Diagnosys: Automatic Generation of a Debugging Interface to the Linux Kernel

Tegawendé F. Bissyandé,
Laurent Réveillère
University of Bordeaux, France
{bissyande,reveillere}@labri.fr

Julia L. Lawall,
Gilles Muller
INRIA/LIP6-Regal, France
{Julia.Lawall,Gilles.Muller}@lip6.fr

ABSTRACT

The Linux kernel does not export a stable, well-defined kernel interface, complicating the development of kernel-level services, such as device drivers and file systems. While there does exist a set of functions that are exported to external modules, this set of functions frequently changes, and the functions have implicit, ill-documented preconditions. No specific debugging support is provided.

We present *Diagnosys*, an approach to automatically constructing a debugging interface for the Linux kernel. First, a designated kernel maintainer uses Diagnosys to identify constraints on the use of the exported functions. Based on this information, developers of kernel services can then use Diagnosys to generate a debugging interface specialized to their code. When a service including this interface is tested, it records information about potential problems. This information is preserved following a kernel crash or hang. Our experiments show that the generated debugging interface provides useful log information and incurs a low performance penalty.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids*; D.4.5 [**Operating Systems**]: Reliability

General Terms

Design, Experimentation, Reliability

Keywords

Diagnosys, Debugging, Wrappers, Linux, Device drivers

1. INTRODUCTION

Debugging is difficult. And debugging an operating system kernel-level service, such as a device driver, file system, or network protocol, is even more difficult. When a crash occurs, the service developer is presented with a backtrace, containing the location of the instruction that caused the crash and the pending return pointers on the stack. This information may be unreliable or incomplete. Even when the backtrace information is present and correct, it does not capture context information such as the values of local variables and the effect of recent decisions that are often essential to identify the problem. Indeed, kernel service code contains many execution paths, taking conditions from the operating environment into account, and is difficult to test deterministically. Support is needed for providing more information at the time of the crash, without introducing a substantial performance penalty or imposing an additional burden on the developer.

As Linux is becoming more and more widely used, in platforms ranging from embedded systems to supercomputers, there is an increasing interest from third-party developers, having little expertise in Linux internals, in developing new Linux kernel services. Such services must integrate with the Linux kernel via the various kernel-level APIs. Developing code at this level is a challenging task. Indeed, the Linux kernel development process is based on the assumption that the source code of all kernel-level services is available within the publicly available kernel source tree, and thus kernel APIs are, for efficiency, only as robust as required by their internal client services. Furthermore, kernel developers can freely adjust the kernel APIs, as long as they are willing to update all of the affected service code. The kernel implementation is thus, by design, maximally efficient and evolvable, enabling it to rapidly meet new performance requirements, address security issues, and accommodate new functionalities. But these assumptions complicate the task of the developers of new services who require more safety and help in debugging. Advances in bug-finding tools [3, 18, 20], specialized testing techniques [17, 21], and code generation from specifications [30] have eased but not yet fully solved these difficulties. Current approaches put substantial demands on the developer, both to learn how to use the approach and to effectively integrate it into his development process.

We concretize the difficulty confronting a Linux service developer in interacting with the Linux kernel as the notion of a *safety hole*. We define a safety hole as a fragment of code that introduces the potential for a fault to occur in the interaction between a kernel-level service and the rest of the kernel. For example, code in the definition of a kernel API function that dereferences a parameter without testing its value represents a safety hole, because a service could invoke the function with `NULL` as the corresponding argument. Likewise, code in the definition of a kernel internal

ASE '12, September 3–7, 2012, Essen, Germany

API function that returns NULL as the result represents a safety hole, because a calling service could dereference this result without checking its value.

To address the problem of safety holes in Linux kernel internal API functions, we propose an approach, named *Diagnosys*, that automatically generates a debugging interface to the Linux kernel tailored for a particular kernel-level service under development, based on a prior static analysis of the Linux kernel source code. This debugging interface amounts to a wrapper on the kernel exported functions, that logs information about potentially dangerous uses of these functions. Localizing the interface in this way, at the boundary of the interaction between the service and the OS kernel, ensures that the feedback provided by the interface is in terms of the code that the developer has written, and is thus expected to be familiar with. Because the interface is only visible to the service, it has no impact on the performance of code within the kernel, even code that uses functions that contain safety holes. When the service executes, the interface generates log messages whenever service code invokes a kernel API function containing a safety hole in a potentially risky way. Such a debugging interface requires no manual intervention from the service developer until there is a crash or hang, and is thus well-suited to intensive service development, when the developer is modifying the code frequently, and bugs are likewise frequent. Because the debugging interface is automatically generated, it can be regenerated for each new version of the Linux kernel, as the properties of the kernel APIs change.

Diagnosys is composed of two tools: SHAna (Safety Hole Analyzer), which statically analyzes the kernel source code to identify safety holes in the definitions of the kernel exported functions, and DIGen (Debugging Interface Generator), which uses the information about the identified safety holes to construct a debugging interface tailored to a given service. Diagnosys also includes a runtime system, provided as a kernel patch. SHAna is run by a Linux kernel maintainer once for each Linux version, to take into account the current definitions of the Linux kernel internal API functions. DIGen is run by a service developer as part of the service compilation process. During the execution of the resulting service, the debugging interface uses the runtime system to log information in a crash-resilient buffer about any unsafe uses of functions containing safety holes. On a kernel crash or hang, the service developer can subsequently consult the buffer to obtain the logged information.

The main contributions of this paper are as follows:

- We identify the interface of kernel exported functions as a sweet spot at which it is possible to interpose the generation of debugging information, in a way that improves debuggability but does not introduce an excessive runtime overhead.
- We identify safety holes as a significant problem in the interface between a service and the kernel. Indeed, of the 703 Linux 2.6 commits for which the changelog refers explicitly to a function exported in Linux 2.6.32, 38% corrected faults that are related to one of our identified safety holes.
- We propose an approach to allow a service developer to seamlessly generate, integrate, and exploit a kernel debugging interface specialized to the service code. This approach has a low learning curve, and in particular does not require any particular Linux kernel expertise.
- Using fault-injection experiments on 10 Linux kernel services, we demonstrate the improvement in debuggability provided by our approach. We find that in 90% of the cases in which a crash occurs, the log contains information relevant to the origin of the defect, and in 95% of these cases, a message relevant to the crash is the last piece of logged information. We also find that in 93% of the cases in which a crash or hang occurs, the log information reduces the number of files that have to be consulted to find the cause of the bug.
- We show that the generated debugging interface incurs only a minimal runtime overhead on service execution, allowing it to be used up through early deployment.

The rest of this paper is organized as follows. Section 2 illustrates problems in kernel development that have been related to safety holes and gives an overview of kinds of safety holes that we take into account. Section 3 discusses the challenges in kernel debugging, focusing on crashes and hangs derived from safety holes. Section 4 presents Diagnosys, including the process of collecting information about the occurrences of safety holes and the associated preconditions, and the process of generating a debugging interface. Section 5 evaluates our approach. Finally, Section 6 discusses related work, and Section 7 concludes.

2. SAFETY HOLES

To understand the challenges posed by safety holes, we first consider some typical examples in Linux kernel internal API functions and the problems that these examples have caused, as reflected by Linux patches. Then, we present a methodology for identifying kinds of safety holes, and use this methodology to enumerate the kinds of safety holes considered in the rest of the paper. Finally, we consider how to statically identify preconditions on these safety holes, to limit the generation of log messages in the debugging interface to cases that may actually cause a crash or hang.

2.1 Examples of safety holes

Because the Linux kernel does not define a precise internal API, we focus on the set of functions that are made available to dynamically loadable kernel modules using either EXPORT_SYMBOL or EXPORT_SYMBOL_GPL. Dynamically loadable kernel modules provide a convenient means to develop new services, as they allow the service to be loaded into and removed from a running kernel for the testing of new service versions. We refer to kernel functions that are made available to such modules as *kernel exported functions*.

Fig. 1a shows an excerpt of the definition of the kernel exported function skb_put, which dereferences its first argument without first checking its value. Many kernel functions are written in this way, assuming that all arguments are valid. This code represents a safety hole, because the dereference is invalid if the corresponding argument is NULL. Such a fault occurred in Linux 2.6.18 in the file *drivers/net/forcedepth.c*. In the function nv_loopback_test, skb_put is called with its *skb* argument being the result of calling dev_alloc_skb, which can be NULL. The fix, as implemented by the patch shown in Fig. 1b, is to avoid calling skb_put in this case. skb_put remains unchanged.

```
1  unsigned char *skb_put(struct sk_buff *skb, unsigned int len)
2  {   unsigned char *tmp = skb_tail_pointer(skb);
3      SKB_LINEAR_ASSERT(skb);
4      skb->tail += len; ...
5  }
```

a) Excerpt of the definition of `skb_put`

```
1  commit 46798c897e235e71e1e9c46a5e6e9adfffd8b85d
2      tx_skb = dev_alloc_skb(pkt_len);
3  +   if (!tx_skb) { ... goto out; }
4      pkt_data = skb_put(tx_skb, pkt_len);
```

b) Excerpt of the bug fix patch

Figure 1: Bug fix of the usage of `skb_put`

Fig. 2a shows an excerpt of the definition of the kernel exported function `open_bdev_exclusive`, which returns a value constructed using the kernel function `ERR_PTR` when an error is detected. Dereferencing such a value will crash the kernel. Thus, this return statement also represents a safety hole. In Linux 2.6.32, in the file *fs/btrfs/volumes.c*, the function `btrfs_init_new_device` called `open_bdev_exclusive` and compared the result to `NULL` before dereferencing the value. This test, however, does not prevent a kernel crash, because an `ERR_PTR` value is different from `NULL`. Fig. 2b shows a patch fixing the fault.

```
1  struct block_device *open_bdev_exclusive(
2          const char *path, fmode_t mode, void *holder)
3  {
4      ...
5      return ERR_PTR(error);
6  }
```

a) Excerpt of the definition of `open_bdev_exclusive`

```
1  commit 7f59203abeaf18bf3497b308891f95a4489810ad
2      bdev = open_bdev_exclusive(...);
3  -   if (!bdev) return -EIO;
4  +   if (IS_ERR(bdev)) return PTR_ERR(bdev);
```

b) Excerpt of the bug fix patch

Figure 2: Bug fix of error handling code

In the previous cases, the safety hole is apparent in the definition of a kernel exported function. A safety hole, however, may also be interprocedural, making the danger that it poses more difficult to spot. For example, as shown in Fig. 3(a,b), the kernel exported function `kmap`, defined in *arch/x86/mm/highmem_32.c*, passes its argument to the function `page_zone` via the macro `PageHighMem`, which in turn forwards the pointer, again without ensuring its validity, to the function `page_to_nid`. This function then dereferences it, unchecked. This safety hole resulted in a fault, which was fixed by the patch shown in Fig. 3c.

```
1  void *kmap(struct page *page)
2  { might_sleep();
3    if ( !PageHighMem(page) )
4    ...
5  }
```

a) Excerpt of `kmap`

```
1  static inline int page_to_nid
2          (struct page *page) {
3    return ( page->flags >> ...)
4                & NODES_MASK;
5  }
```

b) Excerpt of `page_to_nid`

```
1  commit 649f1ee6c705aab644035a7998d7b574193a598a
2      page = read_mapping_page(...);
3  +   if (IS_ERR(page)) { ... goto out; }
4      pptr = kmap (page);
```

c) Excerpt of the bug fix patch

Figure 3: Bug fix of a use of `kmap`

2.2 Taxonomy of safety holes

As illustrated in Section 2.1, some fragments of code executed by kernel exported functions, while themselves being correct, can provoke kernel crashes or hangs when the function is used incorrectly. We distinguish between *entry* safety holes, in which the crash or hang is provoked within the execution of the kernel exported function, due to an invalid argument provided by the service, and *exit* safety holes, in which the crash or hang is provoked within the subsequent execution of the service due to a possible effect of the kernel exported function that the service has not taken into account.

As a first source of kinds of safety holes, we consider the fault kinds identified by Chou *et al.* in their 2001 study of Linux code [5]. A fault is not in itself a safety hole, because the faulty code can be completely contained within a single function definition. Likewise, a safety hole is not in itself a fault, as illustrated by the above examples. Nonetheless, we observe that many fault kinds involve multiple disjoint code fragments. When some of these fragments are present in a kernel exported function and the remainder may be present in a service implementation, we say that the kernel exported function contains a safety hole. For example, a `NULL` pointer dereference fault typically involves an initialization of a variable to `NULL` followed by a dereference of this variable. Returning `NULL` from a kernel exported function can cause it to be dereferenced in service code, and receiving `NULL` as an argument in a kernel exported function can lead to a `NULL` pointer dereference in the kernel exported function code.

These observations suggest a methodology for translating fault kinds into kinds of safety holes. When the suffix of a sequence of code fragments associated with a fault kind is found in a kernel exported function and depends in some way on the calling context, *e.g.*, via arguments of that function, then that suffix represents an entry safety hole. Likewise, when a prefix of such a sequence is found in a kernel exported function and has some impact on the function's result, then that prefix represents an exit safety hole.

Table 1 summarizes the fault kinds identified by Chou *et al.*, as well as the entry and exit safety hole kinds that we have derived from these fault kinds according to the above methodology. For example, given the above analysis of the structure of a Null fault, the corresponding entry safety hole is a dereference of an unchecked pointer parameter, while the corresponding exit safety hole is a return of a `NULL` value.

2.3 Safety hole preconditions

From a collection of safety holes, our goal is to create a debugging interface that informs the service developer of possibly dangerous uses of kernel exported functions within his code. Nevertheless, merely invoking a kernel exported function that contains an entry or exit safety hole does not necessarily cause a fault. Instead, some properties of the argument or return values, such as the presence of a `NULL` value, must typically be satisfied. Thus, we require information not just about safety holes, but also about the preconditions that must be satisfied to ensure that no fault can occur [16]. We furthermore distinguish between safety holes that are *certain*, if violation of the precondition is guaranteed to result in a crash or hang within the execution of the kernel exported function, or *possible*, if violation of the precondition may cause a crash or hang on at least one possible execution path. All exit safety holes are *possible*, as

Table 1: Categorization of common faults in Linux [5]. EF refers to the *exported function*.

Category	Actions to avoid faults	safety hole	safety hole description	Analysis type
Block	To avoid deadlock, do not call blocking functions with interrupts disabled or a spinlock held	`entry` `exit`	EF calls a blocking function (function referencing GFP_KERNEL) EF returns after disabling interrupts or while holding a lock	interprocedural intra/interprocedural
Null	Check potentially NULL/ERR_PTR pointers returned from routines	`entry` `exit`	EF dereferences an argument without checking its validity EF returns a NULL/ERR_PTR pointer	interprocedural interprocedural
Var	Do not allocate large stack variables ($> 1K$) on the fixed-size kernel stack	`entry` `exit`	EF allocates an array whose size depend on a parameter EF returns a large value	intraprocedural interprocedural
INull	Do not make inconsistent assumptions about whether a pointer is NULL/ERR_PTR	`entry` `exit`	EF dereferences an argument without checking its validity EF returns a NULL/ERR_PTR pointer	interprocedural interprocedural
Range	Always check bounds of array indices and loop bounds derived from user data	`entry` `exit`	EF uses an unchecked parameter to compute an array index EF returns a value obtained from user level	intraprocedural interprocedural
Lock	Released acquired locks; do not double-acquire locks	`entry` `exit`	EF acquires a lock derived from a parameter EF returns without releasing an acquired lock	interprocedural interprocedural
Intr	Restore disabled interrupts	`entry` `exit`	EF calls a blocking function EF returns with interrupts disabled	interprocedural intraprocedural
Free	Do not use freed memory	`entry` `exit`	EF dereferences a pointer-typed parameter value EF frees memory derived from a parameter	none interprocedural
Float	Do not use floating point in the kernel		*These fault kinds depends on local properties and are therefore*	none
Real	Do not leak memory by updating pointers with potentially NULL realloc return values		*not relevant to the interface between a service and the kernel exported functions*	none
Param	Do not dereference user pointers	`entry` `exit`	EF dereferences a pointer-typed parameter EF returns a pointer-typed value obtained from user level	none interprocedural
Size	Allocate enough memory to hold the type for which you are allocating	`entry` `exit`	EF allocates memory of a size depending on a parameter EF returns an integer value	intraprocedural none

the usage context of the function result is unknown.

3. KERNEL DEBUGGING

Each of the examples presented in Section 2 could crash the kernel. When this occurs, the kernel generates an oops report, consisting of the reason for the crash, the values of some registers and a backtrace, listing the function calls pending on the stack. Using this information in debugging raises two issues: 1) the reliability of the provided information, and 2) the relevance of the provided information to the actual fault. Debugging kernel hangs raises further issues.

Reliability of kernel oops reports. Linux kernel backtraces suffer from the problem of *stale pointers*, *i.e.*, addresses within functions that have actually already returned. To illustrate this problem, we consider a crash occurring in the function `btrfs_init_new_device` previously shown in Fig. 2. The crash occurred because the kernel exported function `open_bdev_exclusive` returns an `ERR_PTR` value in case of an error, while `btrfs_init_new_device` expects that the value will be `NULL`. This caused a subsequent invalid pointer dereference.

To replay the crash, we installed a version of the `btrfs` module from just before the application of the patch. To cause `open_bdev_exclusive` to fail we first create and mount a `btrfs` volume and then attempt to add to this volume a new device that is not yet created. This operation is handled by the `btrfs_ioctl_add_dev` ioctl which calls `btrfs_init_new_device` with the device path as an argument. This path value is then passed to `open_bdev_exclusive` which fails to locate the device and returns an `ERR_PTR` value. Fig. 4 shows an extract of the resulting oops report. Line 1 shows that the crash is due to an attempt to access an invalid memory address. Line 5 shows that the faulty operation occurred in the function `btrfs_init_new_device` a priori during a call to `btrfs_ioctl_add_dev` (line 8). Source files and line numbers can be obtained by applying the standard debugger `gdb` to the compiled module and to the compiled kernel.

This backtrace contains possibly stale pointers, as indicated by the `?` symbol on lines 8 and 9. While `btrfs_ioctl_add_dev` really does call `btrfs_init_new_device`, this is not the case of `memdup_user`. Since it cannot be known a priori whether a function annotated with `?` is really stale, the service developer has to find and study the definitions of all of the functions at the top of the backtrace, until finding the reason for the crash, including the definitions of functions that may be completely unrelated to the problem. A goal of the kernel debugger kdb,[1] which was merged into the mainline in Linux 2.6.35, was to improve the quality of backtraces. Nevertheless, backtrace quality remains an issue.[2]

```
1  [847.353202] BUG: unable to handle kernel paging request at
2               ffffffee
3  [847.353205] IP: [<fbc722d9>] btrfs_init_new_device+0xcf/0x5c5
4               [btrfs]
5  [847.353229] *pdpt = 00000000007ee001 *pde = 00000000007ff067
6  [847.353233] Oops: 0000 [#1] ...
7  [847.353291] EIP is at btrfs_init_new_device+0xcf/0x5c5 [btrfs] ...
8  [847.353298] Process btrfs-vol (pid: 3699, ...
9  [847.353312] Call Trace:
10 [847.353327] [<fbc7b84e>] ? btrfs_ioctl_add_dev+0x33/0x74 [btrfs]
11 [847.353334] [<c01c52a8>] ? memdup_user+0x38/0x70 ...
12 [847.353451] ---[ end trace 69edaf4b4d3762ce ]---
```

Figure 4: Oops report following a `btrfs` `ERR_PTR` pointer dereference crash.

Relevance of kernel oops reports. A kernel oops backtrace contains only the instruction causing the crash and the sequence of function calls considered to be on the stack. The actual reason for a crash, however, may occur in previously executed code that is not represented. For the fault shown in Fig. 2, the oops report mentions a dereference of the variable *bdev* in the function `btrfs_init_new_device`, but the real source of the problem is at the initialization of *bdev*, to the result of calling `open_bdev_exclusive`. This call has returned and thus no longer appears on the stack.

[1] https://kgdb.wiki.kernel.org/

[2] https://lkml.org/lkml/2012/2/10/129

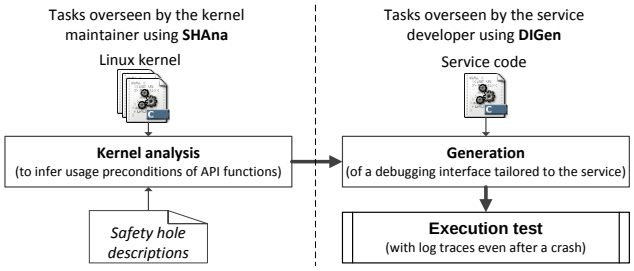

Figure 5: The steps in using Diagnosys

Such situations make debugging more difficult as the developer must thoroughly consult kernel and service source code to localize important initialization code sites.

Kernel hangs. By default, the Linux kernel gives no feedback in the case of a kernel hang. It can, however, be configured to panic when it detects no progress over a certain period of time. When the hang is due to an infinite loop, the backtrace resulting from the panic can occur anywhere within this loop; the point of the panic may thus have no relation to the actual source of the problem.

4. DIAGNOSYS

The goal of Diagnosys is to improve the quality of the information available when a crash or hang occurs and this crash or hang results from a safety hole in a kernel exported function. The use of Diagnosys involves three phases (Fig. 5): 1) identification of safety holes in kernel exported functions and inference of the associated preconditions, using the static analysis tool SHAna, 2) automatic generation of a debugging interface using DIGen based on the inferred preconditions, and 3) testing service code with the support of the debugging interface. The first phase is carried out only once by a kernel maintainer, for each new version of the mainline Linux kernel,[3] and the remaining phases are carried out by each service developer who would like to use Diagnosys.

4.1 Identifying safety holes and preconditions

SHAna first searches the kernel code for occurrences of the kinds of safety holes listed in Table 1, and then computes the preconditions that are required to hold to ensure that these safety holes do not cause a kernel crash or hang. The analysis focuses on unsafe operations that occur in code that is in or is reachable from an exported function. For each such occurrence, a backward analysis amounting to a simple version of Hoare logic [16] produces the weakest precondition to be satisfied on entry to the function, for entry safety holes, and on exit from the function, for exit safety holes, such that the safety hole will not cause a crash. SHAna is implemented using the Coccinelle program analysis tool [25], which analyzes all of the code, regardless of the code's targeted hardware platform, making it possible to find all safety holes in a single run. The result of SHAna is a list mapping each kernel exported function identified as containing safety holes to the associated preconditions.

[3]Each Linux distribution may add some specific patches to the Linux kernel. These are unlikely to affect the kernel API. Furthermore a service that should ultimately be integrated into the mainline kernel must be developed against the API supported by that kernel.

The analysis starts from the definition of an exported function, recognized as one declared using `EXPORT_SYMBOL` or `EXPORT_SYMBOL_GPL`. Table 1 indicates for each category of safety hole whether intraprocedural, interprocedural or no analysis is used. In search scenarios that only require intraprocedural analysis, the analyzer scans the definition of the exported function to identify code fragments that represent safety holes. For example, in searching for Intr exit safety holes, SHAna only looks for interrupt disabling operations in the kernel exported function itself, because interrupt state flags should not be passed from one function to another [29]. In the case of interprocedural analysis, SHAna starts from the definition of an exported function and iteratively analyzes all called functions.[4] For example, in searching for Null entry safety holes, SHAna searches through both the kernel exported function itself and all called functions that receive a parameter of the kernel exported function as an argument to find unchecked dereferences. SHAna furthermore includes unchecked dereferences of values that somehow depend on the value of an unchecked parameter.

4.2 Generating and integrating a debugging interface

Based on the results of SHAna, DIGen generates a debugging interface as a collection of wrapper functions that augment the definitions of kernel exported functions with the necessary checks and calls to logging primitives, to detect and record violations of safety hole preconditions. Ideally, the kernel maintainer who runs SHAna would also generate a single debugging interface for the entire kernel that could be used by all service developers. Unfortunately, many kernel source files target specific hardware platforms, and thus have mutually incompatible header file dependencies, making it impossible to compile a single debugging interface wrapping all of the kernel exported functions at once. Accordingly, we shift the interface generation process into the hands of the service developer, who generates an interface specific to his service. Because the functions invoked by a single service can necessarily be compiled together, this approach avoids all compilation difficulties, while producing a debugging interface that is sufficient for an individual service's needs. We now describe the generation of the debugging interface and how it is integrated into a service under development.

Generating a debugging interface. For each kernel exported function that is used in the service and for which SHAna identified at least one safety hole, DIGen generates a wrapper function. The general structure of such a wrapper is shown in Figure 6. Based on the argument values, the wrapper first checks each entry safety-hole precondition (line 4) and then, if the precondition is not satisfied, logs a message indicating the violation. This message includes the safety hole category, which specifies the kind of safety hole and whether the violation is *certain* or *possible* (line 5), as defined in Section 2.3. The wrapper then calls the original function. If the original function has a return value, this value is stored in a local variable, `__ret`, and then the preconditions on any exit safety holes are checked based on this information (lines 9-10). Finally, the return value, if any, of the original function is returned as the result (line 12).

[4]The analysis stops at function pointers.

```
1  static inline ⟨rtype⟩ __debug_⟨kernel function⟩ (...) {
2    ⟨rtype⟩ __ret;
3    /* Check preconditions for entry safety holes */
4    if ⟨an entry safety−hole precondition is violated⟩
5      diagnosys_log(⟨EF id⟩, ⟨SH cat⟩, ⟨info (e.g., arg number)⟩);
6    /* Invocation of the intended kernel function */
7    __ret = ⟨call to kernel function⟩;
8    /* Check preconditions for exit safety holes */
9    if ⟨an exit safety−hole precondition is violated⟩
10     diagnosys_log(⟨EF id⟩, ⟨SH cat⟩, ⟨info (e.g., err ret type)⟩);
11   /* Forward the return value */
12   return __ret;
13 }
14 #define ⟨kernel function⟩ __debug_⟨kernel function⟩
```

Figure 6: Wrapper structure for a non-void function

Integrating a debugging interface into a service. The generated debugging interface is implemented as a header file to be included in the service code. Once compiled with the interface included, the service uses the wrapper functions instead of the corresponding kernel exported functions.

Diagnosys provides an automated script, `dmake`, that manages the generation of a debugging interface. This script (1) compiles the original service code, (2) identifies the kernel exported functions referenced by the resulting object files, (3) generates an interface dedicated to these functions, and (4) recompiles the service with the interface included.

4.3 Running service code with Diagnosys

To be able to use a Diagnosys-generated debugging interface, the service developer must use a version of the Linux kernel providing support for the Diagnosys runtime system. This support is expressed as a kernel patch, which we have implemented for Linux 2.6.32, that extends the kernel with a crash resilient logging system. The patch additionally configures the kernel to send all crashes and hangs (Linux soft and hard lockups) to the kernel panic function, which the patch extends to reboot into a special *crash kernel* if Diagnosys is activated or to continue with a normal panic, otherwise. Finally, the Diagnosys runtime system includes a tool that can be run from user space to install a copy of the Diagnosys kernel as a crash kernel, initialize the reserved log buffer, and activate and deactivate logging.

Once the Diagnosys logging system has been activated, the service developer may test his code as usual. During service execution, if a wrapper function detects a safety hole for which the precondition is violated, the wrapper logs information about the safety hole in a reserved area of memory, annotated with a timestamp and including the memory address of the call site. The reserved area of memory is managed through a ring buffer that retains information about only the most recent violations.

On a kernel crash or hang, the Diagnosys runtime system uses a Kexec-based [24] mechanism to reboot into a new instance of the Diagnosys-enabled kernel. The Kexec-based mechanism performs the reboot without reinitializing any hardware, including the memory, thus ensuring that the accumulated Diagnosys log is still available. The service developer may then access the log messages through a pseudo character device. The messages are made available in the order in which they were generated. When a crash occurs, the Diagnosys runtime system also inserts the kernel stack trace into the Diagnosys log before rebooting.

Table 2: Diagnosys prototype code size

Diagnosys component	Tool		Code size (LOC)	Language
Kernel code analyzer	SHAna		2438 + 1331	SmPL[25] + OCaml
Wrapper generator	`dmake` + DIGen		49 + 1301	sh + OCaml
Logging system		user-space	115 + 355	sh + ansi C
		kernel-space	645	ansi C code patch

4.4 Implementation

Table 2 gives the code sizes of the various parts of our prototype Diagnosys implementation. The implementation includes the SHAna analysis of Linux kernel, DIGen and `dmake` for generating and compiling wrappers for a given service, and the patch for the runtime system.

5. EVALUATION

In designing Diagnosys, we have chosen to focus on the interface between the service code and the kernel. We first assess the number of safety holes in this interface and their past impact on kernel robustness, as evidenced by commits to the Linux kernel. We then assess the difficulty of debugging kernel faults derived from safety holes, by studying the feedback made available to the service developer on a crash or hang without Diagnosys, namely the kernel backtrace. Then, we assess the coverage of Diagnosys with respect to the possible crashes and hangs that are triggered by misuse of the interface between the service code and the kernel, and show that the Diagnosys log messages allow the service developer to find the cause of a crash or hang more rapidly than when relying on a kernel backtrace alone. Finally, we show that Diagnosys incurs a sufficiently low runtime overhead to be embedded in a service, up to the early deployment phase.

Our experiments use code from Linux 2.6.32, which was released in December 2009. This version is used in the current Long Term Support version of Ubuntu® (10.04), in Red Hat Enterprise Linux 6, in Oracle Linux, etc. Our performance experiments are carried out on a Dell 2.40 GHz Intel® Core™ 2 Duo with 3.9 GB of RAM. Unless otherwise indicated, the OS is running a Linux 2.6.32 kernel that has been modified to support the Diagnosys logging infrastructure. 1MB is reserved for the crash-resilient log buffer.

5.1 Prevalence and impact of safety holes

Diagnosys is only beneficial if SHAna identifies safety holes in functions that are used by a wide range of drivers and if these functions are likely to be used in an incorrect way. In this section, we assess the number of safety holes collected by SHAna and then study the impact these safety holes have had on the robustness of the Linux kernel itself.

In Linux 2.6.32, SHAna reports 22,940 safety holes in 7,505 exported functions. Table 3 summarizes for each kind of safety hole the number of functions that SHAna identifies as containing at least one occurrence of that kind of safety hole. In the largest category, INull/Null, about 94% of the reported functions perform unsafe dereferences directly, and 5% forward the parameter value to other functions that unsafely use them with no prior check. Around 1% perform unsafe dereferences on variables whose validity are indirectly correlated to that of pointer parameters. Defects due to safety holes in the latter two categories are more difficult for the service developer to identify.

Static analysis is necessarily approximate, as it does not have complete access to run-time values. This may lead to

Table 3: Prevalence of safety holes in Linux 2.6.32

Safety hole	Number of exported functions collected in the	
	entry sub-category	exit sub-category
Block	367	815
INull/Null	7,220	1,124
Var	5	11
Lock/Intr/LockIntr	815	23
Free	-	11
Size	8	-
Range	-	8

false positives, in which a safety hole is reported that in fact cannot lead to a crash. Such false positives can increase the logging time and clutter the log with irrelevant messages. Nevertheless, having studied the complete set of results for Linux 2.6.32, we have found only 414 false positives, out of the 22,940 reported safety holes. 405 of these are due to the presence of multiple, configuration-specific, definitions of some functions. SHAna annotates safety holes derived from calls to such functions with the file in which the relevant function instance is defined, so that the service developer can remove those that are not relevant to his configuration.

Of the 147,403 call sites across the entire kernel source code where exported functions are used, half invoke a function containing an identified safety hole. This suggests that the kernel exported functions containing safety holes are likely to be useful to new services.

To assess the past impact of the identified safety holes over the course of the development of Linux, we have searched through the 278,078 commits to Linux 2.6,[5] from 2.6.12 to 2.6.39.3, to identify those whose changelog mentions the name of at least one kernel function exported in Linux 2.6.32, ignoring changelogs in which the function name is used as a common word (*e.g.*, "sort", "panic", etc.). Linux commits by convention make only a single logical change, thus making the analysis precise.[6] 703 of these commits contain bugs described in kernel changelogs[7] that are related to the usage of exported functions. 267 of them, *i.e.*, 38% are related to the categories of safety holes that we consider in this paper.

5.2 Kernel debugging with Diagnosys

As discussed in Section 3, kernel debugging is made difficult by unreliable backtraces and by the questionable relevance of the information in crash reports. To assess the qualitative benefits of Diagnosys, we have replayed a crash and a hang reported in kernel commit logs.

Replaying a kernel crash. As an example of kernel crash, we again consider the *btrfs* example of Fig. 2. Study of the corresponding crash report in Fig. 4 showed that the source of the problem was not readily available in the backtrace. We have therefore replayed the same execution scenario when using Diagnosys. A typical Diagnosys log line contains the timestamp of the log, the source file and line number where the unsafe call was performed, the name of the exported function, the category of the safety hole and possibly the name of a relevant argument or an unsafe return value. In the case of the replay of the *btrfs* crash, Fig. 7 shows the last line added to the Diagnosys log before the crash, which is the line that the developer is likely to consult first. This line shows that the function `open_bdev_exclusive` activated an Inull exit safety hole by returning an ERR_PTR. It also reports the runtime timestamp and the call site where the safety hole was violated. Combining this information with the information about the crash site in the oops report and the service source code shows that the problem is the inadequate error handling code after `open_bdev_exclusive`.

```
1  [4294934950]|@/var/diagnosys/tests/my_btrfs/volumes.c:1441|
2              open_bdev_exclusive|INULL(EXITED)|ERR PTR|
```

Figure 7: Diagnosys log line in the execution of `btrfs`

Replaying a kernel hang. Kernel hangs are notoriously hard to debug[8] as the panic, which occurs long after the actual fault, can produce a backtrace that is hard to correlate to the source of the problem. Diagnosys records information about previous potentially dangerous operations.

Just before the release of Linux 2.6.33, the `nouveau_drm` nVidia® graphics card driver contained a hang resulting from the use of the kernel exported function `ttm_bo_wait`. This function exhibits a Lock entry safety hole and a Lock exit safety hole, as it first unlocks and then relocks a lock received via its first argument. The `nouveau_drm` driver called this function without holding this lock, hanging the kernel.

In Fig. 8a, the last line of the Diagnosys log shows that `ttm_bo_wait` has been called without the expected lock held. Correlating this information with the source code suggests taking the lock before the call and releasing it after the call, as shown in the Linux patch in Fig. 8b.

```
1  [437126]|@/var/diagnosys/tests/nouveau/nouveau_gem.c:929|
2          ttm_bo_wait|LOCK/ACQUIRE(POSSIBLE)|bo->lock|
```

a) Diagnosys log line in the execution of `nouveau_drm`.

```
1  commit f0fbe3eb5f65fe5948219f4ceac68f8a665b1fc6
2  +  spin_lock(&nvbo->bo.lock);
3     ret = ttm_bo_wait(&nvbo->bo, false, false, no_wait);
4  +  spin_unlock(&nvbo->bo.lock);
```

b) Bug fix related to the usage of `ttm_bo_wait`.

Figure 8: Fault involving a Lock safety hole in `nouveau_drm`

5.3 Quantifying the debugging benefit

To be useful, Diagnosys must cover a high percentage of the misuses of kernel exported functions. We first evaluate this by artificially creating and activating misuses of exported functions in kernel services and measuring how many are trapped by Diagnosys. Additionally, Diagnosys must be able to produce log messages that ease the debugging process. We evaluate the debugging effort by measuring the number of files and functions that have to be studied to identify the cause of a crash, with and without Diagnosys. Our experiments involve a number of commonly used kinds of services: networking code, USB drivers, multimedia drivers, and file systems. Services of these kinds make up over a third of the Linux 2.6.32 source code. We have selected a range of services that run on our test hardware (Table 4).

[5] git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git.

[6] Documentation/SubmittingPatches in the Linux kernel source tree, http://www.kernel.org

[7] http://www.kernel.org/pub/linux/kernel/v2.6/

[8] http://www.linuxjournal.com/article/5749

Table 4: Tested Linux 2.6.32 services

Category	Service module	Description	Used functions with safety holes
Networking	e1000e	Ethernet adapter	57
	iwlagn	Intel WiFi Next Gen AGN	57
	btusb	Bluetooth generic driver	26
USB drivers	usb-storage	Mass storage device driver	51
	ftdi_sio	USB to serial converter	31
Multimedia device drivers	uvcvideo	Webcam device driver	28
	snd-intel8x0	ALSA driver	35
File systems	isofs	ISO 9660 file system	26
	nfs	Network file system	198
	fuse	File system in userspace	86

Coverage of Diagnosys. To determine the coverage of Diagnosys, we first mutate existing services so as to artificially create bugs. Then, we inject faults at run-time to potentially cause the mutation to trigger a crash.

Since the largest percentage of our identified safety holes are related to `NULL` and `ERR_PTR` dereferences, we focus on these safety holes. One prominent source of such values is as the result of a call to a function that has failed in performing some sort of allocation. Robust kernel code checks for these values and aborts the ongoing computation. Nevertheless, omission of these tests is common. For example, in Linux 2.6.32, even for the standard kernel memory allocation functions `kmalloc`, `kzalloc`, and `kcalloc`, over 8% of the calls that may fail do not test the result before dereferencing the returned value or passing it to another function.

Based on these observations, our experiments focus on missing `NULL` and `ERR_PTR` tests in the service code. Our mutations remove such tests from the service code, one by one, and use the `failslab` feature of the Linux fault injection infrastructure [6] within the initialization of the tested value to inject failures into the execution of any call to a basic memory allocation function that this initialization involves. Because the initialization can invoke basic memory allocation functions multiple time, a single mutation experiment may involve multiple injected faults.

One possible result is that there is no observable effect. This occurs when the called function does not involve a memory allocation, when the failure of memory allocations does not lead to a `NULL` or `ERR_PTR` result, or when the safety hole is *possible* and is not encountered in the actual execution. Another possibility is that there is a crash, but there is no relevant information in the Diagnosys log. In this case, either the log information has been overwritten, SHAna has not detected the safety hole, or the call to a kernel exported function occurs in a header file that, for technical reasons, has to be included before the Diagnosys wrapper definitions. A third possibility is that there is a crash and the information is logged, representing a success for Diagnosys.

We have evaluated the coverage of Diagnosys on the 10 services listed in Table 4. Removing the `NULL` and `ERR_PTR` tests one by one leads to 555 mutated services. For each mutated service, we have exercised the various execution paths of the affected module in order to execute the mutated code. The results are shown in Table 5. 56% of the mutations resulted in a kernel crash. After reboot, in 90% of the crashes, the log contained information relevant to the crash origin and in 86% of the crashes, a log was present and it was additionally in the last position. For one service, the latter only holds for 66% of the crashes, but this amounts to only one missing log, as this service has few mutation sites.

Table 5: Results of the mutation experiments

Category	Kernel module	# of mutations	# of crashes with			Coverage
			no log	log is not last	log is last	
Networking	e1000e	57	0	0	20	100%
	iwlagn	18	1	0	8	88.9%
	btusb	9	1	0	7	87.5%
USB drivers	usb-storage	11	0	0	3	100%
	ftdi_sio	9	0	0	6	100%
Multimedia device drivers	snd-intel8x0	3	1	0	2	66.7%
	uvcvideo	34	3	3	17	73.9%
File systems	isofs	28	3	0	9	75.0%
	nfs	309	13	9	157	87.7%
	fuse	77	3	1	41	91.1%

Ease of the debugging process. Provided with an oops report containing a backtrace and debugging tools that can translate stack entries into file names and line numbers, a developer typically starts from the point of the crash, visiting all files and caller functions until the origin of the crash is localized. When the crash occurs deep in the execution, the number of functions and files to visit can become large.

We have considered 199 of the mutations performed in our coverage tests that lead to crashes, from btusb, nfs, and isofs. We also consider 31 mutations in nfs code that add statements for arbitrarily acquiring and releasing locks in services in order to provoke kernel hangs, focusing on locks that are passed between functions as they can trigger safety holes in core kernel code.

We have compared the 230 oops reports with the corresponding Diagnosys logs. In 92% of these crashes, the Diagnosys log contains information on the origin of the fault. For those cases, debugging with the oops report alone required consulting 1 to 14 functions, including on average one possibly stale pointer, in up to 4 different files distributed across kernel and service code. In 73% of the cases for which the Diagnosys log contains relevant information, we find that using Diagnosys reduces by at least 50% the number of files and functions to consult. In 19% of the cases for which the Diagnosys log contains relevant information, the crash occurred in the same file as the mutation, but the Diagnosys log made it possible to more readily pinpoint the fault by providing line numbers that are closer to the mutation site.

5.4 Overhead

Introducing wrappers on kernel-exported functions incurs a performance overhead on service execution. To assess the impact of this overhead, we execute various real-world kernel services with and without a debugging interface.

Network driver performance. Our first test involves a Gigabit Ethernet device that requires both low latency and high throughput to guarantee high performance. We evaluate the impact of a debugging interface by exercising the e1000e Linux device driver using the TCP_STREAM, UDP_STREAM and UDP_RR tests from the netperf benchmark.[9] For these experiments, the netperf utility was configured to report results accurate to 5% with 99% confidence. Table 6 summarizes the performance for the e1000e driver when run without and with a debugging interface. The debugging interface only reduces the throughput by 0.4% to 6.4%.

File system performance. Our second test involves the NFS file system, whose implementation uses around 200

[9] http://www.netperf.org

Table 6: Performance of the e1000e driver

Test		Without Diagnosys	With Diagnosys	Overhead
TCP_STREAM	Throughput	907.91 Mb/s	904.32 Mb/s	0.39%
UDP_STREAM	Throughput	951.00 Mb/s	947.73 Mb/s	0.34%
UDP_RR	Throughput	7371.69 Tx/s	6902.81 Tx/s	6.36%

Table 7: Performance of the NFS file system

Record block size(Kb)	Without Diagnosys (Access rate: K/sec) read/write	With Diagnosys (Access rate: K/sec) read/write	Overhead read/write
128	45309/31672	42141/28072	6.99%/11.36%
256	49780/36577	48196/32900	3.18%/10.05%
512	49764/39957	45765/37981	8.03%/4.94%

exported functions exhibiting safety holes. The experiment consists of sequential block read and write phases accessing records of a given size based on patterns generated by the IOzone file system benchmark.[10] 8G of data are accessed. For this experiment, the client and server run on the same machine, connected using a loopback interface. Read and write operations are performed in the direct I/O mode with varying record sizes. With a debugging interface integrated into the `nfs` file system, we have recorded around 16 million calls to the interface wrapper functions when using a record size of 512 Kb. As shown in Table 7, the overhead varies between 3% and 11%, depending on the record size.

6. RELATED WORK

In the last decade, studies have shown that kernel-level services, in particular device drivers, are responsible for the majority of OS crashes. Ganapathi *et al.* have found that 65% of all Windows XP crashes are due to device drivers [14]. Ten years ago, Chou *et al.* found that the fault rate in Linux drivers is 3–7 times higher than that of other parts of the kernel [5]. Palix *et al.* have shown that while this error rate is decreasing, Linux drivers still contain many defects [26]. They have also found that file systems have recently had a high fault rate, indeed even higher than that of drivers.

System robustness testing. Fault injection has been applied to the Linux kernel to evaluate the impact of various fault classes [1, 7]. Our work identifies the safety holes in kernel interfaces that explain their observations. Marinescu and Candea [21] focus on the returns of error codes from userspace library functions. These are analogous to our Null exit safety holes. Their approach, however, is not applicable to other types of safety holes.

Static bug finding. Model checking, theorem proving, and program analysis have been used to analyze OS code to find thousands of bugs [3, 9, 18, 27]. Nevertheless, these tools take time to run and the results require time and expertise to interpret. Thus, these tools are not well suited to the frequent modifications and tests that are typical of initial code development. Numerous approaches have proposed to statically infer so-called *protocols*, describing expected sequences of function calls [9, 18, 19, 20, 28]. These approaches have focused on sequences of function calls that are expected to appear within a single function, rather than the specific interaction between a service and the rest of the kernel.

Some of our kinds of safety holes could be eliminated by the use of advanced type systems. For example, Bugrara and Aiken propose an analysis to differentiate between safe and unsafe userspace pointers in kernel code [4]. They focus, however, on the entire kernel, and thus may inform the service developer about faults in code other than his own.

Logging. Runtime logs are frequently insufficient for failure diagnosis especially in case of unexpected crashes [8]. *LogEnhancer* [32] enriches log messages with extra information, but does not create new messages. Diagnosys creates new log messages along the kernel-service boundary, where they can be most helpful to service developers.

Robust interfaces. LXFI [31] isolates kernel modules and includes the concept of *API integrity*, which allows developers to define the usage contract of kernel interfaces by annotating the source code. LXFI, however, aims at limiting the security threat posed by the privileges granted to kernel modules, while Diagnosys focuses on various categories of common faults encountered in kernel code.

Healers automatically generates a robust interface to a user-level library without access to the source code [11]. It relies on fault injection to identify the set of assumptions that a library function makes about its arguments. Healers can obtain information about runtime values, such as array bounds, that may be difficult to detect using static analysis. However, Healers does not address safety hole kinds such as Lock that require calling-context information. Supporting Lock would require testing the state of all available locks, which would be expensive and are likely unknown.

Programming with contracts. A software *contract* represents the agreement between the developer of a component and its user on the component's functional behavior [13, 15, 22, 23]. Contracts include pre- and post-conditions, as well as invariants. A safety hole is essentially the dual of a contract, in that a contract describes properties that the context should have, while a safety hole describes properties that it should not have.

Contract inference is analogous to the execution of SHAna. Arnout and Meyer infer contracts based on exceptions found in .NET code [2]. Daikon infers invariants dynamically by running the program with multiple inputs and generalizing the observations [10]. Diagnosys targets situations that lead to unhandled exceptions, either in the kernel or the service code. Linux kernel execution is highly dependent on the particular architecture and devices involved, and thus a service developer would have to actively use Daikon in his own environment. SHAna allows the collection of safety holes to be centralized. Finally, only one of the invariants in the Daikon invariant list,[11] NonZero, may correspond to one of our safety hole kinds, namely INull. Daikon does not handle common safety hole kinds such as Free, or kernel-specific safety hole kinds such as Param, for user/pointer bugs.

The Extended Static Checker for Java (ESC/Java) [13] relies on programmer annotations to check method contracts. Annotation assistants such as Houdini [12] automate the inference of annotations. Houdini supports various exceptions involving arguments, such as NullPointerException and IndexOutOfBoundsException, but does not provide tests for the validity of allocated memory.

[10] http://www.iozone.org/

[11] http://groups.csail.mit.edu/pag/daikon, Documentation, Sec. 5.5

7. CONCLUSION

Defects in kernel-level services can cause the demise of the entire system, often leaving developers without any clue as to what went wrong. Debugging such problems is particularly challenging early in service development, when the code changes frequently and the developer is not necessarily aware of the usage preconditions of kernel interfaces. We have designed Diagnosys, a tool that detects safety holes in Linux kernel exported functions and supports the generation of a debugging interface, tailored for a particular service, according to this information. At runtime Diagnosys provides a crash-resilient logging system for recording information about risky uses of kernel functions containing safety holes.

Using fault injection tests on 10 Linux kernel-level services, we have shown that our interface alerts the developer to the critical defects in his code. Using a driver for a Gigabit Ethernet device and a NFS file system, we have shown that the performance impact of our approach is within the limits of what is acceptable when testing a kernel-level service in the initial stages of development, and can even be used up to the phase of initial deployment.

This work was supported in part by the ANR grant ABL. Source code and results from this work can be found at `http://momentum.labri.fr/projects/diagnosys`.

8. REFERENCES

[1] A. Albinet, J. Arlat, and J.-C. Fabre. Characterization of the impact of faulty drivers on the robustness of the Linux kernel. In *DSN'04*, pages 867–876.

[2] K. Arnout and B. Meyer. Uncovering hidden contracts: The .NET example. *Computer*, 36:48–55, 2003.

[3] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In *EuroSys'06*, pages 73–85.

[4] S. Bugrara and A. Aiken. Verifying the safety of user pointer dereferences. In *IEEE Symposium on Security and Privacy*, pages 325–338, Oakland, CA, USA, 2008.

[5] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *SOSP'01*, pages 73–88, Banff, Canada.

[6] J. Corbet. Injecting faults into the kernel. `http://lwn.net/Articles/209257/`, November 2004.

[7] D. Cotroneo, R. Natella, and S. Russo. Assessment and improvement of hang detection in the Linux operating system. In *SRDS'09*, pages 288–294.

[8] Y. Ding, M. Haohui, X. Weiwei, T. Lin, Z. Yuanyuan, and P. Shankar. Sherlog: Error diagnosis by connecting clues from run-time logs. In *ASPLOS'10*, pages 143–154, Pittsburgh, PA, USA.

[9] D. 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'01*, pages 57–72, Banff, Alberta, Canada.

[10] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69:35–45, December 2007.

[11] C. Fetzer and Z. Xiao. Healers: a toolkit for enhancing the robustness and security of existing applications. In *DSN'03*, pages 317–322, San Francisco, CA, USA.

[12] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. In *FME'01*, pages 500–517, London, UK.

[13] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI'02*, pages 234–245.

[14] A. Ganapathi, V. Ganapathi, and D. Patterson. Windows XP kernel crash analysis. In *LISA'06*, pages 49–159, Washington, DC, USA.

[15] R. Hirschfeld, M. Perscheid, C. Schubert, and M. Appeltauer. Dynamic contract layers. In *SAC'10*, pages 2169–2175. ACM, 2010.

[16] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge University Press, 2000.

[17] V. Kuznetsov, V. Chipounov, and G. Candea. Testing closed-source binary device drivers with DDT. In *USENIX Annual Technical Conference*, June 2010.

[18] J. L. Lawall, J. Brunel, N. Palix, R. R. Hansen, H. Stuart, and G. Muller. WYSIWIB: A declarative approach to finding API protocols and bugs in Linux code. In *DSN'09*, pages 43–52, Lisbon, Portugal.

[19] C. Le Goues and W. Weimer. Specification mining with few false positives. In *TACAS'09*, volume 5505 of *LNCS*, pages 292–306, York, UK.

[20] Z. Li and Y. Zhou. PR-miner: automatically extracting implicit programming rules and detecting violations in large software code. In *ESEC/FSE-13*, pages 306–315, Lisbon, Portugal, 2005.

[21] P. Marinescu and G. Candea. Efficient testing of recovery code using fault injection. *ACM Transactions on Computer Systems (TOCS)*, 29(3), Nov. 2011.

[22] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., 1st edition, 1988.

[23] C. Mills. *Using Design by Contract in C*. OnLamp.com, O'Reilly, 1st edition, October 2004.

[24] H. Nellitheertha. Reboot Linux faster using kexec. `http://www.ibm.com/developerworks/linux/library/l-kexec/index.html`, 2004.

[25] Y. Padioleau, J. L. Lawall, R. R. Hansen, and G. Muller. Documenting and automating collateral evolutions in Linux device drivers. In *EuroSys'08*, pages 247–260, Glasgow, Scotland.

[26] N. Palix, S. Saha, G. Thomas, C. Calvès, J. L. Lawall, and G. Muller. Faults in Linux: Ten years later. In *ASPLOS'11*, pages 305–318.

[27] H. Post and W. Küchlin. Integrated static analysis for Linux device driver verification. In *IFM'07*, pages 518–537, Oxford, UK.

[28] M. K. Ramanathan, A. Grama, and S. Jagannathan. Path-sensitive inference of function precedence protocols. In *ICSE'07*, pages 240–250.

[29] A. Rubini and J. Corbet. *Linux Device Drivers*, page 109. O'Reilly Media, second edition, 2001.

[30] L. Ryzhyk, P. Chubb, I. Kuz, and G. Heiser. Dingo: Taming device drivers. In *EuroSys'09*, pages 275–288.

[31] M. Yandong, C. Haogang, Z. Dong, W. Xi, Z. Nickolai, and K. M. Frans. Software fault isolation with API integrity and multi-principal modules. In *SOSP'11*.

[32] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage. Improving software diagnosability via log enhancement. In *ASPLOS'11*, pages 3–14.