Generation of Debugging Interfaces for Linux Kernel Services

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

Generation of Debugging Interfaces for Linux Kernel Services

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

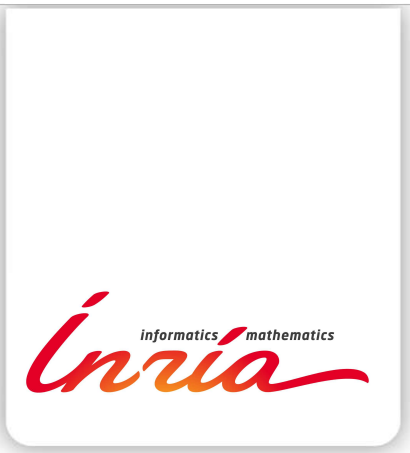
RESEARCH REPORT

N° 7800

November 2011

Project-Team Regal

ISSN 0249-6399 ISRN INRIA/RR--7800--FR+ENG



Generation of Debugging Interfaces for Linux Kernel Services

Tegawendé F. Bissyandé*, Laurent Réveillère*, Julia L. Lawall†, Gilles Muller †

Project-Team Regal

Research Report n° 7800 — November 2011 — 25 pages

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, these are continually changing, and have implicit, ill-documented preconditions, which, if not satisfied, can cause the entire system to crash or hang. However, no specific debugging support is provided.

In this paper, we present *Diagnosys*, an approach to automatically constructing a debugging interface for the Linux kernel. In our approach, a designated kernel maintainer uses Diagnosys to identify pre and post conditions on the use of the exported functions. The maintainer then publishes this information for download by service developers, who can use Diagnosys to generate a debugging interface specialized to their code. This interface is then included within a service implementation, such that when the service is tested it records information about potential problems. The recorded information is then made available to the service developer on reboot after a kernel crash or hang.

We have tested our approach on several recent releases of the Linux kernel. We first show that the debugging interfaces generated by our approach provide useful log information in the case of faults. We then show that safety holes are prevalent in the Linux kernel, and that many of the functions exported by the Linux kernel to kernel-level services have non-trivial implicit pre and post conditions that must be respected to ensure their safe execution. Finally, we show that our approach incurs only a slight performance penalty.

Key-words: Linux kernel, automatic generation, static analysis, driver development

* LaBRI, University of Bordeaux
† LiP6, INRIA/REGAL

RESEARCH CENTRE
PARIS – ROCQUENCOURT

Domaine de Voluceau, - Rocquencourt
B.P. 105 - 78153 Le Chesnay Cedex

Génération d'interfaces d'aide à la mise au point pour les services noyau de Linux

Résumé : Le noyau de Linux ne définit pas une interface de programmation précise et stable, ce qui complique le dévelopment des services noyau tels que les pilotes de périphériques et les systèmes de fichiers. Bien qu'il existe un ensemble de fonctions exportées pour utilisation par les modules du noyau, celles-ci sont continuellement modifiées et présentent des préconditions d'usages à la fois implicites et peu documentées, et qui, lorsqu'elles ne sont pas satisfaites, peuvent entrainer un crash ou un blocage de tout le système d'exploitation. Cependant, Linux n'offre aucun moyen spécifique de débogger de tels problèmes.

Dans cet article, nous présentons *Diagnosys*, une approche pour construire automatiquement une interface d'aide à la mise au point des services noyau de Linux. Dans notre approche, un mainteneur du noyau utilise Diagnosys pour identifier des pré et post conditions sur l'utilisation des fonctions exportées. Le mainteneur met ensuite à disposition cette information qui sera téléchargée par des développeurs de services qui utilisent à leur tour Diagnosys pour générer une interface spécialisée à leur code. Cette interface est ainsi ajoutée au code de leur service de sorte que lors du test du service, elle enregistre des informations sur les problèmes potentiels. L'information enregistrée est ensuite disponible après rédémarrage du noyau.

Nous avons testé notre approche sur plusieurs versions du noyau de Linux. Nous montrons d'abord que les interfaces générées par notre approche fournissent des informations de journalisation utiles en cas de fautes. Nous montrons ensuite que les portions de code représentant des lacunes de sécurité sont répandues dans le noyau de Linux, et que de nombreuses fonctions exportées pour les services noyau de Linux présentent des pré et post conditions implicites et non triviales qui doivent être respectées pour assurer une exécution sûre. Finalement, nous montrons que notre approche induit uniquement de faibles pertes de performation.

Mots-clés : Le noyau linux, génération automatique, analyse statique

1 Introduction

The Linux development process is oriented around the assumption that the source code of all kernel-level services, such as device drivers, file systems, and network protocols, is available within the publicly available kernel source tree. Given this assumption, internal kernel APIs need only be as robust as required by their internal client services, avoiding the overhead of extra tests. Furthermore, developers can freely adjust the internal APIs, as long as they are willing to update all of the affected service code. The code is maximally efficient and evolvable, enabling the kernel to rapidly meet new performance requirements, address security issues, and accommodate new functionalities.

While this situation is beneficial for the state of the Linux kernel as a whole, it may not be ideal for the developers of new services who require more safety and help in debugging. When code is developed outside the Linux kernel source tree, the developer is on his own, with no help from kernel experts and little documentation. Instead, he must study the kernel source code to identify the various relevant kernel internal API functions and then study their definitions, as well as potentially the definitions of all of the functions they call, to identify constraints on their usage. Overlooking or misunderstanding these constraints can cause the kernel to crash or hang in ways that are difficult to debug. Advances in bug-finding tools [10, 20, 23], specialized testing techniques [19, 24], and code generation from specifications [32] have eased but not yet fully solved these difficulties.

We concretize the difficulty confronting a Linux service developer 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 internal API function that dereferences a parameter without testing its value represents a safety hole, because a service could invoke the function with `NULL` as the corresponding argument. Likewise, code in the definition of a kernel internal API function that returns `NULL` as the result represents a safety hole, because a calling service could dereference this result without checking its value.

To address the problem of safety holes in Linux kernel internal API functions, we propose an approach, named *Diagnosys*, that automatically generates a debugging interface tailored for a particular kernel-level service under development, based on a static analysis of the Linux kernel source code. A service developer includes this interface with the service code. On execution of the service, the interface generates log messages whenever service code uses a kernel internal API function that contains a safety hole in a potentially risky way. Such a debugging interface is well-suited to intensive service development, when the developer is modifying the code frequently, and bugs are likewise frequent. Because a Diagnosys debugging interface is automatically generated, it can be regenerated for each new version of the Linux kernel, as the properties of the internal APIs change. To limit the runtime overhead, the generated debugging interface is localized at the boundary of the interaction between the service and the OS kernel. This strategy furthermore focuses the feedback provided by the interface on the part of the code that the developer has written, and thus is expected to be the most familiar with.

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 that is provided as a kernel patch. SHAna is run by a Linux kernel maintainer once for each Linux version, to take into account the current definitions of the Linux kernel internal API functions. DIGen is run by a service developer as part of the service compilation process. During the execution of the resulting service, the debugging interface uses the runtime system to log information in a crash-resilient buffer about any unsafe uses of functions containing safety holes. On a kernel crash or hang, the service developer can subsequently consult the buffer to obtain the logged information for use in the debugging process. Our work thus goes in the direction of recent work that has focused on various aspects of improving the usefulness of run-time log messages [12, 34, 36].

The main results of this paper are as follows:

- We assess the improvement in the experience of debugging a fault using our generated debugging interface, as compared to existing tools. A Diagnosys debugging interface gives information about previously executed potentially dangerous operations that may lead to a crash or hang. This information is complementary to the information already provided by the kernel.
- We demonstrate the overall need for our approach by identifying the number of faults observed in mainline kernel code derived from safety holes. Of the Linux 2.6 patches for which the changelog refers explicitly to one of the functions exported in Linux 2.6.32, 38% corrected faults related to one of our identified safety holes.
- Using fault injection experiments designed to trigger invalid pointer dereferences, we find that in 90% of the cases in which a crash occurs, the log contains information relevant to the origin of the defect. In 95% of these cases, a log message relevant to the crash appears in the last position of the log, making it easy for the developer to identify.
- We demonstrate that our approach imposes only a moderate burden on the kernel maintainer. Out of 22,940 reported safety holes in Linux 2.6.32, we find that it is sufficient for the kernel maintainer to check only 465, involving ambiguous definitions, for false positives.
- We show that the generated debugging interface incurs only a minimal runtime overhead on service execution. Using the netperf benchmark, we observe overheads of up to 10% for a Gigabit Ethernet device, and using the Bonnie file system benchmark we observe overheads of up to 17% for block reads or writes.

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 presents Diagnosys, including the process of collecting information about the occurrences of safety holes and their associated preconditions, and the process of generating

a debugging interface. Section 4 evaluates our approach. Finally, Section 5 discusses related work, and Section 6 concludes.

2 Safety Holes

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

2.1 Examples of safety holes

The Linux kernel does not define a precise internal API. Thus, in identifying safety holes, 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 testing of new service versions. We refer to kernel functions that are made available to such modules as *kernel exported functions.*

Figure 1a shows an excerpt of the definition of the kernel exported function *skb_put* for which the first argument has pointer type. This function dereferences its first argument without first checking its value. Many kernel functions are written in this way, with the assumption that all provided arguments are valid. This code nevertheless represents a safety hole, because the dereference is invalid if the corresponding argument is `NULL`. Such a fault occurred in Linux 2.6.18 in the file *drivers/net/forcedepth.c* in the function *nv_loopback_test*, in which *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 test this result prior to calling *skb_put*, and to avoid calling *skb_put* if it turns out to be `NULL`. The definition of *skb_put* remains unchanged.

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

a) Excerpt of the definition of *skb_put*

```
1  commit 46798c897e235e71e1e9c46a5e6e9adfffd8b85d
2     x2tx_skb = dev_alloc_skb(pkt_len);
3  +  if (!tx_skb) { ... goto out; }
4     pkt_data = skb_put(tx_skb, pkt_len);
```

b) Excerpt of the bug fix patch

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

As another example, 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. Such a value has pointer type, but any attempt to dereference it will crash the kernel. Thus, this return statement also represents a safety hole. In Linux 2.6.32, in the file *fs/btrfs/volumes.c*, the function *btrfs_init_new_device* called *open_bdev_exclusive* and tested the result for being `NULL` before dereferencing the value. This test, however, was not sufficient to prevent a kernel crash, because a value created using `ERR_PTR` is different from `NULL`. The fault was fixed by testing for an `ERR_PTR` value rather than for `NULL`, by the patch shown in Figure 2b.

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

a) Excerpt of the definition of *open_bdev_exclusive*

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

b) Excerpt of the bug fix patch

Figure 2: Bug fix of error handling code

In the previous cases, the safety hole is apparent in the definition of a kernel exported function. In other cases, however, a safety hole may be interprocedural, and the danger that it poses may be 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.

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

a) Excerpt of *kmap*

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

b) Excerpt of *page_to_nid*

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

c) Excerpt of the bug fix patch

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

2.2 Taxonomy of safety holes

As illustrated above, some fragments of code executed by kernel exported functions, while themselves being correct, can provoke kernel crashes or hangs when

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

Table 1: Categorization of common faults in Linux [5]. EF refers to the *exported function*. The *inter* and *intra* qualifiers indicate whether the analysis is interprocedural or intraprocedural

the function is used incorrectly. We distinguish between *entry* safety holes, in which the crash or hang is provoked within the execution of the kernel exported function, due to an invalid argument provided by the service, and *exit* safety holes, in which the crash or hang is provoked within the execution of the service due to a possible effect of the kernel exported function that the service has not taken into account.

As a source of kinds of safety holes, we consider the faults kinds identified by Chou *et al.* in their 2001 study of Linux code [5]. A fault is not in itself a safety hole, because the faulty code can be completely contained within a single function definition. Likewise, a safety hole is not in itself a fault, as illustrated by the above examples. Nonetheless, we observe that many fault kinds involve a sequence of code fragments. For example, a NULL pointer dereference fault typically involves an initialization of a variable to NULL followed by a dereference of this variable. This observation leads to a methodology for translating fault kinds into kinds of safety holes. When the suffix of a sequence of code fragments associated with a fault kind is found in a kernel exported function and depends in some way on the calling context, such as the argument values, of that function, then that suffix represents an entry safety hole. Likewise, when a prefix of such a sequence is found in a kernel exported function and has some impact on the result of that function, then that prefix represents an exit safety hole.

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

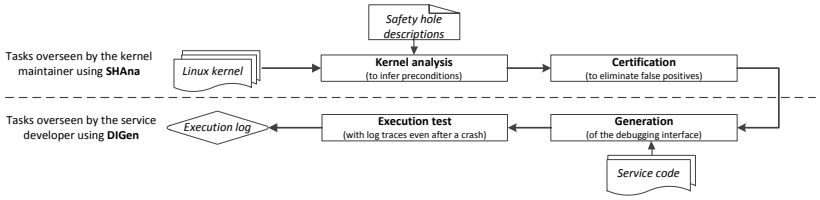


Figure 4: The steps in using Diagnosys

2.3 Safety hole preconditions

From a collection of safety holes, our goal is to create a debugging interface that informs the service developer of possibly dangerous uses of kernel exported functions within his code. Nevertheless, merely invoking a kernel exported function that contains an entry or exit safety hole does not necessarily cause a fault. Instead, some properties of the argument or return values, such as the presence of a `NULL` value, must typically be satisfied. Thus, we design Diagnosys such that its static analysis tool SHAna identifies not just safety holes, but also the preconditions that must be satisfied for a fault to be possible [16]. The safety holes and associated preconditions are then used in generating the debugging interface.

3 Diagnosys

The use of Diagnosys involves four phases: 1) identification of safety holes in kernel exported functions and inference of the associated preconditions, using the static analysis tool SHAna, 2) certification of the inferred preconditions by a kernel maintainer, 3) automatic generation of a debugging interface using DIGen based on the inferred preconditions, and 4) testing service code with the support of the debugging interface. The first two phases are carried out by a kernel maintainer, for each new version of the Linux kernel, and the last two phases are carried out by a service developer. These phases are illustrated in Figure 4.

3.1 Identifying safety holes and their preconditions

SHAna first searches the kernel code for occurrences of safety holes in the implementations of exported 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 reachable from an exported function. For each such occurrence, a backward analysis amounting to a simple version of Hoare logic [16] produces the weakest precondition to be satisfied on entry to the function, for entry safety holes, and on exit from the function, for exit safety holes, such that the safety hole may cause a crash. 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 exported function, or *possible*, if satisfaction of the precondition may cause a crash or hang on at least one possible execution path.

The result of SHAna is a list mapping each kernel exported function identified as containing safety holes to the associated preconditions.

The analysis starts from the definition of an exported function, recognized as one that is declared using *EXPORT_SYMBOL* or *EXPORT_SYMBOL_GPL*. Table 1 indicates for each category of safety hole, whether intraprocedural, interprocedural or no analysis is used. In search scenarios that only require intraprocedural analysis, the analyzer scans the definition of the exported function to identify code fragments that represent safety holes. For example, in searching for the various Lock and Intr entry safety holes, SHAna only looks for interrupt disabling operations in the kernel exported function itself, because interrupt state flags should not be passed from one function to another [31]. In the case of interprocedural analysis, SHAna starts from the definition of an exported function and iteratively analyzes all called functions. For example, in searching for Null entry safety holes, SHAna searches through both the kernel exported function itself and all called functions that receive a parameter of the kernel exported function as an argument to find unchecked dereferences. SHAna furthermore includes unchecked dereferences of values that in some way depend on the value of an unchecked parameter. In a few cases, we do not collect safety holes, because the condition seems too common and an error seems relatively unlikely. For example, 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.

3.2 Certifying preconditions

As is standard in static analysis, SHAna makes some approximations 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 construction of an interface would cause it to generate unnecessary log messages, which could degrade performance and clutter the 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. To reduce the workload, SHAna maintains information about safety holes across OS versions, so that the kernel maintainer need only validate reported safety holes in those functions whose definitions have changed.

3.3 Generating and integrating a debugging interface

Based on the results of SHAna, DIGen generates a debugging interface in the form of a collection of wrapper functions that augment the definitions of kernel exported functions with the necessary checks and calls to logging primitives in order to detect and record violations of safety hole preconditions. Ideally, the kernel maintainer would generate a single debugging interface for the entire kernel that could be used by all service developers. The Linux kernel, however, is highly configurable, targetting a wide range of hardware platforms, and thus many kernel source files have incompatible header file dependencies. Therefore, it is not possible to compile all of the kernel exported functions at the same time. Accordingly, we shift the interface generation process into the hands of

the service developer, who generates an interface specific to his service. Because the functions invoked by a single service can necessarily be compiled together, this approach avoids all compilation difficulties, while producing a debugging interface that is sufficient for and individual service's needs. We now describe the generation of the debugging interface and how it is integrated into a service under development.

Generating a debugging interface For each kernel exported function used in the service 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 5. Based on the argument values, the wrapper function first checks each entry safety-hole precondition (line 4) and then, if the precondition is not satisfied, logs a message indicating the violation of a safety hole precondition. For an entry safety hole, this message includes an indication of whether the violation is *certain* or *possible* (line 5), as defined in Section 3.1. The wrapper then calls the original function. If the original function has a return value, this value is stored in a local variable, `__ret`, and then the preconditions on any exit safety holes are checked based on this information (lines 9-10). Finally, the return value `__ret` of the original function is returned as the result of the wrapper (line 12).

```
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       diagnosis_log("<skind>: <info on the activated safety hole>");
6    /* Invocation of the intended kernel function */
7    __ret = <call to kernel function>;
8    /* Check postconditions for exit safety holes */
9    if <an exit safety-hole precondition is violated>
10      diagnosis_log("[EXIT]: <info on the activated safety hole>");
11   /* Forward the return value */
12   return __ret;
13 }
14 #define <kernel function> __debug_<kernel function>
```

Figure 5: Structure of a wrapper for a non-void function

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

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

3.4 Testing service code with Diagnosys

To be able to use a Diagnosys-generated debugging interface, the service developer must use a version of the Linux kernel in which support for the Diagnosys

runtime system has been installed. This support is expressed as a kernel patch, which we have implemented for Linux 2.6.32, that extends the kernel with a crash resilient logging system. The patch additionally configures the kernel to send all crashes and hangs (Linux soft and hard lockups) to the kernel panic function, which the patch extends to reboot into a crash kernel if Diagnosys is activated or continue with normal panic, otherwise. Finally, the Diagnosys runtime system includes a user-space tool `dlog` for installing a copy of the Diagnosys kernel as a crash kernel, initializing the reserved log buffer, and activating and deactivating the logging functionality.

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

On a kernel crash or hang, the Diagnosys runtime system uses a Kexec-based [26] mechanism to reboot into a new instance of the Diagnosys-enabled kernel. The Kexec-based mechanism performs the reboot without reinitializing any hardware, including the memory, thus ensuring that the accumulated Diagnosys log is still available. The service developer may then access the log messages through a pseudo character device. The messages are made available in the order in which they were generated. Using support available in Linux versions since 2.6.33, the Diagnosys runtime system is also able to insert the standard kernel oops message into the Diagnosys log before rebooting.

3.5 Implementation

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

Diagnosys component	Tool		Code size (LOC)	Language
Kernel code analyzer	SHAna		2438 + 1331	SmPL + OCaml
Wrapper generator	`dmake` + DIGen		40 + 858	sh + OCaml
Logging system	`dlog`	user-space	355	ansi C
		kernel-space	540	ansi C code patch

Table 2: Diagnosys prototype code size

4 Evaluation

In this section, we assess 1) the improvement that Diagnosys provides in debuggability, 2) the coverage of SHAna in terms of the number of safety holes found and their relevance to previously found kernel bugs, and 3) the overhead incurred by the use of Diagnosys from the point of view of the kernel maintainer and the service developer.

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

Several of our experiments involve execution of service code. Because we do not have ready access to services under development, we have taken services that are already integrated into the kernel source tree as examples. We consider 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, as presented in Table 3.

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 3: Tested services

4.1 Evaluating the benefit of Diagnosys

We first compare the process of debugging service code using standard tools with the process of debugging service code using Diagnosys. For this, we replay a crash and a hang reported in the kernel commit logs.

Replaying a kernel crash As an example of a kernel crash, we consider the code from the function *btrfs_init_new_device* previously shown in Fig. 2. The crash occurred because the kernel exported function *open_bdev_exclusive* returns an ERR_PTR value in case of an error, while *btrfs_init_new_device* expects that the value will be NULL. This caused a subsequent pointer dereference to crash. The crash was fixed in Linux 2.6.33.

We have extracted the source code of the btrfs module from just before the application of the patch, and compiled and installed the resulting module. To replay the crash, we must cause the execution of *open_bdev_exclusive* to fail during the execution of *btrfs_init_new_device*. For this, we first create and mount a BTRFS volume and then we attempt to add to this volume a new device that we have not yet created. This operation is handled by the *btrfs_ioctl_add_dev* ioctl which calls *btrfs_init_new_device* with the device

path as an argument. This path value is then passed to *open_bdev_exclusive* which fails to locate the device and returns an ERR_PTR value.

Fig. 6 shows an extract of the oops report that was recovered from the kernel console at the end of the above experiment. Line 1 of this report 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* during a call to *btrfs_ioctl_add_dev* (line 8). Information such as source files and line numbers are not readily apparent, but can be retrieved by applying the standard debugger `gdb` to the compiled module and to the compiled kernel. While the oops report gives adequate information about the point in the code at which the crash occurred, it does not give any information about the previous instructions that led to this crash.

```
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 ]---
```

Figure 6: Oops report following a *btrfs* ERR_PTR pointer dereference crash.

Fig. 7 shows the log generated when using Diagnosys. This log shows that the function *open_bdev_exclusive* returned an ERR_PTR value and thus activated an exit safety hole. Combining this information with the function names in the oops log and the service source code shows that the problem is the inadequacy of the error handling code after *open_bdev_exclusive*. The Linux patch of Fig. 2 fixes the error handling code. The service developer can focus on his own code, and does not have to probe the kernel source or object code to obtain the needed information.

```
1  [EXIT] : 'open_bdev_exclusive' returned an invalid pointer (ERR_PTR)
```

Figure 7: Log line in the execution of *btrfs* with Diagnosys.

Replaying a kernel hang The patch shown in Fig. 8 fixes a hang in the `nouveau_drm` driver. This driver implements a direct rendering manager for nVidia® graphics cards. The affected code consists of a call to the kernel exported function *ttm_bo_wait*. This function exhibits a Lock entry safety hole and a Lock exist safety hole, as it first unlocks and then relocks the *bo* lock. Calling this function without holding this lock causes the kernel to hang.

When we do not use the Diagnosys debugging interface, the hang leaves the developer with little information. Using Diagnosys, the hang causes a kernel panic, which in turn causes a reboot using Kexec that preserves the Diagnosys log. The log (Fig. 9) shows that *ttm_bo_wait* has been called without the expected lock held. Correlating this information with the source code suggests

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

Figure 8: Bug fix related to the usage of *ttm_ bo_ wait*.

taking the lock before the call and releasing it after the calls, as shown in the Linux patch in Fig. 8.

```
[POSSIBLE] : 'ttm_bo_wait' called without spinlock (bo->lock) held
```

Figure 9: Log line in the execution of *nouveau_ drm* with Diagnosys.

4.2 Prevalence and impact of safety holes

The benefit of a Diagnosys-generated debugging interface is determined by the quality of the information collected by SHAna. In this section, we assess the number of safety holes collected by SHAna and the impact these safety holes have had on the robustness of the Linux kernel. Finally, using fault injection, we assess the completeness of the set of safety holes collected by SHAna.

Prevalence of safety holes

Table 4 summarizes for each kind of safety hole the number of 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 reported 22,940 safety holes in 7,505 exported functions. The most frequently occurring kinds of safety holes are *IsNull/Null*, *Lock/Intr/LockIntr* and *Block*. Over 7000 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, and 5% forward the parameter value to other functions that unsafely use them 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 checking its state. The remaining 2% assume that the transmitted mutexes or spinlocks are already held in the calling context and unsafely attempt to release them.

To estimate the utility of the kernel exported functions in new services, we consider 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 exported functions are used, 1 out of 2 calls invokes a function containing a known safety hole. 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.

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
Size	8	-
Range	-	8

Table 4: Prevalence of safety holes in Linux 2.6.32

Impact of the identified safety holes

To assess the impact of the identified safety holes, we have searched through the changelogs of the history of Linux 2.6[1] to identify patches that mention the kernel functions exported in Linux 2.6.32.[2] We have manually reviewed these patches to identify those that are related to the usage of exported functions. As shown in Table 5, 38% (*i.e.*, 267/703) of the usage defects are related to the categories of safety holes that we consider in this paper.

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

Coverage of the identified safety holes

Finally, 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 use fault injection, to trigger actual crashes systematically 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 (those for which the flag information

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

[2] We consider commits whose changelogs mention the name of an exported function, ignoring those in which the function name is used as a common word (*e.g.*, "sort", "panic", etc.).

does not contain __GFP_NOFAIL or __GFP_RETRY) 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 tests in the service code and on failures in the execution of the basic kernel memory allocation functions, potentially deep within the execution of kernel exported functions. For the former, we transform existing service implementations to remove the tests on returned pointer-type values, one by one. For the latter, for each removed test, we use the *failslab* feature of the Linux fault injection infrastructure [6] to cause failures to be injected into the execution of any basic memory allocation functions called within the execution of the code initializing the tested variable.

Results One possible result of a fault injection test 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 6 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 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.

4.3 Overheads

The benefits of Diagnosys come at the cost of some overhead, both in terms of the effort required for the kernel maintainer to certify the safety holes identified by SHAna, and in terms of the runtime cost of the checks and logging operations performed by the debugging interface during service execution. We show that the information maintained by SHAna substantially reduces the effort required from the kernel maintainer over time, while the service execution overhead is minimal.

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 6: Results of fault injection campaigns

Certification overhead

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. Indeed, the Linux kernel provides different definitions of some functions for different architectures, and these different definitions may exhibit different safety holes. At a rate of about 5 minutes per safety hole, this certification requires about a week of work (38 hours). 87% of the safety holes annotated as potential false positives are actual false positives.

Among the remaining reported safety holes, there are 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 certify the safety holes annotated as potential false positives by SHAna.

To further reduce the certification overhead, Diagnosys maintains information about safety holes across OS versions, so that the kernel maintainer need only validate reported safety holes in those functions whose definitions have changed. To demonstrate the potential benefit of this information, we have also certified the SHAna annotated safety holes in 5 versions that were released after Linux 2.6.32. As shown in Figure 10, 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.

Service execution overhead

Testing preconditions and logging incur a performance overhead on the execution of a service. This overhead must be sufficiently small to avoid interfering with the normal service execution.

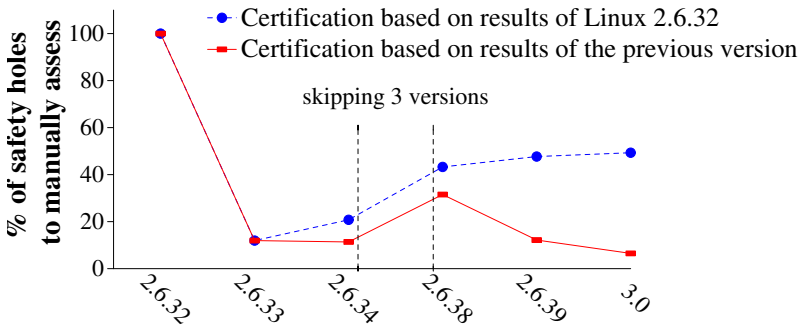


Figure 10: Certification overhead

Execution time of the Diagnosys precondition checks and logging operations To measure the execution time of the Diagnosys precondition checking and logging operations, we have used the Klogger framework [12].[3] We compare the execution time of a call to an exported function with an empty body to that of a call to an exported function containing a single precondition test. Table 7 summarizes the overhead for one instance of a 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 7: Checking overhead ± standard deviation

Table 8 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 [30] 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. Diagnosys' logging primitive is 1.3x faster than Ftrace's *trace_printk*, and 5x faster than *printk*.

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

Table 8: Performance of the Diagnosys logging primitive

[3] Klogger kernel patch for Linux 2.6.31.4

Impact of Diagnosys on service performance To understand the global performance overhead induced by the Diagnosys approach, we execute 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 evaluate the impact of a debugging interface by exercising the e1000e Linux device driver using the TCP_STREAM, UDP_STREAM and UDP_RR tests from the netperf benchmark [17]. For these experiments, the netperf utility was configured to report results accurate to 5% with 99% confidence. Table 9 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 0.4% to 6.4%, and increases the CPU utilization by 0.4% to 10%. Nevertheless, while small, the existence of this overhead suggests why kernel developers would not want to systematically implement exported 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 9: 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 [3]. 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 10, 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%.

5 Related work

In the last decade, studies have shown that kernel-level services, in particular device drivers, are responsible for the majority of OS crashes. Ganapathi *et al.* have found that 65% of all Windows XP crashes are due to device drivers [14]. Ten years ago, Chou *et al.* found that the error rate in Linux drivers is 3–7 times higher than that of other parts of the kernel [5]. Palix *et al.* have shown that while this error rate is decreasing, Linux drivers still contain many defects [27].

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 10: Performance of the NFS file system

They have also found that file systems have recently had a high fault rate, indeed even higher than that of drivers.

System robustness testing Fault injection has been applied to the Linux kernel and to application software to evaluate the impact of various fault classes [1, 7, 24]. In our work, we have pinpointed the safety holes in kernel interfaces that explain their observations, thus providing developers with information on how to avoid such situations. Marinescu and Candea [24] propose a fault injection framework that focuses on the returns of error codes from userspace library functions. These amount to activations of our Null exit safety holes. No support, however, is provided for identifying the impact of other types of safety holes that may be activated during the execution following these injected faults.

Static bug finding Model checking, theorem proving, program analysis and combinations thereof have been used to statically analyze OS code to find thousands of bugs [2, 9, 10, 15, 20, 28]. Nevertheless, these tools take some time to run and the results take some time to interpret. Thus, these tools are not well suited to the frequent modifications and tests that are typical of initial code development. A number of approaches have proposed to statically infer so-called *protocols*, describing expected sequences of function calls [11, 20, 21, 23, 29]. 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. Little attention has also been paid to protocols that involve only one function call such as protocols on the use of NULL pointers.

Some of our kinds of safety holes could be eliminated by the use of a more advanced type system. For example, Bugrara and Aiken have proposed an analysis that differentiates between safe and unsafe userspace pointers in kernel code [4]. Nevertheless, this work focuses on the kernel as a whole, and not the interface between the kernel and a new service under development, thus potentially informing the service developer about faults in code that he is not concerned with.

Run-time fault tolerance Systems tolerating run-time faults aim at improving system reliability by protecting the kernel from faults triggered by driver code. Nooks [33], SafeDrive [37], TwinDrivers [25] and DD/OS [22] use, respectively, hardware protection, language-based techniques or virtual machines.

However, as the designers of these have admitted, these systems are not widely used, perhaps because of their heavyweight nature [18].

Logging systems Runtime logs usually offer the starting point for diagnosing software failures. Nonetheless, these logs are frequently insufficient for failure diagnosis especially in case of unexpected crashes [8]. The interest of our approach is then illustrated by *LogEnhancer* [36], which enriches log messages with extra information to improve diagnosis. Nevertheless, *LogEnhancer* does not create new messages. Diagnosys creates new log messages along the boundary between a service and the rest of the kernel where they can be most helpful to the developers of new kernel services.

Robust interfaces Because they provide a direct entry to the core kernel as well as to other kernel modules, the use of kernel exported functions makes the kernel highly vulnerable to faulty or malicious kernel modules. Mao *et al.* have designed LXFI [35] to isolate kernel modules. They propose 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 bug fixes based on different categories of common faults encountered in kernel code.

Healers is an automated approach to generating a robust interface to a user level library without access to the source code [13]. Healers relies on fault injection to identify the set of assumptions that a library function makes about its arguments. Because fault injection relies on test executions of the library, Healers can obtain some information about runtime values, such as array bounds, that may be difficult to detect using static analysis. On the other hand, information about the calling context, such as the set of locks held, is expensive to obtain at runtime, because it would require testing the state of all available locks. Indeed, Healers does not address safety hole kinds such as Lock, which can easily be detected via static analysis of the library function code.

6 Conclusion

Current commodity OSes are monolithic, implying that kernel-level services, run closely integrated with the kernel. When these services present defects, they can cause the demise of the entire system, often leaving developers without any clue as to what went wrong. In this paper, we have focused on easing the initial development, when service programmers are not necessarily aware of the usage preconditions of kernel interfaces.

We have designed Diagnosys, a tool to investigate the Linux kernel so as to detect and classify all kernel exported functions that present safety holes. To support the developer of a new service in the testing of his code, Diagnosys automatically generates a debugging interface to these functions, and provides a logging system that is resilient to crashes and hangs for recording information about uses of these functions.

Using various drivers and file systems that we have injected with faults, we have shown that our interface gives useful feedback by alerting the developer to the critical defects in his code. Using a driver for a Gigabit Ethernet device and

a NFS file system, we have shown that the performance impact of our approach is within the limits of what is acceptable when testing a kernel-level service in the initial stages of development.

Appendix

Locking functions: *{mutex,spin,read,write}_lock*
{mutex,spin,read,write}_trylock
Interrupt management functions: *cli, local_irq_disable*
Functions combining locking and interrupt management: *{read,write,spin}_lock_irq*
{read,write,spin}_lock_irqsave, local_irq_save, save_and_cli
Kernel/userland access primitives: *getuser, memcpy_fromfs, copy_from-*
_user

Availability

The source code of Diagnosys as well as the experimental materials for this paper can be found at `http://www.labri.fr/perso/bissyand/diagnosys/`

References

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

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

[3] T. Bray. The Bonnie file system benchmark. `http://www.textuality.com/bonnie/`.

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

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

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

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

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

[9] D. Engler and K. Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *SOSP '03*, pages 237–252, Bolton Landing, NY, USA.

[10] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI'00*, pages 1–16, San Diego, CA, USA.

[11] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. In *SOSP '01*, pages 57–72, Banff, Alberta, Canada.

[12] Y. Etsion, D. Tsafrir, S. Kirkpatrick, and D. G. Feitelson. Fine grained kernel logging with klogger: experience and insights. In *EuroSys '07*, pages 259–272, Lisbon, Portugal.

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

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

[15] P. J. Guo and D. Engler. Linux kernel developer responses to static analysis bug reports. In *USENIX'09*, pages 285–292, San Diego, CA, USA, 2009.

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

[17] R. Jones. Netperf: A network performance benchmark, version 2.4.5. `http://www.netperf.org`.

[18] A. Kadav, M. J. Renzelmann, and M. M. Swift. Tolerating hardware device failures in software. In *SOSP'09*, pages 59–72, Big Sky, MT, USA.

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

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

[21] C. Le Goues and W. Weimer. Specification mining with few false positives. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 5505 of *Lecture Notes in Computer Science*, pages 292–306, York, UK, Mar. 2009.

[22] J. LeVasseur, V. Uhlig, J. Stoess, and S. Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *OSDI'04*, pages 17–30, San Francisco, CA, USA.

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

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

[25] A. Menon, S. Schubert, and W. Zwaenepoel. Twindrivers: semi-automatic derivation of fast and safe hypervisor network drivers from guest OS drivers. In *ASPLOS'09*, pages 301–312, Washington, DC, USA.

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

[27] N. Palix, S. Saha, G. Thomas, C. Calvès, J. L. Lawall, and G. Muller. Faults in Linux: Ten years later. In *ASPLOS'11*, Newport Beach, CA, USA.

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

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

[30] S. Rostedt. Debugging the kernel using ftrace. `http://lwn.net/Articles/365835/`, December 2009.

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

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

[33] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. In *SOSP'03*, pages 207–222, Bolton Landing, NY, USA.

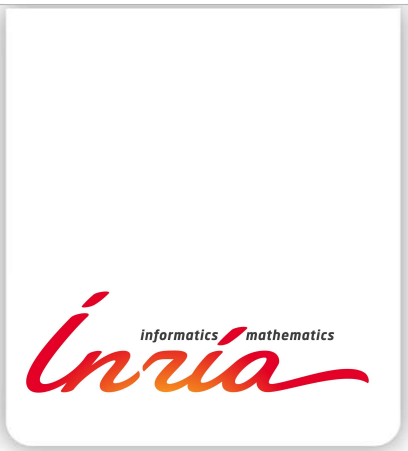
[34] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan. Detecting large-scale system problems by mining console logs. In *SOSP '09*, pages 117–132, Big Sky, MT, USA.

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

[36] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage. Improving software diagnosability via log enhancement. In *ASPLOS '11*, pages 3–14, Newport Beach, CA, USA, 2011.

[37] F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harren, G. Necula, and E. Brewer. SafeDrive: Safe and recoverable extensions using language-based techniques. In *OSDI'06*, pages 45–60, Seattle, WA, USA.

Contents



RESEARCH CENTRE
PARIS – ROCQUENCOURT

Domaine de Voluceau, - Rocquencourt
B.P. 105 - 78153 Le Chesnay Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399