Ahead of Time Static Analysis for Automatic Generation of Debugging Interfaces to the Linux Kernel

Tegawendé F. Bissyandé, Laurent Réveillère, Julia Lawall, Gilles Muller

Journal of Automated Software Engineering manuscript No.
(will be inserted by the editor)

Ahead of Time Static Analysis for Automatic Generation of Debugging Interfaces to the Linux Kernel

Tegawendé F. Bissyandé · Laurent Réveillère · Julia L. Lawall · Gilles Muller

Received: 2013 / Accepted: 21 April 2014

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.

Keywords Diagnosys · Debugging · Wrappers · Linux · Device Drivers, Plugin · Reliability · Testing.

1 Introduction

Debugging is difficult. And debugging an operating system kernel-level service, such as a device driver, file system, or network protocol, is particularly difficult. When a service crashes, 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. And 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

This is an extended version of a paper presented at the 2012 International Conference on Automated Software Engineering, Essen - Germany [7]

T.F. Bissyandé
SnT, University of Luxembourg
4, rue Alphonse Weicker - L-2721 Luxembourg
E-mail: tegawende.bissyande@uni.lu

L. Réveillère
LaBRI, University of Bordeaux - 351, Cours de la Libération
33400 Talence
E-mail: laurent.reveillere@labri.fr

J.L. Lawall & G. Muller
Inria/LIP6/UPMC/Sorbonne University Regal - 4, Place Jussieu
75252 Paris
E-mail: Julia.Lawall, Gilles.Muller@lip6.fr

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 a 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 challenging. Indeed, the Linux kernel is designed around 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, and whose code is not yet integrated into the mainline kernel. Advances in bug-finding tools [3,32,33], specialized testing techniques [31,35], and code generation from specifications [47] 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 API function that returns `NULL` as the result represents a safety hole, because a calling service could dereference this result without checking its value. We stress, however, that a safety hole is not a fault. It constitutes an implementation choice for a function that poses a risk to the safety of other functions that call that function. Unfortunately, such a risk is often undocumented.

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. The generation of the interface is automatic, based on a prior static analysis of the Linux kernel source code, that is performed only once, for each release. The generated 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 generated automatically, 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 API functions. The service developer does not interact with SHAna. DIGen is run by a service developer as

part of the service compilation process, using a Diagnosys-specific variant of the standard *make* command. During the execution of the resulting service, the debugging interface uses the runtime system to log information about any unsafe uses of functions containing safety holes in a buffer that is preserved across reboots. On a kernel crash or hang, the service developer can consult the buffer after reboot 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, after carefully analysing all 703 commits of Linux 2.6 that explicitly refer, in their log message, to a function exported in Linux 2.6.32, we have found that 38% of these commits 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. We also describe the runtime logging system and discuss how, in practice, Diagnosys supports debugging tasks. Section 5 evaluates our approach. Finally, Section 6 discusses related work, and Section 7 concludes.

2 Characterization of Safety Holes

In kernel programming, we consider a safety hole to be *a fragment of kernel code that can cause the kernel to crash or hang if the code is executed in a context that does not respect some implicit preconditions*. 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.

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 API functions.*

Figure 1a shows an excerpt of the definition of the kernel API 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`. Since kernel code runs in a single protection domain, dereferencing an invalid pointer will crash the entire system. 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 Figure 1b, is to avoid calling `skb_put` in this case. `skb_put` remains unchanged.

```
1  unsigned char *skb_put(struct sk_buff *skb, ...)
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
3      tx_skb = dev_alloc_skb(pkt_len);
4  +   if (!tx_skb) { ... goto out; }
5      pkt_data = skb_put(tx_skb, pkt_len);
```

b) Excerpt of the bug fix patch

Fig. 1: Bug fix of the usage of `skb_put`

Figure 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. Indeed, many allocation, initialization and copy functions may fail and return such a value, and leaving these values unchecked is a common fault. 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`. Figure 2b shows the patch that was used to fix 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
3      bdev = open_bdev_exclusive(...);
4  -   if (!bdev) return -EIO;
5  +   if (IS_ERR(bdev)) return PTR_ERR(bdev);
```

b) Excerpt of the bug fix patch

Fig. 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 Figure 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 Figure 3c.

Safety holes can also be found in the implementations of critical sections with locks. For example, between Linux 2.6.32 and 2.6.33, a bug was first introduced and then fixed in the `nouveau drm` driver for nVidia® cards, in which the `ttm_bo_wait` exported function was called by `nouveau_gem_ioctl_cpu_prep` without holding the *bo* lock. Since `ttm_bo_wait` attempts to release and then re-acquire that lock, as depicted in Figure 4(a), a subsequent call to another function, `nouveau_bo_busy`, which acquires the lock, hanged the machine. This bug was fixed by ensuring that the *bo* lock is held when invoking the `ttm_bo_wait` API function (cf. Figure 4(b)).

```
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
3      page = read_mapping_page(...);
4  +   if (IS_ERR(page)) { ... goto out; }
5      pptr = kmap (page);
```

c) Excerpt of the bug fix patch

Fig. 3: Bug fix of a use of `kmap`

```
1  // drivers/gpu/drm/ttm/ttm bo.c
2  int ttm_bo_wait(struct ttm_buffer_object *bo, ...){ ...
3      spin_unlock(&bo->lock);
4      driver->sync_obj_unref(&tmp_obj);
5      spin_lock(&bo->lock); ...
6  }
```

a) Excerpt of the definition of `ttm_bo_wait`

```
1  commit f0fbe3eb5f65fe5948219f4ceac68f8a665b1fc6
2  if (req->flags & NOUVEAU_GEM_CPU_PREP_NOBLOCK) {
3  +   spin_lock(&nvbo->bo.lock);
4      ret = ttm_bo_wait(&nvbo->bo, false, false, no_wait);
5  +   spin_unlock(&nvbo->bo.lock);
6  }
```

b) Excerpt of the bug fix patch

Fig. 4: Bug fix of critical section code

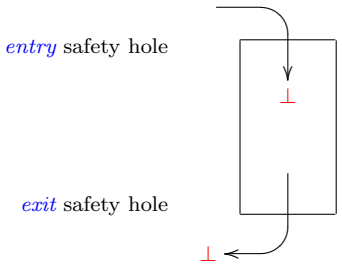

Fig. 5: Schematisation of the two families of safety holes

2.2 Taxonomy of safety holes

The examples described above show (1) how safety holes lead to bugs that reach the mainline kernel, and (2) how safety holes involve various types of bugs. We now describe a methodology, based on our observations in these examples, for identifying types of safety holes systematically, by recasting a type of code fault as a collection of one or more types of safety holes. We then use this methodology to enumerate the types of safety holes considered in this paper.

2.2.1 Methodology

As illustrated by the bug fixes in Section 2.1, some fragments of code executed by kernel API functions, while themselves being correct, can provoke kernel crashes or hangs when the API function is used incorrectly. In the most obvious case, the service code passes to the API function some bad arguments or calls it from an inappropriate context (e.g., with a lock held or with interrupts disabled) causing an execution error in the core kernel: we refer to this kind of safety hole as an *entry* safety hole. In other cases, the kernel API function may return a value or return from a context that might not meet the expectations of the service code: we refer to this kind of safety hole as an *exit* safety hole. Such *exit* safety holes appear when, for example, the API function may return *NULL* rather than a pointer to a valid region of memory, or returns with a lock held or with interrupts turned off. In these cases the crash or hang may happen in the service code, with the API function that caused the problem no longer on the call stack. Figure 5 schematically represents the two families of *entry* and *exit* safety holes, by indicating where, in each case, the execution error occurs. In the figure, the box represents the definition of an API function, while the symbol ⊥ indicates the point of crash.

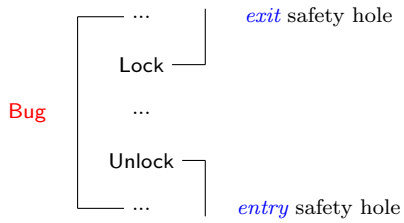

Fig. 6: Derivation of safety hole kinds from the elements of a Lock bug type

In general, we observe that any bug type that involves multiple disjoint core fragments can lead to an entry or exit safety hole. An an example, consider Figure 6, based on the example of a bug involving a lock/unlock combination for defining a critical section. The middle of the figure represents correct code, with Lock followed by a corresponding Unlock. The left side of the figure represents the potential bug case. Within a known block of code, represented by the pair of right angles, a bug may occur if both elements of code are expected to be present, but one is missing, i.e., a Lock missing a subsequent Unlock, or an Unlock with no previous lock. On the right side of the figure, we consider how the relevant fragments of code can be intended to be split across an API function, represented by either of the shown right angles, and the service code that uses the API. The API function may end with a Lock, expecting the calling service to perform the corresponding Unlock, representing an exit safety hole, or it may begin with an Unlock, conversely representing an entry safety hole. Thus, a single possible bug type gives rise to as many kinds of entry and exit safety holes as there are ways to split the elements of code relevant to the bug type across the boundary between the API function and the service code.

2.2.2 Taxonomy

Safety hole kinds. Following our methodology, we can, in principle, derive entry and exit safety hole types from any fault type involving multiple fragments of code. As a first source of fault types, we consider those identified by Chou *et al.* in their 2001 empirical study of Linux code [12]. We also refer to a more recent study by Palix *et al.* who revisited the study of Chou *et al.* in 2011 [41].

Table 1 details the different kinds of safety holes that were derived from these fault types, following the terminology of Palix *et al.* [41]. For each fault type, we split the possible fault code fragment into two parts and characterize the part that may be located inside the implementation of an API function f, thus identifying the possible entry and exit safety holes. We have enhanced some of the categories to take into account new kernel features. For example, in Linux kernel 2.6, `ERR_PTR`, in addition to `NULL`, is widely used as an invalid pointer indicating errors. Thus, in the Null and INull categories, we have added `ERR_PTR` as an invalid pointer.

For some categories of faults, however, we have not been able to derive interesting safety holes:

- **NullRef** (*A pointer is dereferenced and then tested for being* `NULL`.) seems meaningless as a safety hole. If the dereference fails, then the machine has crashed. If the dereference succeeds, this value will not cause the machine to crash.
- **Float** (*Do not use floating point in the kernel.*) is a purely local property. Thus, it is not relevant to the interface between a service code and the kernel API functions.
- **Real** (*Do not leak memory by updating pointers with potentially* `NULL` *realloc return values.*), in practice, only manifests itself as a direct assignment of the pre-existing pointer to the result of realloc, and thus it is also a purely local property.
- **Size** (*Allocate enough memory to hold the type for which you are allocating*), in practice, is local to the definition of a function, as sizes are typically expressed in terms of `sizeof`, which is evaluated according to type information as defined in the current context. Thus, it is not

Block	*To avoid deadlock, do not call blocking functions with interrupts disabled or a spinlock held.*	
	Entry safety hole	f possibly/certainly calls a blocking function
	Exit safety hole	f returns after disabling interrupts or while holding a lock
Null/INull	*Check potentially* `NULL` *pointers returned from routines.* *Do not make inconsistent assumptions about whether a pointer is* `NULL`/`ERR_PTR`.	
	Entry safety hole	f possibly/certainly dereferences an argument without checking its validity
	Exit safety hole	f returns `NULL`/`ERR_PTR` pointer
Var	*Do not allocate large stack variables (> 1K) on the fixed-size kernel stack.*	
	Entry safety hole	f possibly/certainly allocates an array whose size depends on a parameter
	Exit safety hole	f returns a large value
Range	*Always check bounds of array indices and loop bounds derived from user data.*	
	Entry safety hole	f possibly/certainly uses an unchecked parameter to compute an array index
	Exit safety hole	f returns value obtained from user level
Lock	*Release acquired locks; Do not double-acquire locks*	
LockIntr	Entry safety hole	f possibly/certainly acquires a lock derived from a parameter
	Exit safety hole	f returns without releasing an acquired lock
Intr	*Restore disabled interrupts.*	
LockIntr	Entry safety hole	f possibly/certainly calls a blocking function
	Exit safety hole	f returns with interrupts disabled
Free	*Do not use freed memory.*	
	Entry safety hole	f possibly/certainly dereferences a pointer-typed parameter value
	Exit safety hole	f frees memory derived from a parameter
Param	*Do not dereference user pointers.*	
	Entry safety hole	f possibly/certainly dereferences a pointer-typed parameter
	Exit safety hole	f returns a pointer-typed value obtained from user level

Table 1: Recasting common faults in Linux into safety hole categories. f refers to an API function

interprocedural between service code, where the call might be, and kernel code, where the fault might be activated.

Possible vs. certain safety holes. As a safety hole involves a fragment of code inside the definition of an API function, it belongs to one or more potential execution paths through that function. Such an execution path may be *possibly* followed during a call to the API function, if the code involved in the safety hole is only executed on certain inputs or calling contexts, or it may be *certainly* followed, implying that the code involved in the safety hole is always reached when the function is called.

Figure 7 illustrates a combination of execution paths from an entry point (noted as ●) of an API function to any of the different potential exit points (noted as ■). The execution paths pass through different program points (noted as ○) and split at some of these points (noted as ◇). This graph contains two safety-hole related potential crash points, indicated by colored boxes. The topmost one (orange/light grey) is labelled as *possible*, because only a portion of the execution paths through the function reach this code. The lower one (red/dark grey) is labelled as *certain*, because every execution of the function reaches this code. Of course, whether a crash occurs, depends on whether the implicit preconditions of the safety hole are satisfied. In the case of an exit safety hole, the potential crash depends on how the service uses the result of the API function. As this cannot be determined from analyzing the definition of the API function, we consider all exit safety holes to be *possible*. For more implementation details, we refer the reader to the thesis in [4].

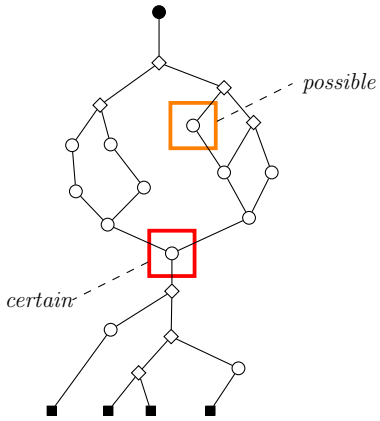

Fig. 7: Localization of *possible* and *certain* safety holes in the execution paths of an API function

3 Kernel Debugging

To further understand the difficulties that can be caused by safety holes, we consider the problem of kernel debugging. Each of the examples presented in Section 2.1 could crash the kernel. When the kernel crashes, it 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; if the hang is due to an infinite loop, the point at which an oops is finally generated may not be the same as the point that causes the loop not to terminate.

Reliability of kernel oops reports. Linux kernel backtraces suffer from the problem of *stale pointers*, either because the base pointer register is corrupted or because function return pointers appear on the stack, e.g., as function arguments or as meaningless data in the space allocated to uninitialized stack-allocated local variables. To illustrate this problem, we consider a crash occurring in the function `btrfs_init_new_device` previously shown in Figure 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 created and mounted a `btrfs` volume and then attempted to add to this volume a new device that was 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. Figure 8 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`, ostensibly 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`, so this seems likely to be a legitimate stack entry, this is not the case for `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 starting from the top of the backtrace, until finding the reason for the crash, including the definitions of functions that may be completely unrelated

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

Fig. 8: Oops report following a `btrfs` `ERR_PTR` pointer dereference crash.

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, some stale pointers remain.[2]

Relevance of kernel oops reports. A kernel oops backtrace only contains 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 Figure 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. Such situations make debugging more difficult as the developer must thoroughly consult kernel and service source code to locate possibly relevant initialization code sites.

Kernel hangs. By default, the Linux kernel gives no feedback in the case of a kernel hang. It is possible, however, to configure the kernel such that it generates a 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, and thus 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 as the result of a safety hole in a kernel internal API function. To this end, the Diagnosys approach involves three phases, as illustrated in Figure 9:

1. Identification of safety holes in kernel internal API functions and inference of the associated usage preconditions, using the static analysis tool SHAna. This phase is carried out only once by a kernel maintainer, for each new version of the mainline Linux kernel.
2. Automatic generation of a debugging interface using DIGen based on the inferred preconditions. This phase is carried out by each service developer for each specific service under development. It is transparently encapsulated in a script `dmake` that replaces the standard `make` command.
3. Testing service code with the support of the debugging interface and of the Diagnosys crash resilient logging system, CRELSys, that preserves runtime logs across boots. This phase is also carried out by the service developer. To get a benefit from Diagnosys, a small amount of investment is needed here from the developer, to learn how to interpret the log messages.

In this section we succinctly describe the design of SHAna, DIGen and CRELSys. Some implementation details are also provided.

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

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

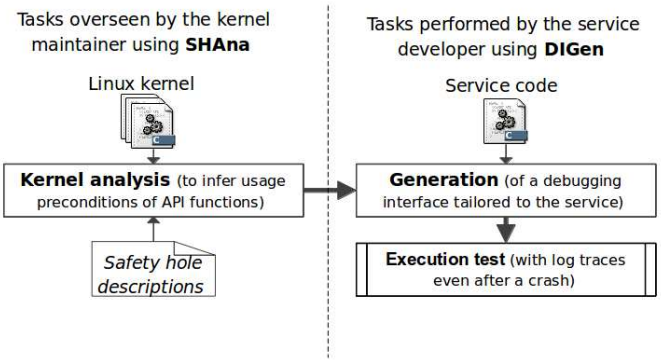

Fig. 9: The steps in using Diagnosys

4.1 SHAna: Safety Hole Analyzer

SHAna is a tool built on top of Coccinelle for identifying safety holes in kernel API functions and inferring their usage preconditions. Based on the safety hole descriptions, SHAna first searches the kernel code for occurrences of safety holes in the implementations of API functions and then computes the preconditions that must be satisfied for these safety holes to cause a kernel crash or hang. The analysis focuses on unsafe operations that occur in code that is in, or is reachable from, an API function. For each such occurrence, a backward analysis amounting to a simple version of Hoare logic [27] produces the weakest liberal precondition to be satisfied such that the safety hole may cause a crash. This weakest precondition is computed for the entry point of the function, for entry safety holes, and for the exit points of the function, for exit safety holes.

4.1.1 Theory of precondition inference

Hoare logic [26] is a formal system for reasoning about the correctness of computer programs. In particular this logic introduces the *Hoare triple* to describe how the execution of statements of an imperative program changes the state of the computation.

A Hoare triple (1) consists of two assertions, the precondition P and the postcondition Q, and a program S. Hoare logic states that: *If the execution starts in a state where the precondition P is met, at the end of the program execution the postcondition is realized* [23]. Consider the simple Hoare triple below (2) where the precondition $x = y$ is true before the execution of the program. Once the program is executed, the post-condition is true; the value of y has not been changed, and is still equal to the initial value of x, but the value of x is 3 greater than it was previously.

$$\{P\}S\{Q\} \tag{1}$$

$$\{x = y\}x := x + 3\{x = y + 3\} \tag{2}$$

This logic can be used to express function specifications, i.e., the contract between the implementation of a core API function and the client plug-in (the service code). In this case, the precondition represents the predicate that describes the condition on which the API function relies for correct operation. The postcondition on the other hand describes the condition that the API function establishes after correctly running. If some service code calls the API function after fulfilling the function's precondition, the function will execute to completion and when it terminates, the postcondition will be true.

Starting with a postcondition, the weakest precondition $wp(S, Q)$ of S with respect to Q is defined by Equation 3, where the program S is allowed to be invoked in the most general condition P that would still lead to the realization of the postcondition Q.

$$P = wp(S, Q) \Leftrightarrow \{P\}S\{Q\} \wedge \forall P', \{P'\}S\{Q\} \Rightarrow (P' \Rightarrow P) \quad (3)$$

Using a simplified version of this logic we infer the weakest preconditions of API functions that can trigger faults in the different kinds of safety holes. For ease of prototyping, we use the program matching tool Coccinelle [39] to implement an interprocedural static analyzer[3] that finds the safety holes and constructs the preconditions.

In our analysis, the postcondition is satisfied when the execution of the API function completes without leading to an execution error for entry safety holes and without returning invalid values or leaving a context that is unsafe for the execution of service code. As previously defined in Section 2.2, a precondition associated with an entry safety hole is classified as:

- *certain*, if satisfaction of the precondition is guaranteed to result in a crash or hang within the execution of the kernel API function
- *possible*, if satisfaction of the precondition may cause a crash or hang on at least one possible execution path

The preconditions of exit safety holes are always classified as *possible*. The result of SHAna is a list mapping each kernel API function identified as containing safety holes to the associated preconditions.

4.1.2 Analysis process

The starting point of the analysis for identifying safety holes and inferring their preconditions is the definition of an API function. The analysis recognizes a kernel API function as one that is declared using *EXPORT_SYMBOL* or *EXPORT_SYMBOL_GPL*. Table 2 provides a complete summary of the description of the safety holes for all categories of faults. The table also 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 API function to identify code fragments that represent safety holes. For example, in searching for the various *Lock* and *Intr* entry safety holes, SHAna only looks for interrupt disabling operations in the kernel API function itself, because interrupt state flags should not be passed from one function to another [46]. Performing only an intraprocedural analysis reduces the cost in cases where it is known in advance, due to specific properties of the code, that a more precise analysis will not give any further information.

In the case of interprocedural analysis, SHAna starts from the definition of an API function and iteratively analyzes all called functions. For example, in searching for *Null* entry safety holes, SHAna searches through both the kernel API function itself and all called functions that receive a parameter of the kernel API function as an argument to find unchecked dereferences.

In a few cases, we do not collect safety holes, because the condition seems too common and an error seems relatively unlikely. For example, according to Table 2, collecting Free entry safety holes would entail collecting every function that dereferences a pointer argument, as there is no way to check whether a value has been freed. This does not seem useful in practice, and thus SHAna does not collect safety holes in this case. In other cases, however, we augment the scope of the categories of safety holes by considering new fault kinds that may arise at the boundary of kernel API functions. For example, the fault types considered by Chou *et al.* [12] and later by Palix *et al.* [41] only include double-locks and deadlocks in the *Lock* category. Nevertheless, we have seen in mainline bug fixes that attempting to unlock a lock that has not been acquired, or that has been released (double-unlock), is a damaging fault (cf. Figure 4). Thus, we also implement an analysis to search for this kind of safety hole. Finally, for the *Null* category of safety holes, SHAna furthermore includes unchecked dereferences of values that in some way depend on the value of an unchecked parameter, and for which the failure caused by the dereference may be hard to diagnose by the service developer.

[3] The implementation of the analysis can be found at `http://diagnosys.labri.fr`

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

Table 2: Categorization of common faults in Linux [12, 41]. *f* refers to the *API function*. The "analysis type" column indicates whether the analysis is interprocedural or intraprocedural

We now discuss the specific strategies that we have implemented for searching for occurrences of the various categories of safety holes. Table 3 summarizes examples of conditions that enable the safety holes to manifest themselves into faults. SHAna bases its analysis on such *enabling conditions*.

Block: To identify API functions that contain *Block* entry safety holes, SHAna performs an interprocedural analysis, searching for functions that may block. We focus on the common case of functions that are allowed to block during memory allocations. We recognize such functions as those that contain a function call having `GFP_KERNEL` as an argument, as this argument allows the basic kernel memory functions to block, if needed, to wait for more memory to become available.

A *Block* exit safety hole amounts to taking a lock before exiting the API function. This condition is the same as the one indicating a *Lock/Intr/LockIntr* exit safety hole. The associated analysis is described in the **Lock/Intr/LockIntr** section below.

IsNull/Null: To identify API functions with *IsNull/Null* safety holes, SHAna performs an interprocedural analysis that detects unsafe dereferences of pointer-typed parameters. The search identifies dereferences that are performed prior to any validity check in one or all of the control-flow paths. We have also augmented this search strategy by detecting, in the implementation of API

Category	Entry safety hole	Enabling condition
Block	*f* calls a blocking function	service code takes a lock or disables interrupts
Null	*f* dereferences an argument without checking its validity	service code passes a `NULL` argument to *f*
Var	*f* allocates an array whose size depends on a parameter	service code calls *f* with a large value for that parameter
INull	*f* dereferences an argument without checking its validity	service code passes a `NULL` argument to *f*
Range	*f* uses an unchecked parameter to compute an array index	service code obtains a value from user level
Lock	*f* acquires lock derived from a parameter	service code re-acquires a lock from an argument to *f*
Intr	*f* disables interrupts	service code disables interrupts
Free	*f* dereferences a pointer-typed parameter value	service code calls or has called a function that may free a value
Param	*f* dereferences a pointer-typed parameter value	service code forwards a value obtained from user level

Table 3: Enabling conditions for triggering API function safety holes

functions, unsafe dereferences of other pointer values, such as *ERR_PTR* values,[4] whose validity is directly or indirectly related to the validity of an API function's unchecked parameter.

To collect API functions that exhibit an *IsNull/Null* exit safety hole, SHAna interprocedurally searches for functions that may return an invalid pointer. We recognize both `NULL` values and *ERR_PTR* values as invalid pointers.

Var: To identify API functions that implement *Var* entry safety holes, SHAna performs an intraprocedural analysis for detecting array allocations with sizes depending on a parameter value, whether it is an integer parameter value or an integer derived from a parameter.

To collect API functions with a *Var* exit safety hole, SHAna searches for functions that return large constant values. We parameterize SHAna to report constant integer values larger than 128.

Range: To identify API functions that exhibit *Range* entry safety holes, SHAna performs an intraprocedural analysis that follows the use of API function parameters to identify places where one unchecked parameter is used to compute an array index.

To collect API functions with a *Range* exit safety hole, SHAna performs an interprocedural analysis for identifying functions that return a value obtained from user-level. The search strategy relies on a list of names of Linux kernel primitives, namely `getuser`, `memcpy_fromfs`, and `copy_from_user`, that give access to user data.

Lock/Intr/LockIntr: To identify API functions that implement a *Lock/Intr/LockIntr* entry safety hole, SHAna performs an intraprocedural analysis, searching for functions that disable interrupts or acquire locks that are derived from parameters. The search strategy relies on a set of commonly used functions for locking and interrupt management.[5] We have furthermore augmented this category of safety holes by identifying API functions that attempt to release locks that they have not acquired themselves. This accounts for a type of real-world usage bug[6] that was not considered by Chou *et al.* Indeed, while many Linux critical sections both begin and end within the body of a single function, some span function boundaries, implying that some functions expect to be called within a critical section. When the calling code fails to respect this condition, deadlock may ensue.

To collect API functions that expose a *Lock/Intr/LockIntr* exit safety hole, SHAna follows the same search strategy as for *Lock/Intr/LockIntr* entry safety hole, relying on the same locking and interrupt management functions, but inverting the roles of the locking and unlocking functions.

Free: To identify API functions that exhibit a *Free* entry safety hole, we consider that any pointer-valued argument to an API function may refer to a freed memory block, thus making any dereference of a pointer-typed parameter a risky operation. Nonetheless, because practically all API functions dereference their parameters, we do not perform precondition inference for this sub-category.

To collect API functions that implement the *Free* exit safety hole, SHAna performs an interprocedural analysis for identifying code places where an API function argument is passed to the kernel memory release function `kfree`.

4.1.3 Certification Process

As is standard in static analysis, SHAna makes some approximations [15] to ensure termination and scalability. These approximations may result in false positives, *i.e.*, reported safety holes that cannot actually lead to a crash or hang. Using such false positives in the inference of API function usage preconditions may cause two important issues:

1. Inferring unnecessary usage preconditions can lead to excessive checking at runtime in the Diagnosys approach, which, in turn, will degrade the performance.

[4] We refer to a value constructed with the function *ERR_PTR* as just an *ERR_PTR* value.

[5] {*mutex,spin,read,write*}*_lock,* {*mutex,spin,read,write*}*_trylock* *local_irq_disable,* {*read,write,spin*}*_lock_irq,* {*read,write,spin*}*_lock_irqsave, local_irq_save, save_and_cli.*

[6] Linux kernel commit: `f0fbe3eb5f65fe5948219f4ceac68f8a665b1fc6`

2. False positives may also lead to the generation of useless runtime warnings, that will clutter the debug log with messages that are not relevant to any encountered crash or hang.

To address the problem of false positives, our approach requires that a kernel maintainer study the inferred safety holes to discard those that represent false positives. This step is referred to as *certification* and is performed by a kernel maintainer, who is expected to be knowledgeable about the internals of core kernel code. The certification process is only necessary once for each version of the kernel.

To reduce the workload, SHAna maintains in a database information about safety holes across OS versions, so that the kernel maintainer need only validate reported safety holes in those functions, including functions called by API functions, whose definitions have changed. To ease the management of this information, we have built a certification helper utility that, given data from a static analysis, automatically selects the analysis results related to functions whose definitions have changed, and thus need to be checked. When no certification has been performed the utility shows all results and records the result of manual checking. In a subsequent certification process, e.g., for a later release of the kernel, the certification utility first checks whether manually checked analysis results need to be revisited.

4.2 DIGen: Debugging Interface Generator

DIGen is a tool that builds on the results of SHAna to introduce runtime checks and monitoring into the execution of kernel-level services. The goal of DIGen is to create log messages that reflect the usage preconditions of kernel API functions, to keep track of any violation of these preconditions by service code.

Based on the results of SHAna, DIGen generates a debugging interface as a collection of wrapper functions that augment the definitions of kernel API 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 could 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 such a debugging interface. Accordingly, we put the interface generation process into the hands of the service developers, who generate an interface specific to their 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 a debugging interface and how it is integrated into a service under development.

4.2.1 Generating a debugging interface

The first step in the generation of a debugging interface for a given kernel-level service (e.g., a device driver) is to determine the API functions that are actually called by the service code. A reliable way to identify such functions is to analyze the object code of the kernel-level service to look for references to functions whose implementation is not included in the object code. DIGen thus disassembles the object code, using the GNU user-level command *objdump*, and recovers information about the symbol table entries of the compiled loadable kernel module (*.ko* file). Figure 10(a) provides an example of a toy Linux kernel module that tests the latency of a primitive from the *Ftrace* library [45]. Figure 10(b) shows an excerpt of the symbol table entries obtained from the object code of the module.

In each row of the symbol table,[7] the first field is a number representing the symbol value (i.e., an address). The next field is a set of characters and spaces indicating the flag bits that are set on the symbol (e.g., *l* represents *local*, *g* represents *global*, *u* represents *global unique*, etc.). Next, is

[7] More information on the format used by *GNU objdump* can be found at `http://www.gnu.org/software/binutils/`.

```
#include <linux/init.h>
#include <linux/ftrace.h>
[...]
MODULE_LICENSE("Dual BSD/GPL");
void testing_ftrace () {
    printk (KERN_EMERG "Testing Ftrace\n");
    int i=0;
    for (i=0; i<100000; i++) {
        start = rdtsc();
        trace_printk ("TRACE\n");
        end = rdtsc ();
        latency[i]=end-start;
    }
}
[...]

static int driver_init (void) {
    testing_ftrace (); return 0;
}
static void driver_exit (void) { }
module_init(driver_init);
module_exit(driver_exit);
```

a) Linux toy module for testing Ftrace

```
1  test_ftrace.ko:     file format elf32-i386
2
3  SYMBOL TABLE:
4  [...]
5  00000000 l    d  __mcount_loc 00000000 __mcount_loc
6  00000000 l    d  __versions   00000000 __versions
7  00000000 l    d  .data  00000000 .data
8  [...]
9  00000000 l    df *ABS* 00000000 test_ftrace.c
10 00000000 l     F .text  00000025 driver_exit
11 00000030 l     F .text  00000030 driver_init
12 [...]
13 00000000 l     O .modinfo      00000023 __mod_description10
14 00000023 l     O .modinfo      00000015 __mod_license9
15 00000040 l     O .modinfo      00000034 __mod_author8
16 00000000 l    df *ABS* 00000000 test_ftrace.mod.c
17 [...]
18 00000000 g     F .text  00000025 cleanup_module
19 00000000         *UND* 00000000 __trace_bprintk
20 00000030 g     F .text  00000030 init_module
21 00000000         *UND* 00000000 mcount
22 00000000         *UND* 00000000 printk
```

b) Excerpt of the symbol table entries

Fig. 10: Identification of core kernel primitives in module object code with GNU *objdump*

the section with which the symbol is associated. When this section is not connected to any other section, i.e., is absolute, the field contains instead *ABS*. However, if the section is referenced in the object file but not defined there, the field contains *UND* (undefined). In our example, the symbols *printk*, *mcount* and *__trace_bprintk*, which are defined in the core kernel and used in the tested module, are marked as undefined (lines 11, 13-14.).

Overall, the symbols listed by disassembling the object code not only include symbols that are apparent in the service code (e.g., *printk*), but also include symbols that are called from inside core kernel header files (e.g., *cleanup_module*) or that were integrated by macro expansion (e.g., *__trace_bprintk* is in the expansion of *trace_printk*). The list of undefined symbols is then checked against the list of kernel API functions. All "undefined" functions in the service code, that are known to be kernel API functions, are listed by DIGen for the next step in the generation of a debugging interface.

For each kernel API 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 function is shown in Figure 11. 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 an annotation expressing whether the violation is *CERTAIN* or *POSSIBLE* (line 5), as defined in Section 2.2. 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 and on the context (lines 9-10). As exit safety holes are always *possible*, exit safety hole log messages are simply annotated with EXIT. Finally, the return value, if any, of the original function is returned as the result (line 12).

For performance reasons, Diagnosys does not log formatted strings in kernel memory. Instead, it logs integers representing unique information identifiers that are decoded and translated on-the-fly during log retrieval. Once formatted, 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 API function, the category of the safety hole, and possibly the name of a relevant argument or an unsafe return value.

4.2.2 Integrating a debugging interface into a service

For each generated wrapper function, DIGen also generates a `#define` macro (Figure 11, line 14) that, upon expansion, will replace in the service code all calls to the kernel API function with

```
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⟩
```

Fig. 11: Structure of a wrapper for a non-void function

its debugging counterpart. Thus, once compiled with the interface included, the service uses the wrapper functions instead of the corresponding kernel API functions directly.

To facilitate the integration of a debugging interface into a kernel-level service under test, Diagnosys provides an automated script, `dmake`. This script manages the generation of the interface in four steps: `dmake` (1) compiles the original service code, (2) identifies the kernel API functions referenced by the resulting object files, (3) generates an interface dedicated to these functions, and (4) recompiles the service with the interface included. The resulting compiled kernel module object thus produced is ready for loading into a running kernel for execution tests.

4.3 CRELSys: Crash-Resilient & Efficient Logging System

To be able to use Diagnosys, the service developer must use a version of the Linux kernel that includes the Diagnosys runtime system, CRELSys. CRELSys is implemented as a kernel patch, that 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*. Finally, Diagnosys provides a tool that can be run from user space to install a copy of the CRELSys patched kernel as a crash kernel, initialize the reserved log buffer, and retrieve and format the logs. We describe first how the logs are stored in the kernel memory and then how they are preserved upon a crash.

4.3.1 Storing logs in kernel memory

When using a Diagnosys-enabled kernel, the service developers test their 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.

To reserve memory for Diagnosys runtime logs, we leverage the kernel *memmap* boot parameters,[8] which instruct the kernel to completely ignore an area of memory during its own allocations and use. Thus, CRELSys, which is aware of the location of this memory area, can use it without any risk of conflict with other parts of the kernel. CRELSys manages this reserved area of memory through a fixed-size ring buffer. When the ring buffer fills up, adding another log element overwrites the first one. The use of this reserved area of memory allows information to be maintained without involving the file system, which is slow and is not intended to be used by kernel-level code [30]. The use of a ring buffer limits the amount of memory consumed, even if many log

[8] For more details, see `https://www.kernel.org/doc/Documentation/kernel-parameters.txt`.

messages are generated. Indeed, our experience in using Diagnosys shows that debugging tasks usually only need to take into account recent events, so the information retained in the ring buffer is sufficient.

4.3.2 Preserving logs across a system reboot

On a kernel crash or hang, CRELSys redirects the execution into a *panic* state where, using a *Kexec*-based mechanism [38], the system is rebooted into a new instance of the Diagnosys-enabled kernel. Kexec is a mechanism of the Linux kernel that allows booting of a new kernel over the currently running one without resetting hardware (including RAM memory) and by skipping the bootloader phase to gain speed. The use of Kexec allows the new kernel to sit in the same place in memory as the currently running one. Because the reboot is performed without reinitializing memory, and the same *memmap* boot parameters are provided, the Kexec mechanism ensures that the accumulated Diagnosys log is still available. The service developer may then access the log messages after the reboot.

To allow the service developer to access the log messages from user space, we have implemented a character device driver that transfers data directly to and from a user process. The driver sets up a pseudo device to accept user commands (e.g., *cat /dev/crelsys*) and produce the log messages that were stored in kernel memory. 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.

5 Assessment

In designing Diagnosys, we have chosen to focus on the interface between the service code and the kernel. We first study the effect of this choice qualitatively, by considering the process of debugging some of the real-life bug examples presented in Section 2.1. We then assess the number of safety holes in this interface and study their past impact on kernel robustness, as evidenced by commits to the Linux kernel. 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. Then, we show that Diagnosys incurs a sufficiently low runtime overhead to be embedded in a service, up to the early deployment phase. Finally, we show that the certification of static analysis results is affordable.

Our experiments use code from Linux 2.6.32, which is used in the 10.04 Long Term Support version of Ubuntu®, 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 CRELSys. 1MB is reserved for CRELSys' crash-resilient log buffer.

5.1 Kernel Debugging with Diagnosys

We now describe some debugging experiments that we have performed to highlight the benefits of the Diagnosys approach. In particular, we investigate how Diagnosys solves the issues of unreliable backtraces and the questionable relevance of the information found in crash reports that make kernel debugging difficult. To this end, we replay a kernel crash from the `btrfs` file system, previously presented in Figure 2. To account for other common failures, we also replay a hang reported in the kernel commit logs, previously presented in Figure 4.

5.1.1 Replaying a kernel crash

Kernel crashes are the most common type of OS failures. When they occur, developers must identify the origin and the cause of the failure based on any generated oops reports that they manage to capture. As an example of a kernel crash, we again consider the `btrfs` example used for illustration in Section 2.1. Figure 12 shows the bug fix patch that was introduced in mainline code to fix the usage of the *open_bdev_exclusive* API function.

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

Fig. 12: Excerpt of a bug fix patch in `btrfs` file system

For the purpose of this experiment, we have recovered and installed a version of the `btrfs` file system, from right before the relevant patch was applied. The goal of the experiment was then to execute the code so that a fault would manifest itself to reflect the need for this patch. To cause *open_bdev_exclusive* to fail, we first created and mounted a `btrfs` volume and then attempted to add to this volume a new device that was not yet created. As discussed in Section 2.1, the *open_bdev_exclusive* API function then returns `ERR_PTR`, after failing to locate the device to open.

Figure 13 shows the crash report that we have collected from the kernel console at the end of the above experiment. Our previous study of this report, in Section 3, in the context of debugging, showed that the source of the problem was not readily available in the backtrace. Specifically, the backtrace contains only stale pointers, making it challenging to pinpoint the origin of the crash, and the backtrace does not contain information about the root cause of the crash.

We have replayed the same execution scenario when using Diagnosys. Figure 14 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 `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*. Using Diagnosys, service developers can focus on their own code, and do not have to probe the kernel source or object code to obtain the needed information.

5.1.2 Replaying a kernel hang

Kernel hangs are notoriously hard to debug[9] as they can simply freeze the computer leaving the developer without any information on the ongoing failure. Alternatively, the kernel can be configured to panic after a certain delay. Nevertheless, this panic may occur long after the actual fault, and thus may produce a backtrace that is hard to correlate to the actual source of the problem. In such situations, Diagnosys, which records information about previous potentially dangerous operations, can aid service developers. To illustrate the benefits of Diagnosys, we replay a bug in the the `nouveau_drm` nVidia® graphics card driver that was discussed in Section 2.1.

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 API 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.

[9] See an article at http://www.linuxjournal.com/article/5749

```
1  [  847.353202] BUG: unable to handle kernel paging request at fffffffee
2  [  847.353205] IP: [<fbc722d9>] btrfs_init_new_device+0xcf/0x5c5 [btrfs]
3  [  847.353229] *pdpt = 00000000007ee001 *pde = 00000000007ff067 *pte = 0000000000000000
4  [  847.353233] Oops: 0000 [#1]
5  [  847.353235] last sysfs file: /sys/devices/system/cpu/cpu0/cpufreq/scaling_cur_freq
6  [  847.353238] Modules linked in: btrfs zlib_deflate crc32c libcrc32c ib_iser rdma_cm ib_cm iw_cm ib_sa [...]
7  [  847.353271]
8  [  847.353274] Pid: 3699, comm: btrfs−vol Not tainted (2.6.32−diagnosis−btrfs #32) Latitude E4300
9  [  847.353276] EIP: 0060:[<fbc722d9>] EFLAGS: 00010246 CPU: 0
10 [  847.353291] EIP is at btrfs_init_new_device+0xcf/0x5c5 [btrfs]
11 [  847.353293] EAX: fffffffea EBX: fbc7cc0d ECX: f716ea80 EDX: fbc9cdc0
12 [  847.353294] ESI: fbc972a0 EDI: 00000004 EBP: f0a61eb8 ESP: f0a61e70
13 [  847.353296]  DS: 007b ES: 007b FS: 0000 GS: 00e0 SS: 0068
14 [  847.353298] Process btrfs−vol (pid: 3699, ti=f0a60000 task=ed840ca0 task.ti=f0a60000)
15 [  847.353299] Stack:
16 [  847.353301]  fbc98044 ee28e008 ee24bc00 ee31c630 f0a61ebc 00001000 fbc7b84e 00000246
17 [  847.353304] <0> f0a61ea4 00000000 00000000 f1f62c00 bff0e12c fffffffea c01c52a8 fbc7cc0d
18 [  847.353308] <0> fbc7cc0d fbc972a0 f0a61ed0 fbc7b87f bff0e12c ee24bc00 ca048334 ee28e000
19 [  847.353312] Call Trace:
20 [  847.353327]  [<fbc7b84e>] ? btrfs_ioctl_add_dev+0x33/0x74 [btrfs]
21 [  847.353334]  [<c01c52a8>] ? memdup_user+0x38/0x70
22 [  847.353349]  [<fbc7cc0d>] ? btrfs_ioctl+0x0/0x243 [btrfs]
23 [  847.353363]  [<fbc7cc0d>] ? btrfs_ioctl+0x0/0x243 [btrfs]
24 [  847.353378]  [<fbc7b87f>] ? btrfs_ioctl_add_dev+0x64/0x74 [btrfs]
25 [  847.353393]  [<fbc7cdaa>] ? btrfs_ioctl+0x19d/0x243 [btrfs]
26 [  847.353396]  [<c01f7031>] ? vfs_ioctl+0x21/0x70
27 [  847.353398]  [<c01f7672>] ? do_vfs_ioctl+0x72/0x580
28 [  847.353401]  [<c01cbe6e>] ? handle_mm_fault+0x23e/0x9d0
29 [  847.353404]  [<c01ce635>] ? unmap_region+0xe5/0x100
30 [  847.353409]  [<c0543a40>] ? do_page_fault+0x160/0x390
31 [  847.353411]  [<c01f7be7>] ? sys_ioctl+0x67/0x80
32 [  847.353414]  [<c0108583>] ? sysenter_do_call+0x12/0x28
33 [  847.353416] Code: 80 b0 1b 00 00 8b 40 6c 85 c0 74 1c c7 45 e0 01 00 00 00 8b 45 e4 83 c0 3c e8 54 [...]
34 [  847.353433] EIP: [<fbc722d9>] btrfs_init_new_device+0xcf/0x5c5 [btrfs] SS:ESP 0068:f0a61e70
35 [  847.353449] CR2: 00000000fffffffee
36 [  847.353451] −−−[ end trace 69edaf4b4d3762ce ]−−−
```

Fig. 13: Oops report following a `btrfs ERR_PTR` crash in Linux 2.6.32

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

Fig. 14: Last Diagnosys log line in the execution of `btrfs`

When we do not use the Diagnosys debugging interface, the hang leaves the developer with little information. Using Diagnosys, when the hang is detected, it causes a kernel panic, which in turn causes a reboot with CRELSys that preserves the log messages. In Figure 15, the last line of the Diagnosys log shows that *ttm_bo_wait* has been called without the expected lock held. The log message indicates the type of safety hole, the place of the offending call to the API function and the lock that needs to acquired to avoid the failure.

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

Fig. 15: Last Diagnosys log line in the execution of `nouveau_drm`

Correlating the information provided by the Diagnosys log message with the source code suggests taking the lock before the call and releasing it after the call, as shown in the Linux patch in Figure 16. This patch reflects the fix that was ultimately made in the mainline code.

```
1  commit f0fbe3eb5f65fe5948219f4ceac68f8a665b1fc6
2  if (req->flags & NOUVEAU_GEM_CPU_PREP_NOBLOCK) {
3  +   spin_lock(&nvbo->bo.lock);
4      ret = ttm_bo_wait(&nvbo->bo, false, false, no_wait);
5  +   spin_unlock(&nvbo->bo.lock);
6  }
```

Fig. 16: Patch to avoid a fault involving a *L*ock safety hole in `nouveau_drm`

5.2 Opportunities for Diagnosys

To understand the degree of opportunity for Diagnosys, we first investigate the properties of the stability of the Linux kernel-module interface, and then we consider the prevalence of safety holes in the API functions of this interface. Finally, we estimate the likely relevance of the APIs containing safety holes to potential kernel services.

We first study the evolution of the kernel API of exported functions to highlight how safety holes can be easily pass under the radar of service developers. Figure 17 presents the number of API functions exported by versions of the Linux kernel released over the course of three years (December 2008-January 2011). The graph distinguishes between the API functions already exported in the first considered version, Linux 2.6.28, and the API functions added or modified since then. The results show that the number of exported API functions has been steadily increasing. For example, Linux 2.6.37 exports 224 more API functions than Linux 2.6.36. The results also show that the set of exported API functions continuously changes [29]. Indeed, over 3,000 (25% of all) API functions present in 2.6.37 were not available in 2.6.28, while over 1,000 (10% of all) API functions present in 2.6.28 have disappeared. In addition, over 3,000 (33% of all) API functions present in 2.6.28 have been modified in 2.6.32. These changes make it difficult for service developers to keep up. Furthermore, we have found that only a small number (15%) of API functions exported by the Linux kernel are documented (Section 9 of the Linux "man" pages), and we have found that the amount of documentation of the API functions of a given release does not significantly improve over time.

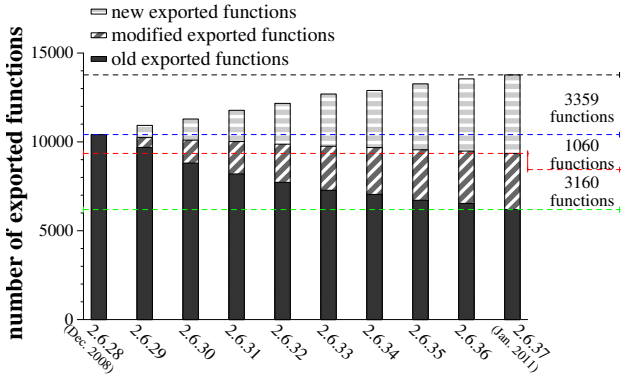

Fig. 17: Evolution in the number of Linux kernel exported API functions (baseline Linux 2.6.28)

We now investigate how widespread are the safety hole types presented in Section 2.2 (p. 5). Table 4 summarizes, for each kind of safety hole, the number of API functions exported in Linux 2.6.32 that SHAna identifies as containing at least one occurrence of that kind of safety hole. In all, SHAna reports 22,932 safety holes in 7,497 API functions. The most frequently occurring kinds are *IsNull/Null*, *Lock/Intr/LockIntr* and *Block*. Over 7,000 API functions process pointer-typed parameters without checking their validity. More than 94% of these functions perform unsafe dereferences directly within the body of their definition, while 5% forward the parameter value

to other functions that unsafely use the value with no prior check. In the *Lock/Intr/LockIntr* entry sub-category, 98% of the over 800 collected functions try to acquire a lock that has been transmitted to them via a parameter, without first making sure that they do not already hold it. The remaining 2% assume that the transmitted mutexes or spinlocks are already held in the calling context and unsafely attempt to release them.

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

Table 4: Prevalence of safety holes in Linux 2.6.32

Finally, we estimate the utility of the kernel API functions to new services, by considering the number of calls to these functions within the kernel code itself. In the 147,403 call sites across the entire kernel source code where API functions are used, half invoke a function containing a known safety hole. We have grouped all functions by category of safety hole that they contain, and computed the median number of calls to functions in each category. Depending on the kind of safety hole, the median number of calls to functions containing an entry safety hole ranges from 3 to 9, while the median number of calls to functions containing an exit safety hole ranges from 8 to 20. This suggests that the kernel API functions containing safety holes are likely to be useful to new services.

5.3 Impact of safety holes on code quality

We next investigate whether API function safety holes are effectively dangerous in kernel programming, *i.e.*, (1) if programmers write programs with bugs that are related to the presence of safety holes in API functions, and (2) if the percentage of those bugs is significant, as compared to the overall set of bugs related to the usage of API functions.

To assess the impact of the identified safety holes over the course of the development of Linux, we have searched through the commit logs of the history of Linux 2.6[10] to identify patches where the commit log mentions the kernel API functions exported in Linux 2.6.32, omitting those commits in which the function name is used as a common word (*e.g.*, "sort", "panic", etc.), to limit the number of false positives during manual processing. We have then manually reviewed these patches to identify those that are actually related to the usage of API functions, and exclude, e.g., those that affect only an API function's definition. Finally, from these relevant patches, we identify those for which the bug fix was made to account for a usage precondition as defined by SHAna. As shown in Table 5, 267 out of 703, i.e., 38%, of the usage defects are related to our categories of safety holes.

5.4 Quantitative improvement in debuggability

To be useful, Diagnosys must cover a high percentage of the misuses of kernel API functions. We first evaluate this, by artificially creating and activating misuses of API 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 then evaluate the debugging

[10] git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git - history back to 2.6.12.

Total number of commits in Linux 2.6	278,078
Commits in any way related to exported functions	11,294
Commits related to the usage of exported functions	703
Commits related to the categorized safety holes	267

Table 5: Linux kernel bug fix commits

effort, in terms of 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 6 presents those services along with the number of API functions exhibiting safety holes that they use.

Category	**Service module**	**Description**	**# of 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

Table 6: Tested Linux 2.6.32 services

Coverage of Diagnosys

To determine the coverage of Diagnosys, we assess the number of false negatives of SHAna, *i.e.*, the set of safety holes that can lead to faults in practice but are not identified by SHAna. For this, we first mutate existing services so as to artificially create bugs. Then, we inject faults at run-time to cause the mutation to trigger actual crashes across the execution of our test services.

Fault model. The largest percentage of our identified safety holes are related to `NULL` and `ERR_PTR` dereferences, and so we focus on these safety holes in our fault injection study. To devise a fault model, we consider how it can happen that such values are manipulated by kernel code. One prominent source of `NULL` and `ERR_PTR` values is to indicate the failure of 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, for the standard kernel memory allocation functions *kmalloc*, *kzalloc*, and *kcalloc*, over 8% of the calls that may fail[11] do not test the result before dereferencing the returned value or passing the returned value to another function.

Based on these observations, our fault injection 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 [13] 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

[11] Kernel allocation functions use flags to indicate whether the process can afford to have a failed allocation. Calls that are not allowed to fail have flag information containing `__GFP_NOFAIL` or `__GFP_RETRY`.

memory allocation functions multiple times, a single mutation experiment may involve multiple injected faults.

Results. Our fault-injection experiments have a number of possible results. One is that there is no observable effect. This can occur when the code initializing the tested variable does not involve a memory allocation, when the effect of the failure of the memory allocation is confined within the kernel code and does not affect the service, or when the safety hole is *possible* and is not encountered in the actual execution. Another possible result is that there is a crash, but there is no information relevant to the cause of the crash in the Diagnosys log. In this case, either the information has been overwritten in the ring buffer or SHAna has not detected the safety hole, representing a false negative. The final possible result is that there is a crash and information related to the crash is found in the Diagnosys log, representing a success for Diagnosys. In this latter case, we can further consider the position of the information relevant to the crash in the Diagnosys log. It is most helpful for the developer if this information is in the most recent entry before the crash occurred, as this position is easily identifiable.

Table 7 presents the fault injection results for 10 services implemented as kernel modules. Overall, we have performed 555 mutations. For each mutation, we have exercised the various execution paths of the affected module. 56% of the experiments have resulted in a service crash. After reboot, in 90% of the crash cases, the log contained information relevant to the origin of the defect. The table also distinguishes between cases where this information is at the last position in the log buffer and the cases where other information that is irrelevant to the crash was logged subsequently. As a metric of debuggability we use the ratio between the number of crashes for which the log contained information in the last position, and the total number of crashes. On average, Diagnosys has improved the debuggability of the service by 86%. In one case, the improvement is as low as 66%, but there are very few mutation sites in this code.

Category	**Kernel module**	**# of mutations**	**# of crashes with**			**% improved debuggability**
			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%

Table 7: Results of fault injection campaigns

Ease of debugging

In a traditional Linux kernel debugging context, a developer, provided with an oops report containing a backtrace and debugging tools that can translate stack entries into file names and line numbers, typically starts from the point of the crash, visiting all files and caller functions until the origin of the crash is localized. When the reason for the crash is in the service, but the actual crash occurs deep within the kernel, the number of functions and files to visit can be large.

To study ease of debugging quantitatively, 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. This results in 230 oops reports.

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.

Finally, we consider the impact of stale pointers on the debugging process. The considered backtraces contain an average of 5 entries that are marked as possibly stale, of which on average one appears between the entry indicating the point of crash and the entry of the function where the mutation was performed. We have furthermore assessed the improvement brought by `kdb`, and established that its backtraces contain fewer unreliable entries, but still include 2 on average.

Our assessment has also shown that kernel backtraces can miss functions, which can be attributed, in some cases, to tail call optimizations. Such corrupted stack traces can then adversely affect debugging.

5.5 Service execution overhead

The testing of preconditions and logging introduced by a Diagnosys wrapper incurs a performance overhead on the execution of a kernel-level service. This overhead must be sufficiently small to avoid interfering with the normal service execution. In this section, we evaluate the overheads introduced by the primitives used by Diagnosys to test preconditions, and investigate at a macroscopic level the impact of Diagnosys on service performance.

Penalties introduced by Diagnosys primitives

To measure the execution time of the Diagnosys precondition checking and logging operations, we have used the *Klogger* framework [19],[12] a state-of-the-art tool for performing fine grained logging of kernel operations. We also compare the execution time of a call to an exported API function having an empty body to that of a call to an exported API function containing a single precondition test. Each experimental test is run 10 times, and we compute the median value and standard deviation. Table 8 summarizes the overhead for one instance of each of the types of validity tests performed by a Diagnosys debugging interface. The observed overhead varies between 1.35% and 11.04%.

Check	**Primitive**	**Performance** (processor clock ticks)	**Overhead (%)**
Pointer validity	*IS_ERR_OR_NULL*	248.13 ± 121.24	3.12%
Spin_lock state	*spin_is_locked*	267.19 ± 121.24	11.04%
Mutex state	*mutex_is_locked*	243.88 ± 109.13	1.35%
Interrupt state	*irqs_disabled*	260.66 ± 91.34	8.32%
Performance of a call to an exported function with an empty body			240.62 ± 95.19

Table 8: Checking overhead ± standard deviation

Table 9 compares the execution time of Diagnosys' logging primitive with that of other logging mechanisms used in the kernel. *printk* is the most commonly used logging function. *Ftrace* [45]

[12] Klogger kernel patch for Linux 2.6.31.4

optimizes the logging process by deferring formatting from tracing time to output time. In Diagnosys, string formatting is not needed as the log message is generated at compile-time and is encoded as a series of integers that uniquely identify an API function, a safety hole type, etc. Diagnosys' logging primitive is thus 1.3x faster than Ftrace's *trace_printk*, and 5x faster than *printk*. In Diagnosys, the time-consuming processing tasks are performed in user space, after a reboot when the service developer attempts to display the log messages.

Logger	printk	Ftrace (trace_printk)	Diagnosys
Execution time (processor clock ticks)	3280.05 ± 82.52	884.16 ± 578.124	673.15 ± 129.26

Table 9: Performance of the Diagnosys logging primitive

Impact of Diagnosys on service performance
To understand the global performance overhead induced by the Diagnosys approach, we tested various real-world kernel services with and without a generated debugging interface.

Network driver performance. Our first test scenario involves a Gigabit Ethernet device that requires both low latency and high throughput to guarantee high performance. We evaluated the impact of a Diagnosys debugging interface by exercising the *e1000e* Linux device driver using the `TCP_STREAM`, `UDP_STREAM` and `UDP_RR` tests from the *netperf* benchmark [28]. For these experiments, the *netperf* utility was configured to report results accurate to 5% with 99% confidence. Table 10 summarizes the performance and CPU overhead for the *e1000e* driver when it is run without and with a debugging interface. The debugging interface only reduces the throughput by between 0.4% and 6.4%, and increases the CPU utilization by between 0.4% and 10%. Nevertheless, while small, the existence of this overhead suggests why kernel developers would not want to systematically implement API functions such that they always perform all of these checks. This shows the need for a pluggable debugging interface dedicated to a service under development, as provided by Diagnosys.

Test		**Without Diagnosys**	**With Diagnosys**	**Overhead**
`TCP_STREAM`	Throughput	907.91 Mb/s	904.32 Mb/s	0.39%
	CPU	52.57%	58.48%	10.10%
`UDP_STREAM`	Throughput	951.00 Mb/s	947.73 Mb/s	0.34%
	CPU	58.92%	65.45%	9.98%
`UDP_RR`	Throughput	7371.69 Tx/s	6902.81 Tx/s	6.36%
	CPU	55.19%	55.37%	0.33%

Table 10: Performance of the e1000e driver

File system performance. Our second test scenario involves the NFS file system, whose implementation uses about 200 exported functions exhibiting safety holes. The experiment consists of sequential read, sequential write/rewrite and random seek phases based on patterns generated by the Bonnie benchmark [10]. For this experiment, the client and server run on the same machine, connected using a network loopback interface, to eliminate the network transmission time. During a run of this benchmark with a debugging interface integrated into the `NFS` file system, we have recorded over 48,000,000 calls to the interface wrapper functions to write and read 8G of data. As shown in Table 11, for data transfers of only one character, amounting to 1 byte, the overhead can be significant, of up to 67%. For block reads and writes, however, the overhead is only up to 17%, and for random seeks and sequential rewrites it is under 3%.

Test		Without Diagnosys (Access rate - K/sec)	With Diagnosys (Access rate - K/sec)	Overhead
Sequential reads	per char	930	642	30.9%
	per block	28795	23811	17.3%
Sequential writes	per char	494	162	67.2%
	per block	42467	38329	9.7%
Sequential rewrites		13647	13327	2.3%
Random seeks		2145	2143	0.9%

Table 11: Performance of the NFS file system

5.6 Certification overhead of analysis results

Although SHAna only needs to be run once per kernel release, the number of results that it returns still makes certification of its results potentially very expensive. Diagnosys includes several techniques to reduce the amount of certification needed.

Highlighting likely false positives. In practice, a major source of false positives is when multiple definitions are provided for a given function, selectable by different, incompatible configuration options. In this case, interprocedural analysis can detect a safety hole that involves the interaction between two definitions that cannot coexist in an actual kernel. When SHAna detects that this is a possibility, due to the existence of multiple definitions of a called function, it annotates the derived safety holes as potential false positives. SHAna also provides information about the file in which the function instance inducing the safety hole is defined.

Of the 22,940 safety holes reported by SHAna for Linux 2.6.32, SHAna itself annotated 465 (2%) as potential false positives, because of the ambiguity of the identification of called functions during interprocedural analysis. Since the Linux kernel provides different definitions of some functions for different architectures, these different definitions may exhibit different safety holes, and therefore results with respect to such functions require thorough validation. At a rate of about 5 minutes per safety hole, we estimate that this certification requires about a week of work (38 hours). Of the 465 potential safety holes, we have found that 405 (87%) are actual false positives.

We have also manually reviewed all of the other safety holes reported by SHAna for Linux 2.6.32. Among the reported safety holes that were not annotated as potential false positives during the analysis, we have identified some cases for which misuse seems very unlikely. For example, some lock-related exported functions such as *unlock_rename* clearly indicate their purpose in their name. Similarly, *clk_get_rate* may return a large integer, but it seems unlikely that a developer would use this integer to declare the size of an array. We have found 9 such false positives in Linux 2.6.32. Most of the associated functions are called fewer than 5 times, with the most frequently used, *clk_get_rate*, being called 144 times. Thus, given the small rate of these safety holes and the low usage of the associated functions, we consider that it is sufficient for the kernel maintainer to manually check only the safety holes that are actually annotated as potential false positives by SHAna.

Preserving certification information across kernel versions. To further reduce the certification overhead, SHAna maintains information about safety holes across OS versions, so that the kernel maintainer need only check reported safety holes in those functions whose definitions have changed. To demonstrate the potential benefit of this information, we have also checked the safety holes that SHAna has annotated as potential false positives in 5 versions that were released after Linux 2.6.32. As shown in Figure 18, the burden on the maintainer is significantly reduced when data from a previous certification are available. Between two certification processes, the workload can drop by 50 to 95%, often to around a day or less, depending on the amount of time elapsed since the release of the previously certified version.

It may be the case that the maintainer is not available to run SHAna and perform the certification for every release. The blue dashed line in Figure 18 shows that even performing a certification

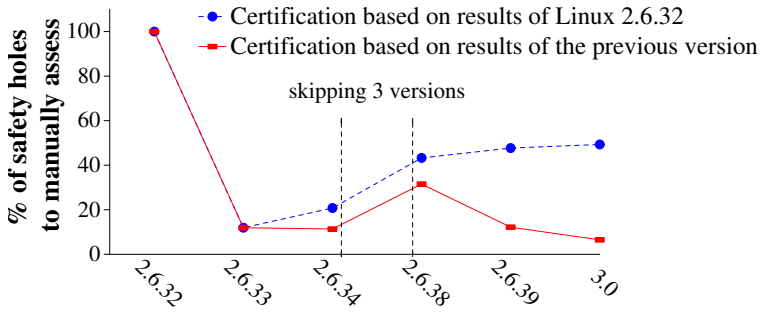

Fig. 18: Certification overhead

based on a previous, but not preceding, version can still substantially reduce the maintainer workload.

In assessing the certification overhead, we have only considered the costs of certifying the results from the mainline kernel. In practice, most users do not use the mainline kernel, but instead one prepared for a particular Linux distribution, such as Debian or Fedora. While such a kernel may diverge slightly from the mainline, the differences are unlikely to affect the kernel API. Furthermore, a service that should ultimately be integrated into the mainline kernel must be developed against the mainline kernel's API.

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 [24]. Ten years ago, Chou *et al.* found that the fault rate in Linux drivers was 3–7 times higher than that of other parts of the kernel [12]. Palix *et al.* have shown that while this error rate is decreasing, Linux drivers still contain many defects [40]. They have also found that file systems have recently had a high fault rate, indeed even higher than that of drivers. More generally on software code, empirical studies show that software bugs are becoming pervasive, specially in large projects with large code base and large development teams [5], for all programming languages [9].

System robustness testing. Fault injection has been applied to the Linux kernel to evaluate the impact of various fault classes [1,14]. Our work identifies the safety holes in kernel interfaces that explain their observations. Marinescu and Candea [35] 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 [3,17,32,43]. 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 [17,32,33,44]. 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 [11]. They focus, however, on the entire kernel, and thus may inform service developers about faults in code other than their own.

Logging. Runtime logs are frequently insufficient for failure diagnosis especially in case of unexpected crashes [49]. *LogEnhancer* [50] 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 [34] 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 [20]. 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 [22,25,36,37]. 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 [18]. 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 service developers would have to actively use Daikon in their own environment. SHAna, in contrast, allows the collection of safety holes to be centralized. Finally, only one of the invariants in the Daikon invariant list,[13] 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) [22] relies on programmer annotations to check method contracts. Annotation assistants such as Houdini [21] 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.

Finally, Parnin and Orso have presented a user study involving actual developers to investigate whether automated debugging tools really help developers [42]. They found, for instance, that expert developers were faster when using the tool, although the tool did not help perform harder tasks. In our assessment, we sought to demonstrate that Diagnosys provides debugging information that is useful even for a novice programmer, and that it helps in pinpointing the root cause even in cases where the root cause is hidden in the stack trace of a crash.

Programming using DSLs Domain-Specific languages provide concise syntax and rich semantics for allowing domain experts, who may not be traditional programmers, to write programs for a given domain. There are claims in the literature that DSL programs are easy to use and maintain [48]. Our own experience in designing DSLs [6] has led us to note that the maintainability issues are shifted in the implementation of the DSL compiler. We have then proposed to use transformation rules to implement embedded compilers so as to improve maintainability [8]. However, a more general study by Donahue found that easy maintainability is not an intrinsic property of DSLs [16].

[13] 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 operating system, often leaving developers without any information about what went wrong. In the Linux kernel, for example, one significant difficulty in developing drivers is that the kernel does not export a debugging interface to its internal functionalities [29]. Many of the functions that are exported to external modules have implicit ill-documented preconditions, which, if not satisfied, can cause the entire system to crash or hang.

In this paper, we have presented a new approach for supporting kernel-level service developers in the early stages of service development. Diagnosys was designed and implemented as an approach to automatically constructing a debugging interface for the Linux kernel. The tool detects safety holes in Linux kernel-level API 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 developers to the critical defects in their 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.

Availability: Materials for this paper, including more detailed information on the analysis specifications, the Diagnosys tool, and the certified versions of the kernel can be found on the project webpage - `http://diagnosys.labri.fr`.

Acknowledgements: This work was supported in part by the ANR Blanc grant ABL.

References

1. Albinet, A., Arlat, J., Fabre, J.C.: Characterization of the impact of faulty drivers on the robustness of the Linux kernel. In: DSN'04: Proceedings of the 2004 International Conference on Dependable Systems and Networks, pp. 867–876. Florence, Italy (2004)
2. Arnout, K., Meyer, B.: Uncovering hidden contracts: The .NET example. Computer **36**, 48–55 (2003)
3. Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., McGarvey, C., Ondrusek, B., Rajamani, S.K., Ustuner, A.: Thorough static analysis of device drivers. In: EuroSys'06: Proceedings of the 2006 ACM SIGOPS/EuroSys European Conference on Computer Systems, pp. 73–85. Leuven, Belgium (2006)
4. Bissyandé, T.F.: Contributions for improving debugging of kernel-level services in a monolithic operating system. Ph.D. thesis, Université Sciences et Technologies-Bordeaux I (2013)
5. Bissyandé, T.F., Lo, D., Jiang, L., Réveillère, L., Klein, J., Le Traon, Y.: Got issues? who cares about it? a large scale investigation of issue trackers from github. In: IEEE 24th International Symposium on Software Reliability Engineering, ISSRE
6. Bissyandé, T.F., Réveillère, L., Bromberg, Y.D., Lawall, J.L., Muller, G.: Bridging the gap between legacy services and web services. In: Proceedings of the ACM/IFIP/USENIX 11th International Conference on Middleware, Middleware '10, pp. 273–292. Springer-Verlag, Bangalore, India (2010)
7. Bissyandé, T.F., Réveillère, L., Lawall, J.L., Muller, G.: Diagnosys: automatic generation of a debugging interface to the linux kernel. In: ASE'12: Proceedings of 27th IEEE/ACM International Conference on Automated Software Engineering, pp. 60–69. Essen, Germany (2012)
8. Bissyandé, T.F., Rveillère, L., Lawall, J.L., Bromberg, Y.D., Muller, G.: Implementing an embedded compiler using program transformation rules. Software: Practice and Experience pp. n/a–n/a (2013)
9. Bissyandé, T.F., Thung, F., Lo, D., Jiang, L., Réveillère, L.: Popularity, interoperability, and impact of programming languages in 100,000 open source projects. In: Proceedings of the 37th IEEE Annual Computer Software and Applications Conference, COMPSAC '13, pp. 303–312. Washington, DC, USA (2013)
10. Bray, T.: The Bonnie file system benchmark. `http://www.textuality.com/bonnie/`
11. Bugrara, S., Aiken, A.: Verifying the safety of user pointer dereferences. In: IEEE Symposium on Security and Privacy, pp. 325–338. Oakland, CA, USA (2008)
12. Chou, A., Yang, J., Chelf, B., Hallem, S., Engler, D.: An empirical study of operating systems errors. In: SOSP'01: Proceedings of the 18th ACM Symposium on Operating System Principles, pp. 73–88. Banff, Canada (2001)
13. Corbet, J.: Injecting faults into the kernel. `http://lwn.net/Articles/209257/` (2004)
14. Cotroneo, D., Natella, R., Russo, S.: Assessment and improvement of hang detection in the Linux operating system. In: SRDS'09: Proceedings of the 28th IEEE International Symposium on Reliable Distributed Systems, pp. 288–294. Niagara Falls, NY, USA (2009)

15. Dillig, I., Dillig, T., Aiken, A.: Reasoning about the unknown in static analysis. Communications of the ACM **53**(8), 115–123 (2010)
16. Donahue, A.: Debugging domain-specific languages. Master's thesis, University of Toronto (2010)
17. Engler, D., Chen, D.Y., Hallem, S., Chou, A., Chelf, B.: Bugs as deviant behavior: a general approach to inferring errors in systems code. In: SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles, pp. 57–72. Banff, Alberta, Canada (2001)
18. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The Daikon system for dynamic detection of likely invariants. Sci. Comput. Program. **69**, 35–45 (2007)
19. Etsion, Y., Tsafrir, D., Kirkpatrick, S., Feitelson, D.G.: Fine grained kernel logging with KLogger: experience and insights. In: EuroSys, pp. 259–272. Lisbon, Portugal (2007)
20. Fetzer, C., Xiao, Z.: Healers: a toolkit for enhancing the robustness and security of existing applications. In: DSN'03: Proceedings of the 2003 International Conference on Dependable Systems and Networks, pp. 317–322. San Francisco, CA, USA (2003)
21. Flanagan, C., Leino, K.R.M.: Houdini, an annotation assistant for ESC/Java. In: FME, pp. 500–517. London, UK (2001)
22. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: PLDI'02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation, pp. 234–245. Berlin, Germany (2002)
23. Frade, M.J., Pinto, J.S.: Verification conditions for source-level imperative programs. Computer Science Review **5**(3), 252–277 (2011)
24. Ganapathi, A., Ganapathi, V., Patterson, D.: Windows XP kernel crash analysis. In: LISA'06, pp. 49–159. Washington, DC, USA (2006)
25. Hirschfeld, R., Perscheid, M., Schubert, C., Appeltauer, M.: Dynamic contract layers. In: SAC'10: Proceedings of the 2010 Symposium on Applied Computing, pp. 2169–2175. Sierre, Switzerland (2010)
26. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969)
27. Huth, M., Ryan, M.: Logic in Computer Science: Modelling and reasoning about systems. Cambridge University Press (2000)
28. Jones, R.: Netperf: A network performance benchmark, version 2.4.5. `http://www.netperf.org`
29. Kroah-Hartman, G.: The Linux kernel driver interface (all your questions answered and then some). `http://www.kernel.org/doc/Documentation/stable_api_nonsense.txt`
30. Kroah-Hartman, G.: Driving me nuts - things you should never do in the kernel. Linux Journal (133), 9 (2005). URL `http://www.linuxjournal.com/article/8110`
31. Kuznetsov, V., Chipounov, V., Candea, G.: Testing closed-source binary device drivers with DDT. In: ATC'10: USENIX Annual Technical Conference. Boston, MA, USA (2010)
32. Lawall, J.L., Brunel, J., Palix, N., Hansen, R.R., Stuart, H., Muller, G.: WYSIWIB: A declarative approach to finding API protocols and bugs in Linux code. In: DSN'09: Proceedings of the 2009 International Conference on Dependable Systems and Networks, pp. 43–52. Lisbon, Portugal (2009)
33. Li, Z., Zhou, Y.: Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code. In: ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering, pp. 306–315. Lisbon, Portugal (2005)
34. Mao, Y., Chen, H., Zhou, D., Wang, X., Zeldovich, N., Kaashoek, M.F.: Software fault isolation with API integrity and multi-principal modules. In: SOSP, pp. 115–128. Cascais, Portugal (2011)
35. Marinescu, P., Candea, G.: Efficient testing of recovery code using fault injection. ACM Transactions on Computer Systems (TOCS) **29**(3) (2011)
36. Meyer, B.: Object-Oriented Software Construction, 1st edn. Prentice-Hall, Inc. (1988)
37. Mills, C.: Using Design by Contract in C, 1st edn. OnLamp.com, O'Reilly (2004)
38. Nellitheertha, H.: Reboot Linux faster using kexec. `http://www.ibm.com/developerworks/linux/library/l-kexec/index.html` (2004)
39. Padioleau, Y., Lawall, J.L., Hansen, R.R., Muller, G.: Documenting and automating collateral evolutions in Linux device drivers. In: EuroSys'08: Proceedings of the 2008 ACM SIGOPS/EuroSys European Conference on Computer Systems, pp. 247–260. Glasgow, Scotland (2008)
40. Palix, N., Lawall, J., Muller, G.: Tracking code patterns over multiple software versions with herodotos. In: AOSD'10: Proceedings of the 2010 International Conference on Aspect-Oriented Software Development, pp. 169–180. Rennes and Saint-Malo, France (2010)
41. Palix, N., Saha, S., Thomas, G., Calvès, C., Lawall, J.L., Muller, G.: Faults in Linux: Ten years later. In: ASPLOS'11: Proceedings of the 2011 International Conference on Architectural Support for Programming Languages and Operating Systems. Newport Beach, CA, USA (2011)
42. Parnin, C., Orso, A.: Are automated debugging techniques actually helping programmers? In: Proceedings of the 2011 International Symposium on Software Testing and Analysis, pp. 199–209. ACM (2011)
43. Post, H., Küchlin, W.: Integrated static analysis for Linux device driver verification. In: IFM'07: Proceedings of the 6th international conference on Integrated formal methods, pp. 518–537. Oxford, UK (2007)
44. Ramanathan, M.K., Grama, A., Jagannathan, S.: Path-sensitive inference of function precedence protocols. In: ICSE '07: Proceedings of the 29th international conference on Software Engineering, pp. 240–250. Minneapolis, MN, USA (2007)
45. Rostedt, S.: Debugging the kernel using ftrace. `http://lwn.net/Articles/365835/` (2009)
46. Rubini, A., Corbet, J.: Linux Device Drivers, second edn., p. 109. O'Reilly Media (2001)

47. Ryzhyk, L., Chubb, P., Kuz, I., Heiser, G.: Dingo: Taming device drivers. In: EuroSys'09: Proceedings of the 2009 ACM SIGOPS/EuroSys European Conference on Computer Systems, pp. 275–288. Nuremberg, Germany (2009)
48. Strembeck, M., Zdun, U.: An approach for the systematic development of domain-specific languages. Softw. Pract. Exper. **39**(15), 1253–1292 (2009)
49. Yuan, D., Mai, H., Xiong, W., Tan, L., Zhou, Y., Pasupath, S.: Sherlog: Error diagnosis by connecting clues from run-time logs. In: ASPLOS'10: Proceedings of the 2010 International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 143–154. Pittsburgh, PA, USA (2010)
50. Yuan, D., Zheng, J., Park, S., Zhou, Y., Savage, S.: Improving software diagnosability via log enhancement. In: ASPLOS, pp. 3–14. Newport Beach, CA, USA (2011)