) interrupt-related context, and (2) security-related context. Many of these comments can be seen as assertions on the structure of the program (e.g., its call graph), not assertions on the value of variables. Some software evolution related comments can be seen as assertions on the source code history of the program.

- **ByteAddress:(2.5%)**

```
struct audio1575_audio_regs { ...
    uint8_t   micilviv_reg;  /* 65h - 65h */
    uint16_t  micisr_reg;    /* 66h - 67h */
    uint16_t  micipicb_reg;  /* 68h - 69h */ ... }
```

OS code contains many comments about the precise byte layout of large structures such as the one above. Such layouts about devices, network protocols, file systems, etc. are specified in external documents. To follow the specification, programmers have to compute the exact number of bytes and use the right type to declare the storage for each field, e.g., `66h-67h` is 2 bytes therefore it should use type `uint16_t`. It would be better if a tool can check the consistency between the types and the comments.

The description of other categories can be found on our website `http://opera.cs.uiuc.edu/CComment/`.

5 Annotation Languages

Finding 5: At least 10.7% of the exploitable comments can be expressed via existing annotation languages (ignoring the easy-to-use aspect).

Implication 5: There is a great potential of automatically converting these comments into existing formal annotations for automatic bug detection. Such approach could help developers to see the benefits of annotation languages without paying the overhead of writing the annotations first.

Although annotation languages have been recognized as a promising way to improve software reliability, they have not been widely adopted yet. To help developers transition to the era of writing annotations, we can help in two aspects: (1) Currently, developers can not experience the benefits of annotation languages without first learning the annotation languages and spending extra effort to write annotations. If we can use comments to suggest annotations, developer can learn by example and could be more likely to write annotations directly in the future. Therefore, as a first step, we study what and how many comments are already covered by existing annotation languages to show the potential of converting the comments into existing annotation languages. (2) As many annotation languages are available, it may be difficult for developers to know all of them. By studying both existing annotation languages and comments to learn programmers' needs, we hope to provide guidance on developers' selection of annotation languages.

Category	%	Supporting AnnoLang (partially or fully)
Lock	44.1	Lock_Lint
Memory	20.3	Deputy, SAL, Splint
Reserved	10.2	SAL, Splint
Control	6.8	SAL, Splint, Lock_Lint
Other	22.0	Deputy, SAL, Splint

(a) Distribution

AnnoLang	#	%
Splint	30	50.8
Lock_Lint	27	45.8
SAL	19	32.2
Deputy	12	20.3
Sparse	2	3.4
Total	59	/

(b) Coverage

Table 2. Distribution of Annotation Convertible Comments and Coverage of Annotation Languages. AnnoLang denotes annotation language.

Specifically, we conducted a brief survey of five major annotations languages supporting C, i.e., Microsoft SAL [2], Linux's Sparse [34], Sun's Lock_Lint [23], Deputy [6], and Splint [11]. We then studied how many of the manually sampled comments can be expressed by these annotation languages.

Comments that can be converted to existing annotations, called *annotation convertible comments*, make up 10.7% of the exploitable comments. We showed how some of the comments can be expressed in a particular annotation language in earlier sections such as bullet *ControlFlow* in §4.2, bullets *ContextLock* and *LockVarProtection* in §4.4.

The category distribution is shown in Table 2(a). These categories are aggregation of different subcategories. For example, *Lock* is lock related comments including subcategories *ContextLock* and *LockVarProtection* (§4.4). *Memory* includes comments specifying null pointers, buffer bounds and buffer ownership (bullet *Memory* in §4.5). *Reserved* comments specify whether a variable is used or not (example 3 in bullet *DataFlow & Reserved* in §4.2). *Control* denotes certain control flow related comments such as comment 1 & 2 in bullet *ControlFlow* in §4.2. The percentage numbers are relative to the annotation convertible comments. The sum is not 100% because one block comment can belong to multiple categories.

In addition, many important concerns are not or poorly supported by existing annotations, including bitsets, units,

	Type	Inter-face	Relati-onship	Past-Future	Meta	Expla-nation	Total	Exploit-able%
L	54	49	93	19	17	145	377	55.7%
F	34	46	97	21	5	166	369	51.7%
O	31	46	107	8	5	172	369	50.3%

Table 3. Comparison of Comment Content Distribution. L, F, O are Linux, FreeBSD, OpenSolaris Respectively. Exploitable% is the percentage of exploitable comments.

integer value ranges, code relationships, cloned code, calling context, and byte addresses. It would be beneficial to design annotation languages for these concerns.

Table 2(b) shows that Splint has the best coverage (50.8%) of the convertible comments. Sparse has a coverage of only 3.4%, probably for two reasons: (1) the expressiveness of Sparse is limited and (2) developer seldom write comments for concerns that Sparse annotations can express (e.g., address space modifier `__user`).

6 Where, When, and Who

In this section, we briefly study the remaining comment dimensions: *where* are the comments, *when* the comments were written, and *who* wrote them. There are many possible statistics to obtain for these dimensions. However, as our goal in this paper is to listen to programmers through comments, we focus on questions and statistics that may invalidate our findings such as: Are the findings general to all the three OSs? Are the studied comments recent enough or do they correspond to obsolete needs? Have we actually listened to many programmers? Although using version control systems (e.g., command *cvs annotate*) has some known caveats [16] and the information extraction process explained in Section 2 uses some heuristics, the resulting approximations are good enough for our purpose as our goal is only to check the validity of our findings.

6.1 Where

Finding 6: OpenSolaris, started as closed software, does not exhibit much different comment characteristics from its open source counterparts.

Implication 6: Programmers' need in communicating information via comments is not affected by the software development paradigm (i.e., open source or closed source). This also shows that our findings about programmers' needs as reflected in their comments are general across OS code.

Table 3 shows that the distribution of comment content in different OSs is almost identical. The total number of labels for an OS is slightly bigger than 350 because a few comments have multiple labels (7.4% of the entire sample have multiple labels and on average each comment has 1.09 labels). For our Finding 1 to 5, the individual percentage numbers for each OS are 17.6-26.7%, 14.9-19.5%, 4.0-8.3%, 3.4-5.5%, and 8.2-12.7%, which are very similar. Figure 2(a) shows the distribution of comment locations in the code. We also found that many comments are similar regardless of the functionality of the code (e.g., drivers, file systems, or core kernel code). As the three OSs share some code, they also share some comments which may invalidate our claim of generality. Therefore, we conducted a comment clone detection experiment, and found that the pairwise *comment* sharing between each pair of OSs projects is on average less than 7.5%. Therefore, such a small sharing should not threat the validity or generality of our findings.

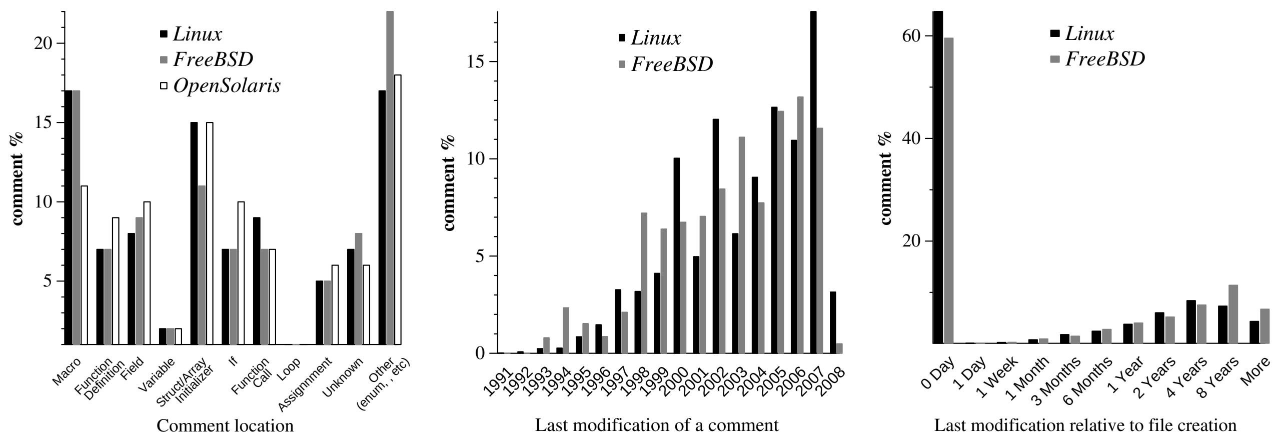
By zooming in to finer-grained categories, we found that certain types of comments occur more often in specific subsystems. For example, *BitsBytes* comments appear more often in drivers and network protocol code as both require more low-level data manipulation. This phenomenon suggests maybe that some specialized training can be provided to the different OS programmers to focus on their most relevant needs, for example, to teach them the most relevant annotations.

6.2 When

For Linux and FreeBSD, whose evolution information is available from 1991 and 1994 respectively, the number of comments continually increases over the years. Figure 2(b) and 2(c) show the absolute time of comments' last modification (e.g., 5 years ago), and the time relative to the file (e.g., 2 months after a file is created or imported in the repository). The former can give a hint about whether a type of comments is still relevant today, the latter can suggest the development phase (e.g., design, maintenance) when the comment was written. We found that most categories in our taxonomy have comments in the sample that were written in the last 5 years, indicating that programmers still use comments to express their needs and those needs have not yet been fully fulfilled by the tools they know.

We can see from Figure 2(c) that most comments were never modified after the file containing them was created or imported in the repository. To understand why comments were modified or added to an existing file in the other cases, we studied the last patch that modified or added a comment for our sample comments (using information derived from `cvs annotate`). We found that most of these comment modifications were either simple fixes of indentation problems and typos, addition of new comments introduced together with new code, or movement of existing comments and code from another file. Only a few comment modifications were due to programmers' desire to document a bug fix or add more information to the software.

Outdated comments: Inevitably, some comments we studied may be out of date. However, the outdated comments were usually correct at the time when they were written, therefore, these comments still express programmers' real needs.



(a) Location (b) Absolute Age (c) Relative Age

Figure 2. Distribution of Comment Location, Absolute Age and Relative Age. All comments, not only the samples, are included. Note that year 2008 only contains less than 2 months' data.

6.3 Who

By studying our 1050 comment sample, we have listened to 309 different OS programmers, according to the version control systems. We found that well-known experts including Linus Torvalds also use comments to express needs that can not be fulfilled by the language or the tools they use.

7 Comment Study on Non-OS Code

Although this study focused on comments in operating systems, which are written in C, we have also tried to study how tied to OS code and C our results are. To understand comments in other types of software, written in other programming languages, we performed a preliminary comment study using 1050 comments from three other open source code bases (350 from each): MySQL (a database server), Firefox (a web browser) and Eclipse (an IDE). Eclipse is written in Java while MySQL and Firefox are mostly a mix of C and C++ (with a little of Java). We also extended our tools to handle C++ and Java code.

Software	Com%	Exp%	F2	F3	F5	F4	F1
OS	23.1-29.7%	52.6±2.9%	16.8%	5.6%	10.7%	4.7%	22.1%
non-OS	21.8-28.5%	57.5±2.9%	21.2%	3.8%	13.7%	1.7%	6.3%

Table 4. OS and non-OS comparison. Com% is the percentage of comments in software. Exp% is the percentage of exploitable comments. F stands for Finding.

Table 4 summarizes the similarities and differences between OS and non-OS code. For the 3 non-OS code bases, 21.8-28.5% of the lines are comments. By studying the 1050 randomly sampled non-OS comments, we found that 57.5±2.9% are exploitable comments. Among the exploitable comments, 21.2% are about particular code relationships (compared to Finding 2), 3.8% are code evolution related comments (compared to Finding 3), and 13.7% can be expressed by the 5 annotation languages we studied if they were extended to process C++ and Java code (compared to Finding 5). These results are similar to the corresponding numbers for OS code, which shows that Findings 2, 3 and 5 seem true regardless of the type of software or the programming language used.

There are still some lock/synchronization related comments (especially in MySQL), although the percentage (1.7%) is smaller than OS's (4.7% shown in Finding 4). This difference is understandable as OSs inherently have to deal with concurrent activities and shared resources, and OS developers pay special attention to the correctness of synchronization.

Non-OS code contains much fewer comments explaining integers and integer macros (only 6.3% as opposed to 22.1% in Finding 1). Indeed, some C++ features, such as templates, the `inline` and `const` keywords, were invented to avoid using C macros. Additionally, OS code needs to handle many low-level bit and byte related operations which are less common in desktop applications and servers.

Compared to an OS, MySQL and Firefox contain far more comments about whether a pointer must be considered as an input parameter, an output parameter, or both (called In/Out comments). Such comments could be expressed by annotation languages such as Splint. Developers could also use the `const` keyword instead of using comments in some cases. Developers often choose not to, perhaps because the const type checking is not flexible enough. We found almost no In/Out comments in Eclipse. This is probably because Eclipse uses clearer method interfaces where parameters are the input and the return of a method is the output in most cases. Such clear interfaces are made possible because of the automatic memory management, the more consistent use of exceptions, and object-oriented programming.

Eclipse contains 4-16 times more comments discussing whether a reference can be null or not than in an OS. This is probably because null pointer dereference is one of the few types of memory related bugs that still exist in Java, leading to a focus by Java programmers on such errors. Other than null related comments, we have not found many memory management related comments. However, even if memory management is easier in Java, the more general problem of resource management (such as who must close/dispose a stream) is still present in Java.

Many comments in Eclipse (dominantly Javadoc comments) are about error management, either explaining the meaning of boolean and integer return values, or describing the possible exceptions thrown by a method. The C++ part of Firefox does not always make use of exceptions and still relies on integers and comments about error code conventions for error management.

Eclipse contains 2 new types of comments: links to Bugzilla databases bugs and non-externalized string markers. MySQL contains a few comments that can be used by its internal coverage and patch validation tools. We found almost no comments specific to object oriented programming or design patterns.

Although the MySQL and Firefox projects started with C++, their code bases still contain a significant amount of C code. Much of the C code is in externally developed C libraries that are included directly in the repository. C is probably preferred over C++ for writing libraries, as it is easier for other languages (e.g., Python) to access C libraries. As libraries are important for code reuse, C may still be a popular language for some time.

8 Related Work

We briefly discuss previous comment studies and tools using comments.

Comment studies Comment studies in the 80's and 90's were concerned with the usefulness of comments for program understanding [36] and ratio metrics between code and comments [29], under the assumption that software quality is better if the code is more commented. More recent work studied the evolution of comments during the software life-cycle [13] and metrics about the co-evolution of code and comments [12, 35]. Marin studied psychological factors that may push programmers to comment, e.g., whether already commented code is an incentive for programmers to comment more on their own modifications to the code [21]. Etzkorn *et al.* [10] used link grammar to parse comments to help program understanding. Ying *et al.* [37] and Storey *et al.* [31] studied a specific kind of comments, "TODO" comments. None of the previous work comprehensively studied the content or the semantics of comments. Moreover, they did not quantify the use of different kinds of comments that we studied.

iComment & Find-Concept Our previous work, iComment [33], leveraged existing comments to detect bugs and bad comments. Different from iComment, the goal of this paper is *not* to detect bugs or bad comments. Instead, this comprehensive comment study provides guidance and insights on various aspects of improving software reliability. To help discover major comment topics, iComment automatically computed most frequently used keywords in comments. However, there was no manual study of these comments. As keywords alone are not enough to judge the usefulness of comments, iComment could not answer how much comments are useful for improving software reliability or the other questions this work addressed. Find-Concept [26] applied natural language processing techniques to find word paraphrases to expand code search. This work did not study the characteristics of comment content.

Annotation languages Many annotation languages have been proposed to extend the C type system [2, 6, 11, 14, 23, 34], to specify locking requirements [23, 34], to annotate function interfaces [2, 11, 34], or to mark control flows [2, 11]. While some comments informally express properties that could be described via existing annotations, no previous work quantified the amount of such comments. Moreover, previous studies did not leverage existing comments to discover and motivate new types of annotations.

Automatic documentation generation Literate programming [17] proposes to embed code inside documentation to produce "literature" instead of embedding comments and documentation in the code. Javadoc [19] let programmers use special *tags* to document functions or data structures. Those tags are later processed by a tool to automatically produce hypertext documentation. While these tools help developers write better documents, previous work did not learn programmers' needs by studying comments.

9 Conclusions

We have argued that many comments are written when programmers cannot find another easy way in the languages or tools they know to express their intentions. Studying comments can thus be a way to listen to programmers' needs. Using 1050 comments randomly sampled from the latest versions of Linux, FreeBSD, and OpenSolaris, we have shown that at least 52.6% of these comments express needs that could be leveraged by existing or future languages and tools, revealing opportunities to improve software reliability and increase programmer productivity.

We hope this study is of interest to tool/language designers and system programmers, and inspires other researchers to perform similar comment content study on other software, from other communities, in different programming languages, to discover different programmers' needs and other limitations of tools or languages. Finally, we hope people will look differently at the nature of comments the next time they see or write comments.

10 Acknowledgments

We greatly appreciate Julia Lawall, Darko Marinov, and the anonymous reviewers for their invaluable feedback and comments. This research is supported by NSF CNS-0720743 grant, NSF CNS-0615372 grant, NSF CNS-0347854 (career award), DOE Early Career Award DE-FG02-05ER25688, and Intel gift grants.

Availability

Our full comment taxonomies, classified samples, and the tools we developed for this study are available on our web page: `http://opera.cs.uiuc.edu/CComment/`.

References

[1] Human interactions in programming. http://research.microsoft.com/hip/.

[2] MSDN run-time library reference – SAL annotations. http://msdn2.microsoft.com/en-us/library/ms235402.aspx.

[3] Natural programming project. http://www.cs.cmu.edu/~NatProg/.

[4] J. Buckley, T. Mens, M. Zenger, A. Rashid, and G. Kniesel. Towards a taxonomy of software change. *Journal of Software Maintenance and Evolution: Research and Practice*, pages 309–332, 2005.

[5] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. R. Engler. An empirical study of operating system errors. In *SOSP*, 2001.

[6] J. Condit, M. Harren, Z. R. Anderson, D. Gay, and G. C. Necula. Dependent types for low-level programming. In *ESOP*, pages 520–535, 2007.

[7] R. Cox, T. Bergan, A. Clements, F. Kaashoek, and E. Kohler. Xoc, an extension-oriented compiler for systems programming. In *ASPLOS*, 2008.

[8] R. DeLine, M. Czerwinski, B. Meyers, G. Venolia, S. Drucker, and G. Robertson. Code thumbnails: Using spatial memory to navigate source code. In *VLHCC*, 2006.

[9] D. R. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *SOSP*, 2001.

[10] L. H. Etzkorn, L. L. Bowen, and C. G. Davis. An approach to program understanding by natural language understanding. *Nat. Lang. Eng.*, 5(3):219–236, 1999.

[11] D. Evans. Static detection of dynamic memory errors. In *PLDI*, pages 44–53, 1996.

[12] B. Fluri, M. Wursch, and H. C. Gall. Do code and comments co-evolve? on the relation between source code and comment changes. In *Working Conference on Reverse Engineering*, pages 70–79, 2007.

[13] Z. M. Jiang and A. E. Hassan. Examining the evolution of code comments in PostgreSQL. In *MSR*, 2006.

[14] S. Johnson. Lint, a c program checker, 1978.

[15] M. Kersten and G. C. Murphy. Using task context to improve programmer productivity. In *FSE*, 2006.

[16] S. Kim, T. Zimmermann, K. Pan, and E. J. J. Whitehead. Automatic identification of bug-introducing changes. In *ASE*, pages 81–90, 2006.

[17] D. E. Knuth. Literate programming. *Computer Journal*, 27(2), 1984.

[18] A. J. Ko, H. Aung, and B. A. Myers. Eliciting design requirements for maintenance-oriented IDEs: a detailed study of corrective and perfective maintenance tasks. In *ICSE*, pages 126–135, 2005.

[19] D. Kramer. Api documentation from source code comments: a case study of javadoc. In *International Conference on Computer Documentation*, 1999.

[20] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: A tool for finding copy-paste and related bugs in operating system code. In *OSDI*, 2004.

[21] D. P. Marin. What motivates programmers to comment? Research report UCB/EECS-2005-18, University of California Berkeley, November 2005.

[22] F. Mérillon, L. Réveillère, C. Consel, R. Marlet, and G. Muller. Devil: An IDL for hardware programming. In *OSDI*, 2000.

[23] S. Microsystems. Lock_Lint - Static data race and deadlock detection tool for C. http://developers.sun.com/sunstudio/articles/locklint.html.

[24] T. Ostrand and E. Weyuker. The distribution of faults in a large industrial software system. In *ISSTA*, 2002.

[25] P. C. Rigby and A. E. Hassan. What can OSS mailing lists tell us? A preliminary psychometric text analysis of the apache developer mailing list. In *MSR*, 2007.

[26] D. Shepherd, Z. P. Fry, E. Hill, L. Pollock, and K. Vijay-Shanker. Using natural language program analysis to locate and understand action-oriented concerns. In *AOSD*, 2007.

[27] O. Spinczyk, A. Gal, and W. Schroder-Preikschat. AspectC++: An aspect-oriented extension to C++. In *Proc. of the 40th Intern. Conf. on Technology of Object-Oriented Languages and Systems*, 2002.

[28] D. Spinellis. A tale of four kernels. In *ICSE*, 2008.

[29] I. Stamelos, L. Angelis, A. Oikonomou, and G. L. Bleris. Code quality analysis in open source software development. *Information Systems Journal*, 2002.

[30] M.-A. Storey, L.-T. Cheng, J. Singer, M. Muller, D. Myers, and J. Ryall. How programmers can turn comments into waypoints for code navigation. In *ICSM*, 2007.

[31] M.-A. Storey, J. Ryall, R. I. Bull, D. Myers, and J. Singer. TODO or to bug: Exploring how task annotations play a role in the work practices of software developers. In *ICSE*, 2008.

[32] M. Sullivan and R. Chillarege. A comparison of software defects in database management systems and operating systems. In *22nd Annual International Symposium on Fault-Tolerant Computing*, 1992.

[33] L. Tan, D. Yuan, G. Krishna, and Y. Zhou. /* iComment: Bugs or bad comments? */. In *SOSP*, 2007.

[34] L. Torvalds. Sparse - A semantic parser for C. http://www.kernel.org/pub/software/devel/sparse/.

[35] R. Warren. Understanding software evolution through comment analysis. Research report, U. of Waterloo, 2002.

[36] S. N. Woodfield, H. E. Dunsmore, and V. Y. Shen. The effect of modularization and comments on program comprehension. In *ICSE*, 1981.

[37] A. T. T. Ying, J. L. Wright, and S. Abrams. Source code that talks: An exploration of eclipse task comments and their implication to repository mining. In *MSR*, 2005.