Improving Dependability of Commodity Operating Systems with Program Analysis

Feng Zhou

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2007-121
http://www.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-121.html

October 8, 2007

Improving Dependability of Commodity Operating Systems with Program Analysis

by

Feng Zhou

B.S. (Tsinghua University) 2000
M.S. (University of California, Berkeley) 2005

A dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:

Professor Eric A. Brewer, Chair
Professor George C. Necula
Professor Paul K. Wright

Fall 2007

The dissertation of Feng Zhou is approved:

Professor Eric A. Brewer, Chair	Date
Professor George C. Necula	Date
Professor Paul K. Wright	Date

University of California, Berkeley

Fall 2007

Improving Dependability of Commodity Operating Systems
with Program Analysis

Abstract

Improving Dependability of Commodity Operating Systems
with Program Analysis

by

Feng Zhou

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Eric A. Brewer, Chair

Modern operating systems are notoriously complex and hard to make dependable. Due to performance, flexibility and historical reasons, most of them are written in relatively low level languages like C and C++. These languages lack safe type systems and provide few guarantees of safety and reliability. We propose to improve the dependability of these systems with more sophisticated program analysis using tools that understands the tricky issues that human beings often mistake about, for example, locking and memory safety. The main challenge we see here is the complexity and scale of the systems, which makes fully automated verification hard. Therefore we propose that these analyses require some help from the developers, in the form of non-intrusive annotations in the code. Moreover, some analyses can be made *hybrid*, consists of both static and dynamic (runtime) checks, thus making the analysis much simpler without losing precision. We argue that we should still strive for soundness despite the scale of the problems, as only sound analyses can provide us with guarantees that the system is in absence of certain categories of bugs.

We present two case studies of applying these techniques to the Linux kernel. Both of them are efficient enough to be used on the whole kernel itself, require at

most a few percent of code changes, and find real bugs in the kernel.

First, we present a static analysis tool that analyzes the interaction between processor execution contexts and locks in the Linux kernel, and tries to finds bugs related to these aspects. *Execution contexts* are certain state of the processor, which can decide at this particular time what operations the kernel is allowed to do. For example, the kernel is not allowed to do a task switch while serving an interrupt (in interrupt context). We analyze the "process context" and "hardware interrupt context", and the usages of "spin lock" primitives in these contexts. The analysis is a flow-sensitive, inter-procedural and context-insensitive analysis. The current prototype analyzes a minimally-configured Linux 2.6.20 kernel (over 850K lines of C code) in less than 2 minutes. It found 6 confirmed bugs in the kernel.

Then, we present a system for detecting and recovering from type safety violations in Linux device drivers. It uses a novel type system, partly specified with annotations, that provides fine-grained isolation for existing Linux device drivers. In addition, we track invariants using simple wrappers for the host system API and restore them when recovering from a violation. This approach achieves fine-grained memory error detection and recovery with few code changes and at a significantly lower performance cost than existing solutions based on hardware-enforced domains, such as Nooks [Swift *et al.*, 2003], L4 [LeVasseur *et al.*, 2004], and Xen [Fraser *et al.*, 2004], or software-enforced domains, such as SFI [Wahbe *et al.*, 1993].

The principles of the analyses and tools presented are general. These aspects such as execution contexts, or device driver interfaces are similar among different operating systems. So we believe these techniques can be adapted to other operating systems as well.

Acknowledgements

I would like to thank my advisor Eric A. Brewer. Eric is always a great source of new ideas, insights, enthusiasm and optimism. Many good ideas arose (or bad ones killed) during discussion in his office. Also I benefited tremendously from the encouragements I received from him to pursue some seemingly vague ideas, or simply to keep improving writing or presentation.

I also want to thank Jeremy Condit, George Necula, Matt Harren, Zachery Anderson and David Gay for insightful discussions during the Ivy group meetings, and for allowing me to use their Deputy compiler in the SafeDrive project. Their continuous improvements and quick bug fixes to Deputy helped SafeDrive enormously. I thank Bill McCloskey for lots of discussion and help with programming language techniques that I am not familiar with. I would like to thank Ben Zhao a lot for his help on Cashmere and other projects, and help with how to do research in general. I also learned a lot from discussions with Rob Ennals, Ilya Bagrak and Rob von Behren.

Finally, I would like to thank the following people for lots of enlightening non-technical discussions and help while I am a graduate student: Yitao Duan, Wei Xu, Ling Huang, Qi Zhu, Yang Yang, Hao Chen, Hao Zhang, Jingtao Wang, Yaping Li, Bowei Du, Michael Demmer, Michael Rosenblum, Rabin Patra, Sergiu Nedevschi, Russel Sears.

Dedicated to

My wife Li, for years of love and support

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Operating systems, and systems software in general, have become ever more important in the increasingly software-powered society we live in. Computers, cars, home appliances and Mars Rovers are all driven by operating systems in one or another form. However, making reliable and dependable operating systems have always been a challenge. And we think it is going to be more challenging with more demanding applications and current hardware and architecture trends. Two key trends, *complexity* and *concurrency*, make the traditional approach of testing, manual or automatic, harder to be effective for future operating system development. We argue that it is both productive and necessary to resort to sophisticated automated analysis of whole operating systems, to find problem early and seek assurance that the system is in absence of certain classes of defects.

We present two case studies, tackling concurrency and type safety problems respectively for the Linux kernel. The first is a static analysis that analyzes execution contexts and locks in the whole Linux kernel. The second is a system utilizing both static and runtime techniques for detecting and recovering from type safety violations in Linux device drivers.

1.1 The Operating System Dependability Challenge

It is quite obvious that commodity operating systems today are not as reliable and dependable as we wish they are. For home and office use, they generally work well, but still fail now and then, especially when dealing with sophisticated or peculiar hardware devices, like a latest video card or some rare digital camera. Worms and viruses are a big and long-lasting problem because there are many hidden bugs in production OSes that are waiting to be exploited. Reliability is especially lacking for long running servers and large server clusters, because hidden bugs are more likely to manifest themselves when servers are run for a long time, or thousands of them are running [Bligh *et al.*, 2007].

The most obvious reason for these problems is the sheer size and complexity of operating systems. Operating systems are giant and complex pieces of software. According to [Anonymous, 2007], Windows Vista has about 50 million source lines of code, all the way from 6 million lines of Windows NT 3.1 in 1993. Even if we do not count the user level code, operating system kernels themselves are still giant. The latest Linux kernel, 2.6.21, contains 8.25 million lines in 21,615 files (including a small percentage of documentation) [Kroah-Hartman, 2007]. This is only more impressive if you consider the rate of change, which is necessitated by the advent of new applications, new hardware and the evolution of the machine architecture. For Linux kernel, the current size of 8.25 million lines is up from 6.62 million of 2.6.11 only 2 1/3 years ago, which “comes to a crazy 85.63 new lines of code being added to the kernel tree every hour” [Kroah-Hartman, 2007].

Another reason for problems in operating systems is the language used. A lot of operating systems bugs can be attributed to the lack of type safety, and other deficiencies of C and C++, used by most commodity operating systems. For example, the most common source of security problem, buffer overruns, would be completely removed if the system is written in a type-safe language.

We expect a couple of recent trends in hardware and architecture to further worsen the situation and make the challenge to write dependable operating systems even bigger.

- *Effectively utilizing parallel processors has become a necessity.* Recently most micro processor vendors have moved from single-core to multi-core processors. The reason is that physical limits (mainly thermal) have stopped the exponential increase of single-thread processor performance. And processor manufacturers have resort to placing many processor cores on a single chip instead as a means to further increase performance. For example, today there is already processors with 8 core and 32 hardware threads on the market (Sun UltraSparc T1). And Intel is reportedly planning a server processor with 32 cores to be released in 2010 [Schmid, 2006].

  This transition has significant ramifications for all software [Asanovic *et al.*, 2006]. Some researchers described the results as *"the free lunch is over"* [Sutter, 2005]. Most software systems now have to take concurrency into account, including those that have historically relied on Moore's Law for adding more functionality while maintaining end-user performance. Therefore for all general-purpose operating systems, supporting SMP has become an indispensable feature. And to get good performance, the operating system need to be more sophisticated so that it can take advantage of an ever-growing number of cores, and in addition, new architectural properties like Cache-Coherence Non-Uniform Memory Access (ccNUMA), and a deep cache hierarchy inside the processor. This translates to more fined-grained locking, a more complicated memory sub-system and etc, unfortunately all complex and error-prone mechanisms.

- *Devices are becoming ever more complex.* Nowadays, it is common to have a single device driver with a few hundred thousand lines of code, because of the multitude of features and smartness the device supports. Examples are modern video cards with sophisticated 3D acceleration support, and network cards with TCP off-loading and other complicated features. The contemporary power-management specification, ACPI, even includes a byte-code language that needs to be interpreted by the operating system [Compaq *et al.*, 2001]. Therefore complexity of code handling devices are growing quickly.

- *The cost of defects is increasing.* Computers and devices are becoming more and more networked. Not only servers, but also a lot of client computers and handle-held devices are in an always-online state. Therefore security is big concern. A single remote exploitable bug in a commodity operating system running on millions of networked computers has the potential of costing billions of dollars. Moreover, commodity operating systems have gradually been pick up by a lot of "critical infrastructure" systems like management systems of power grids, probably because it is simply easier and cheaper to build the system on commodity operating systems. In recent years, there have been many examples of worms and operating system bugs that brought down critical systems, for example causing tens of thousands of ATM machines to fail simultaneously, or shutting down the 911 emergency phone service of an entire city [Schneier, 2003]. NIST calculated that in 2002 alone, software errors cost the United States $59.5 billion dollars [NIST, 2002].

All these trends show that it is high time that we try to solve the dependability challenge for commodity operating systems.

1.2 Other Approaches

The traditional way of ensuring software quality is through testing. This is no different for operating systems. Software test has a long history and is a mature art. Testing can be done either manually or automatically. It can be done at different levels, from testing units to testing the whole system. It can also be done for different purposes, for example, functional tests for correctness, and stress test for behavior under load.

However, we believe a couple of reasons have made testing operating systems hard, and may make them harder in the future.

- *Complexity.* As discussed in the last section, more hardware devices and architectural features to support and etc, are all making the operating systems more complex. Achieving good test coverage for very large software have been prove hard. And there is the added problem that the OS has to be tested on particular hardware to test the specific device or hardware support. So comprehensively testing all aspects of an OS in a controlled and reproducible manner is in itself a big challenge.

- *Concurrency.* Concurrency related bugs are hard to test. Race conditions, deadlocks or interrupt related defects are often not easy to reproduce. And again concurrency bugs in device drivers may depend on specific hardware, and thus hard to test comprehensively.

- *Corner cases.* Security problems like buffer overruns are often corner cases. And testing often has a hard time revealing them because they happen only with specific combination of input. And even when they happen, may not manifest themselves immediately.

Another approach that promises to solve most problems is to rewrite operating systems in a high-level, type-safe language. Efforts in this direction include Cedar [Swinehart *et al.*, 1986], SPIN [Bershad *et al.*, 1995] (in Modula-3) and Singularity [Hunt and Larus, 2007] (in Sing#, an extension of C#). This solves the type-safety problem, and has a lot of promises in making the system more analyzable, which could eventually lead to much more dependable systems. And a lot of progress has recently been made [Hunt and Larus, 2007]. On the other hand, there are plenty of challenges. To name a few, it is obviously painful to rewriting everything, in particular device drivers for tens of thousands of hardware devices. User applications may need to be modified or rewritten too. And because the mainstream type-safe languages are all virtual-machine based and garbage-collected, performance and latency are concerns. Last, note that type-safety does not solve all our problems. For example they probably do not help much with interrupt/concurrency related bugs.

1.3 Towards More Dependable Operating System Kernels

In this study, we focus our work on using programming language techniques to analyze existing commodity operating systems and improve their dependability. Specifically we work with Linux. And apart from a very small number of annotations and changes, we do not modify the OS itself much. We create new static analysis tools and runtime tools to verify certain properties, find potential problems, detect failure at runtime and recover from them.

Concretely, in Chapter 2 we discuss a static analysis that infers and checks execution contexts and locking primitive and other relevant kernel primitives, and tries to find bugs statically. And in Chapter 3 we describe a system that provides fine-grained type safety and recoverability for Linux device drivers without using a new language. We conduct these two case studies to validate the hypothesis that program analysis, combined with runtime techniques, can help improve operating systems dependability greatly.

Software verification, referred to by some as the Holy Grail of computer science, is a well-studied field. But the state of art is still far from perfection. The goal of this work, therefore, is to ask the following question, *what combination of automatic static analysis, runtime instrumentation and changes to OS architecture and coding practices can provide us with a practical, assuring and general paradigm for dramatically improving the dependability of commodity operating systems.* To elaborate a bit more,

- We want the techniques to be *practical* today. That is, it should help find useful problems without too many false alarms, should not require a lot of extra effort to use, and should scale to the complexity and size of modern operating systems.
- We favor techniques that can provide *assurance* that certain kinds of bugs do not exist in the system. Formally these are called *sound* verifications or analyses. They are in a different class from "bug-finding" tools that uses heuristics to find possible bugs, which inevitably misses some bugs in their domain and cannot provide assurance.
- We favor techniques that are *general* for many operating systems. The tool implementation may be tailored towards a specific OS, but the techniques should be general enough and targeting common practices and patterns across OSes, and it should be easy to adapt them to other systems.

We also have a couple of non-goals. First, the need for sound analysis is a quest rather than a requirement. Practicality takes precedence when ensuring soundness becomes too costly. Similarly, favoring generality does not mean we never allow ad-hoc techniques. When an ad-hoc technique improves results or performance a lot, and it is much needed, we will take it. Finally, the programming language techniques we use are not new. It is the combination and actual application and experience on large-scale systems that is novel.

Challenges are abound for achieving these goals. The biggest one is how to balance analysis scalability and precision. As it is necessary for us to analyze systems with millions of lines of code, we risk losing so much precision that the tools are no longer useful. Another challenge is the quirks of the C language. Apart from not having type-safety, the language also have many features, like varargs or inline assembly, that are hard or impossible to analyze.

One design choice we pick against these challenges, is to require light-weight annotations in both of our case studies. Obviously the benefit is that, if designed properly, annotations can help the analysis understand the code better at key places, and increase both scalability and precision. The downside is also obvious. Annotations require expensive human effort and, maybe more seriously, may contain errors that misguide the analysis. To reduce the human effort needed, each analysis does inference on the annotations if possible. Therefore, we keep the requirement for manual annotations to a very low level, normally changing less than 1% of code lines. To counter wrong annotations, most of them are *not trusted* by the analyses. We either check them statically and report possible inconsistencies, or instrument the code to dynamically check the assertions/annotations at runtime. Therefore, we try hard to find any wrong annotations that may lead the analysis awry.

In practice, we have found that adding small number of annotations not a big burden. In fact, the Linux kernel community has been doing this to the mainline kernel for quite a while, for the consumption of their own heuristics-based static analysis tool [Torvalds and Triplett, 2003]. The general feeling is that for complicated code like the kernel, properly designed annotations make good documentation, express meta-data not expressible with the language itself, and thus over time become a useful part of the code even for human readers. We even conjecture that one use of the inference capability of our tools would be to generate kernel documentation with inferred annotations, because a lot of these are not obvious to developers not too familiar with the code and are helpful to their understanding of the code.

Another technique that we use to tackle the challenges is *hybrid checking*, which refers to sound analyses where certain properties are verified at runtime instead of compile time. This is used in recent schemes like Hybrid Type Checking [Flanagan, 2006] and CCured [Necula *et al.*, 2005]. Basically the analysis works together with the compiler. It tries its best to verify the relevant properties. It passes if it is definitely okay, or reports an error if it thinks there is a defect. However, in the middle is when the static analysis cannot decide. At this time, instead of failing or forfeiting soundness by ignoring it, it directs the compiler to generate necessary runtime checks to instrument the program to report an error whenever an actual problem occurs at runtime. The benefit of this approach is that for hard-to-check properties, it gives the analysis another option in addition to poor scalability, lots of false positives, or loss of soundness. For cases where runtime checking is cheap, this is in general a very good option. Compared to full dynamic checking tools, hybrid checking often has far better runtime performance, because it uses the static analysis as a performance optimization, removing lots of checks that can be proved to be unnecessary.

Chapter 2

Execution Contexts and Locks

2.1 Introduction

Concurrency has recently gained a lot of attention in the operating system community. The movement to multicore processors have made SMP support a requirement for any modern operating system. And concurrency control, in the general sense of ensuring correct functioning in the face of asynchronous events and parallel execution paths, is essential to the system when running on multiple processors or cores. Moreover, in order to utilize an increasing number of cores, more fine-grained concurrency control mechanisms are needed, making the part of the system dealing with concurrency more complex, and unfortunately, more error-prone.

A lot of work on concurrency focused on the issue of ensuring synchronized accesses to shared state, using locks or other forms of synchronization. We observe that concurrency control in the kernel generally involves more issues than that. In addition to concurrent execution of code, the kernel also needs to deal with external events that could happen at any time, in the form of exceptions and interrupts. Here *exceptions* refers to events that happen within the processor that needs the processor's immediate attention, for example, page faults or divide-by-zero errors. This is as opposed to *interrupts*, which originate from hardware devices outside the processor. Most modern operating systems, including Linux, allow interleaving of kernel code execution for fast response to interrupts and exceptions. This gives rise to much indeterminism, and thus a series of restrictions on what the kernel code is allowed to do in order to function correctly given all possible interleavings.

In this chapter, we present a tool, called Ctxcheck, that uses program analysis to statically understand two specific concurrency-related issues and their interaction, and try to check the kernel code statically against a known set of possible bugs. The goal is that, if the analysis passes, the code should be free of bugs of these kinds, regardless of any runtime situation, like different orderings and timings of interrupts and exceptions. We introduce these two issues very briefly here and will discuss the background in more details in the next section. One issue we study is *execution context*, or *context* for short, which we use to refer to: 1) whether the processor is in an *interrupt service routine*, 2) whether interrupts are disabled (masked) or enabled. Contexts poses restrictions on the operations allowed. For example, trying to force the current process to sleep in an interrupt handler is almost always a bug. The other issue is *spin locks* safety. Spin locks are used to serialize access to shared data from multiple processors. There are restrictions, sometimes subtle, on how and when these locks should be used in different contexts. Incorrect use will often lead to hard-to-reproduce deadlocks. As we will show, the kinds of bugs related to these two issues and their interaction are often the tricky ones that are not obvious from the code, and often hard to reproduce. Therefore static checking of these bugs should be useful.

We discuss necessary background of these issues in the next section, present the analysis of contexts in Section 2.3, and the checks we do with context and spin locks in Section 2.4. We present evaluation results in Section 2.5, discuss related work in Section 2.7 and concludes this chapter in Section 2.8.

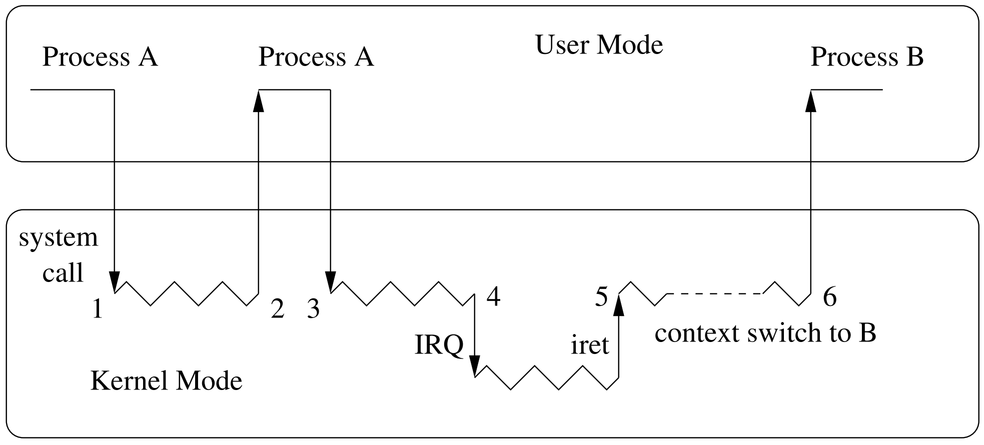


Figure 2.1: Example Linux kernel control path for interrupt handling

2.2 Background

As background to our study, in this section we go over the relevant mechanisms Linux uses for interrupt handling and synchronization.

First let us discuss *execution contexts*. Figure 2.1 shows a typical control path of Linux when handling system calls and interrupts.

- During time 1 to 2, and during 3 to 4, the kernel is processing systems calls. This state is referred to as in *process context*, or specifically in *the context of process A*.

- During 4 to 5, the kernel is handling a hardware interrupt. This is referred to as in *interrupt context*. When an interrupt happens, the kernel handles it immediately, unless interrupt is explicitly disabled (masked) by the kernel. The *interrupt service routine*, the code handling the interrupt, uses the same kernel-mode execution stack as code handling system calls.

Interrupt, exception and parallel code execution all bring concurrency to the kernel. Therefore accesses to shared data structures need to be synchronized. The kernel provides a plethora of primitives for synchronization. They include those normally available to applications, like mutexes and semaphore, and also some unique to the kernel, like spin locks and per-CPU variables. As discussed, we focus on spin locks in this study. Spin locks are only for SMP kernels. Their sole purpose is to prevent unsynchronized accesses by multiple processors to the same shared data. The name comes from the fact that when one processor cannot acquire a lock, it busy-waits, or *spins*, until it acquires it. This is useful for two main reasons. First, unlike mutexes or semaphores, spin locks can be used in interrupts. Second, in a lot of cases, busy-waiting for a few dozens cycles or more, is actually much more efficient than making the current process sleep, which often requires a few thousand cycles, as mutexes and semaphores do when the lock cannot be acquired.

The following are the main spin lock primitives.

1. `spin_lock(lock)`, `spin_unlock(lock)`. These acquire and release a spin lock.

2. `spin_lock_irq(lock)`, `spin_unlock_irq(lock)`. `spin_lock_irq()` disables interrupts on the local CPU, and then acquires the lock. `spin_unlock_irq()` releases the lock and re-enables local interrupts. These are used, in process context, to protect some data that are also used in interrupts. It makes sure that while we are holding the lock, the local CPU will not handle an interrupt that may in turn needs the spin lock, leading to a deadlock. Note that it only disables interrupts on the local CPU, other CPUs may still process interrupts and compete for the lock, which is okay.

3. `spin_lock_irqsave(lock,flags)`, `spin_unlock_irqrestore(lock,flags)`. These are similar to the last pair, and in addition saves and restores the interrupt mask flags. They are useful in places where it is unknown whether interrupts are enabled or disabled.

For more detailed discussion of Linux interrupt handling and synchronization, interested readers are referred to Ch. 4 and Ch. 5 of [Bovet and Cesati, 2005].

2.3 Analyzing Execution Contexts

As we have seen in Section 2.1, *execution contexts* play a big part in what concurrency-related operations the kernel is allowed to do and their consequences. Unfortunately the possible contexts a piece of code could execute in is often not obvious by looking at it. So the first part of our study is to build a static analysis that tries to figure out the possible values of relevant context flags for any code location in the code.

2.3.1 Analysis Overview

There are many context flags in the kernel. Some of them are stored in CPU registers. Others are stored in the task structure (`current`), along with other state of the kernel task executing on the current processor. Our analysis is concerned with the following two flags,

1. `in_irq`. This indicates whether the CPU is handling a hardware interrupt. At runtime, this can be tested with `in_irq()`, which reads a bit from the task struct. For a particular function, this flag does not change.
2. `irqs_enabled`. This indicates whether the current CPU has interrupts enabled. For x86, this is the IF bit in the EFLAGS processor register. Unlike `in_irq`, this flag can be changed at any time by the program in many ways, for example by calling `local_irq_disable()`..

Our analysis is flow-sensitive, inter-procedural, context-insensitive and whole-program. Through intra-procedural and inter-procedural propagation of information known about the code, it basically assigns the possible combination of execution context flags to each location in the code. With this information, we can then proceed (in Section 2.4) to find any inconsistent use of execution contexts, or incorrect use of spin lock primitives.

Note that it is quite common for a function to be used in several different execution contexts. In order to provide enough resolution for these cases, and because our analysis is *context-insensitive*, we actually assign *a set of* possible flag combinations for each location. Also, each flag is tri-state, that is, it can be either *true*, *false* or *unknown*, in order to be able to process under-specified code.

The analysis is summary based. It computes a summary for each function consisting of the following fields,

1. `flags`. A set of `(in_irq,irqs_enabled)` tuples that are possible at the entrance to the function.

2. `must_in_irq`. Whether we know for sure the function needs to be called in ISR, not in ISR or unknown. It could be *true*, *false* or *unknown* respectively. Intuitively, this can be inferred if the function always calls some function that must or must not be used in ISRs.

3. `irqs_must_enabled`. Similarly, whether we know for sure the function needs to be called with IRQs enabled, disabled or unknown.

4. `action`. Whether the function enables the interrupts, disables or keeps it unchanged.

2.3.2 Constructing the Call Graph

We work on a single source file version of the Linux kernel, merged using the merger in the CIL compiler framework [Necula *et al.*, 2002]. The first step of the analysis is to construct a context-insensitive call graph of the whole kernel. This identifies which functions can be called from a function, and vice versa. This is done with a single pass through the file by looking for function calls.

Function pointers need special treatment. We use a custom points-to analysis for function pointers to understand which functions one function pointer could potentially point to. In some tricky places, in order to achieve the scalability we need, we resort to requiring some annotations from the user to help the analysis to go through.

We use a very simple type-based points-to analysis as the baseline. Basically if we do not know anything about the function pointer, we assume that it could point to any function in the kernel that has had its address taken and is type-compatible with the pointer. Type compatibility is very loosely defined. We currently only require that they have the same number of arguments. Stricter rules can be used, but one needs to be quite careful not be become unsound as argument types of function pointers are used quite liberally. It is easy to see that our setting is sound assuming the code obeys the C calling convention and does not process arguments by manually working with the stack. We believe this is true for the part of kernel we care about. Obviously this naive analysis gives a lot of false call paths. But as we will see below, this baseline analysis is seldom used.

To improve precision, we observe that aliasing of function pointers are often much more restricted compared to aliasing of other pointers. For starter there is no dynamic allocation of targets (functions) involved. And for the kernel, there are a small number of patterns or idioms that covers most cases. So we inspect the code and add support for some of the idioms that the kernel code uses and provide much more precise results for these cases. Here are the main idioms that we handle.

- A very common idiom is C++ *vtable*-like structures that stores a set of functions implementing a particular abstract interface. For example the `inode_operations` struct contains function pointers implementing the *inode* interface for any file system. In most cases, enough precision can be achieve by associating a set of functions with each field of the structure type. Therefore we keep track of assignments to function pointer struct fields and only output those functions that are reachable from those assigned to the particular field. This is sound as long as no physical polymorphism of structs is involved, which we did not find any in the kernel.

- Similarly for static function pointers arrays. We track assignments to members of function pointer arrays, and unify the set of possible functions against a particular static/global array.

- The above simple techniques work for most cases. However, we found that generic call-back structures require more resolution. For example, there are a number of global *notifier chains*. Each corresponds to different a set of registered callbacks, but all of them are stored in the `notifier_block` struct. The above algorithm will collapse all callbacks for all notifier chains together, which results in vast number of false paths. Solving this generally will probably require context-sensitivity in the points-to analysis. We decided not to go that way and work around this in a semi-adhoc way by requiring an annotation that links the call site to the functions registering the callbacks. And because each chain has a separate callback-registering function, this solves the problem. The particular annotations (`CALL1/CALL2/CALL3`) are discussed in Section 2.4.

The analysis is implemented in an "on-demand" way. During analysis, it visits all definitions and functions to create a graph representing the assignment relationship between various entities in the program. Entities include local and global function pointer variables, struct or union fields of function pointer types, function pointer arrays, and functions themselves, etc. Then when asked what functions a function pointer can point to, the analysis traverses the graph and returns all reachable functions as the answer, so that transitively assigned functions can be discovered.

2.3.3 Inference of `must_in_irq`

After the call graph is constructed, the next step is to use a simple backwards dataflow analysis (ch.9.2 of [Aho *et al.*, 2007]) to infer `must_in_irq` flags for functions, and use the call graph to propagate it throughout the program. Having these flags help the analysis later-on. And by inferring instead of relying on the programmer to annotate all such functions, we can save much effort by the programmer.

Initially all functions have `must_in_irq=unknown` except those manually annotated as `true` or `false`. For example, `mutex_lock()` is annotated to have `must_in_irq=false` because mutex acquisition cannot be done in interrupt context. Then we apply the following backwards dataflow analysis to each function.

- The flow state is simply a tri-state value `must_in_irq`, initialized to *unknown*.
- The transfer function is the identity function, unless the current instruction is a call to a function that has `must_in_irq=true`, or `must_in_irq=false`, in which case the flow state becomes *true* or *false*, respectively.
- The join function returns `true` or `false` if both inputs are `true` or both are `false`. Otherwise, it returns `unknown`. It is used to join the state from multiple basic blocks, and also from the calling of multiple functions from the same function pointer.

Standard work-list algorithm is used to propagate this to all functions. The call graph is consulted each time the dataflow analysis results in the change of the `must_in_irq` bit of the current function. When this happens, all potential callers of the current function are added to the work-list again so that they can be re-evaluated.

The `irqs_must_enabled` flag is currently not inferred. Its inference is harder than that of `must_in_irq` because `irqs_enabled` could be changed at any point in the code, in particular by function calls. Therefore effective inference could only be done when the `action` flag of each function is known, which is actually part of the output of our whole analysis. Therefore the inference would be more involved. So for now we rely on the programmer to provide `irqs_must_enabled` annotations when they are needed.

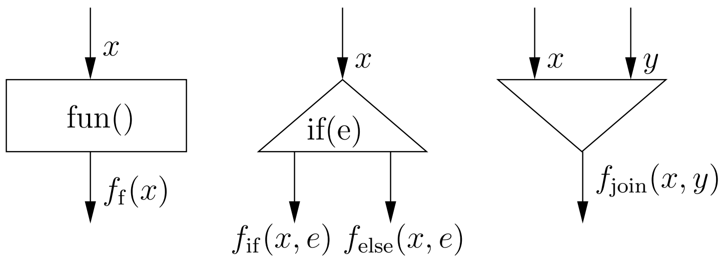
2.3.4 Dataflow Analysis

The core of our analysis is a dataflow analysis that works on one function a time. It takes as input the preconditions of the function (`flags`), and summaries of any other functions that it may call. It produces as output, this function's `action`, and any changes to preconditions of other functions. During the analysis process, we compute on-the-fly the set of possible flags for each location in the code, and these are used to do safety checks that we will discuss in Section 2.4.

The dataflow analysis is a standard forward dataflow one (ch.9.2 of [Aho *et al.*, 2007]). It works on the CIL [Necula *et al.*, 2002] presentation of the function, after a standard control flow graph is constructed.

Figure 2.2 shows the analysis itself, including definition of the flow state and transfer functions. In the following paragraphs, we explain the analysis in more details following the figure.

$f_{\mathrm{f}}(x)$ is the most important transfer function. It is used for all function calls made by the current function. It basically applies the actions of these functions to the current flow state. It also covers calls to primitives like `local_irq_disable()`, `spin_lock_irq()` (both turns local IRQ off). These primitive functions have proper actions through annotations. For example, `local_irq_disable()` and `spin_lock_irq()`



Flow state is a triple:

$$x = (\text{flags}, \text{action}, \text{saved})$$

where the members are,

flags	the set of possible context flags
	each flag is (in_irq,irqs_enabled)
action = $\{\uparrow \mid \downarrow \mid \rightarrow\}$	the function *enables/disables* or *keeps* irq
saved	a map of saved context flags

Transfer functions:

$$\begin{array}{lcl}
f_f(x) & = & \begin{cases} x & \text{if fun() has action } \rightarrow \\ (E(x.\text{flags}), \uparrow, x.\text{saved}) & \text{if fun() has action } \uparrow \\ (D(x.\text{flags}), \downarrow, x.\text{saved}) & \text{if fun() has action } \downarrow \\ (D(x.\text{flags}), \downarrow, \{v \mapsto x\} \cup x.\text{saved}) & \text{if fun() is irq\_save(v)} \\ x.\text{saved}(v) & \text{if fun() is irq\_restore(v)} \end{cases} \\
f_{\text{if,e}}(x) & = & (G(x.\text{flags}, e), x.\text{action}, x.\text{saved}) \\
f_{\text{else,e}}(x) & = & (G(x.\text{flags}, \neg e), x.\text{action}, x.\text{saved}) \\
f_{\text{join}}(x,y) & = & \begin{cases} (x.\text{flags} \cup y.\text{flags}, x.\text{action}, x.\text{saved} \cap y.\text{saved}) & \text{if } x.\text{action} = y.\text{action} \\ \top & \text{otherwise} \end{cases}
\end{array}$$

where,

$$\begin{array}{lcl}
E(\text{flags}) & = & \{(l.\text{in\_irq}, T) | l \in \text{flags}\} \\
D(\text{flags}) & = & \{(l.\text{in\_irq}, F) | l \in \text{flags}\} \\
G(\text{flags}, e) & = & \begin{cases} \{l \mid l \in x.\text{flags} \wedge l.\text{in\_irq} = T|F\} & \text{if } e \implies \text{in\_irq} = T|F \\ \{l \mid l \in x.\text{flags} \wedge l.\text{irqs\_enabled} = T|F\} & \text{if } e \implies \text{irqs\_enabled} = T|F \\ \text{flags} & \text{otherwise} \end{cases}
\end{array}$$

Figure 2.2: Dataflow analysis of execution contexts

are both annotated to have action ↓ (disabling irqs). The function $E(x)$ and $D(x)$ return the flags after the interrupts are enabled or disabled respectively.

The last two cases in $f_{\mathrm{f}}(x)$ handles the saving and restoration of local irq state. In particular `spin_lock_irqsave(lock,flags)` and friends save the interrupt related flags in `flags`, and then disables interrupts. Conversely `spin_lock_irqrestore(lock,flags)` restores the flags stored in `flags`. In addition, there are functions that calls `spin_lock_irqsave()` and returns the flags. We rely on the programmer to provide annotations for these functions so that the analysis can track them properly.

Here are two examples of using the `IRQ_SAVE` annotation. Note that `__ret` is a keyword supported by the analysis, denoting the function's return value. This shows that the annotations accept lvalues as argument and is flexible enough for most cases.

```
unsigned long read_lock_irqsave(rwlock_t *lock) IRQ_SAVE(__ret) {
  unsigned long flags;
  local_irq_save(flags);
  ...
  return flags;
}

static struct rq *task_rq_lock(struct task_struct *p, unsigned long *flags)
        IRQ_SAVE(*flags) {
  ...
  local_irq_save(*flags);
  ...
}
```

$f_{\mathrm{if}}(x)$ and $f_{\mathrm{else}}(x)$ are guards for `if` statements [1]. This is useful for understanding

[1]Note that in CIL, other C statements containing conditionals, for example `for` loops, are all rewritten to use `if` statements. So we only need guards for `if`s

code that tests the current execution context, and do different things in different contexts. For example, the following code snippet from `drivers/char/vt.c` illustrates why this is needed,

```
static int do_con_write(...)
{
  ...
  if (in_interrupt())
    return count;
  might_sleep();
  acquire_console_sem();
  ...
}
```

It tests whether the local CPU is in an interrupt service routine. If true, it returns immediately. If not, it proceeds to do things like acquiring a semaphore, which is only allowed in process contexts. Thus to understand that this is correct code, the analysis needs to understand the `in_interrupt()` test.

Cases like this are handled using the guard function $G(\text{flags}, e)$, which basically keeps those flags that matches the guard condition and drops the others. For example, the first rule states that, if the guard condition implies `in_irq` is true or false, then we only keep those flags that has `in_irq=T` or `F` respectively. To find out whether the `if` conditional implies the predicate in question, we use a fixed list of test functions and support basic boolean logic among the values. For example, the analysis understands that `(in_interrupt() && ...)` also implies `in_irq=T`. Also note that a special case is that if $G(\text{flags}, e) = \emptyset$, the path will be treated as unreachable and not analyzed.

Finally, $f_{\text{join}}(x, y)$ combines flow state from multiple blocks flowing into the same block. We require the code to behave consistently with regard to how it changes the interrupt mask no matter which path it takes. This is true in most cases. In the few cases when it does not, the state becomes $\top$. Then we have to add manual annotations or assertions to work around the problem.

When the analysis is done, the `action` field of the flow state is saved to the function summary.

2.3.5 Inter-Procedural Analysis

The inter-procedural part of the analysis propagates knowledge about behavior of separate functions throughout the program. Because the dataflow analysis may have used summaries of functions that have not been analyzed. When summaries of these functions become available, and in general, more precise, we may need to analyze the current function again. Therefore a function can be analyzed multiples times. A standard work list algorithm is used to schedule these runs of analyses. A function may be added to the work list for any of the following reasons.

- Initially each function is added to the work list.
- When the summary of any function that it calls changes. In other words, when after one round of dataflow analysis, the summary of the current function changes, all of its parents in the call graph are added to the work list again.
- When a call to a function changes the callee's precondition context flags, that is, the `flags` field of the function summary, then the callee is added to the work list again.

Annotation	Meaning
IRQ_ON	Must be called with IRQ enabled
IRQ_OFF	Must be called with IRQ disabled
IN_IRQ	Must be called in ISRs
NOT_IN_IRQ	Must never be called in ISRs
IRQ_SAVE(x)	This function saves IRQ flags into x and disables IRQ
IRQ_RESTORE(x)	This function restores IRQ flags from x
ACT_ENABLED	This function enables IRQ
ACT_DISABLED	This function disables IRQ
ACT_UNCHANGED	This function keeps IRQ unchanged
CALL1(fun)	This function calls `fun()`
CALL2(fun,arg)	Calls any function passed to `fun()` as the `arg`-th argument
CALL3(fun,arg,fld)	Calls any function stored in field `fld` of a struct that gets passed to `fun()` as the `arg`-th argument
UNSAFE	Warn if this function is reached
LOCK_NAME(x)	(On variables) assign name `x` to this lock

Table 2.1: Ctxcheck annotations

2.4 Checking Contexts and Spin Lock Uses

With summaries of functions computed, and possible execution context flags analyzed at each statement in the code, Ctxcheck does a set of checks based on this knowledge it has about the code, to try to find potential problems. The results are a list of problem cases that violates the rules, with a back trace for each explaining why it could happens.

In some cases, we want to give the analysis more knowledge to make it understand the program better, and therefore give better checking results. We do this by adding annotations. These annotations are macros that translate to custom GCC attributes. They are ignored by GCC during the normal build process, and becomes useful when we check the code using our tool. Table 2.1 list the major annotations currently implemented in the analysis.

There are also assertions that can be used to make sure certain conditions are true at a particular location. Table 2.2 list all assertions supported.

Assertion	Meaning
ASSERT_IRQ_ON	IRQ must be enabled
ASSERT_IRQ_OFF	IRQ must be disabled
ASSERT_IN_IRQ	Must be in ISRs
ASSERT_NOT_IN_IRQ	Must not be in ISRs
ASSERT_ACT_ENABLED	Current function should have enabled IRQ
ASSERT_ACT_DISABLED	Current function should have disabled IRQ
ASSERT_ACT_UNCHANGED	Current function should have kept IRQ unchanged

Table 2.2: Ctxcheck assertions

2.4.1 Checking Contexts

The tool does the following simple checks against execution contexts, It will print out a warning if any of the following happens,

1. A function is called in a context that conflicts with its summary. For example, a function with `must_in_irq=F` and called from a location that has `in_irq=T` is a problem.
2. An assertion is violated. For example, when `ASSERT_ACT_ENABLED()` is met and the current location has `action=`$\rightarrow$.
3. An UNSAFE function is called and it is in one of the basic blocks of a guarded `if` statement that tests some context-related predicate.

The last rule merits some explanation. We annotate kernel functions like `BUG()`, `halt()` as unsafe, because almost all uses of them are indications of something wrong. So it makes sense to report places where these could be called. However, there are a lot of possible reasons they could be called and execute context is only one of them. So we approximate separating out all context-related problem by requiring the unsafe call to be in the basic block immediately after an `if` statement that tests some context-related predicate. We show two examples of this in Figure 2.3. Case (a) will be reported as a possible bug if it is called with IRQ on. But case (b) will not be

```
if (in_interrupt())
  BUG();
```

(a)

```
if (in_interrupt()) {
  ...
  do {
    BUG();
  } while (...)
}
```

(b)

Figure 2.3: Examples of `if` guards

reported as a problem, because the `BUG()` call is not in the immediate basic block after `if`.

Obviously this is not sound because it may miss some problems actually due to context reason, as case (b) may well be. But it eliminates a lot of false positives and reports mostly real problems. So we think the sacrifice of soundness in this case is worthwhile.

When working with the kernel code, we added some annotations to the Linux kernel code base, so that the tool can better understand the code and reports more useful results. Without annotations, a lot of functions already have correct summaries. But there are places where either we reported false problems or missed opportunities to do checks. The annotation process is basically an iterative one: We run the tool, see if the analysis gets the function summaries correct. If some are wrong, there will usually be a lot of false problems. Then we add some annotations, or make some other changes if necessary, and run the tool again, so on.

We discuss more details of what and how much we changed in Section 2.5.

2.4.2 Checking Spin Locks

The current version of our tool checks for one particular kind of problem in spin lock usage, namely locks that are used both in ISRs at location A and with interrupts enabled at location B. This is a problem because as interrupts can happen at any time, when the lock is held with interrupts enabled at location B, if an interrupt happens and the ISR wants to acquire the same lock at location A, deadlock happens. Problems of this kind are often hard to find because some seemingly harmless function call may make a lock change its safety property. And in most cases these bugs are not easy to reproduce, but real bugs.

In order to statically find these bugs, first we need to name the locks in the program, in addition to the context information we already have after the analysis. For this we simply assign static *lock classes* to locks. A large portion of locks used are static locks. So we simply use their variables' names as the locks' classes. We found that almost all dynamically allocated locks are stored inside structures. So for most of them, combining the struct's type name and the field name gives a unique name for the lock class. For most cases, using these static lock classes do not yield false positives, indicating that locks of the same class are often used consistently as expected. A side note explaining why this works, is that the code almost never holds multiple locks of the same class simultaneously, as database systems often do when accessing multiple tables in a transaction. We think part of the reason is that such a design would be prone to deadlocks and operating systems do not normally contain deadlock detection mechanisms. So in general, lock usage is much more static in nature compared to generic data processing systems like databases.

There are a few cases where static lock classes do not work. Most of them are generic data structures containing locks and related to concurrency, for example "wait queues". Different instances of these are used in different contexts. Collapsing the locks in these based on type will result in false positives. We currently do not check these locks. Solving this problem is future work.

After we have named the locks, we examine calls like `spin_lock()` and simply record every possible context in which a lock can be used and its corresponding location. When an incompatible context is added to the list of a lock, we report it as a problem.

2.4.3 Error Reporting

Once we found a problem, be it a context problem or a spin lock problem, it is important that we print detailed and useful information that can help understand whether it is a real problem or a false positive, and if it is real, the source of the problem.

The most important piece of information the user needs to understand the result is back traces leading to particular execution context flags in a function summary. In order to print this out, we keep a "back link" whenever we change the flags during the dataflow analysis, or propagate a flag to a function during the inter-procedural propagation. At the end of the analysis, the tool can be told to prints out a report listing all summaries of all functions and explain why each flag gets there. This is done by following the "back links" until we find a change of the flags, for example enabling of interrupts. Note that there can be cycles in the links and care is taken to avoid them.

Now let us look at an actual example. Figure 2.4 shows an output snippet of our tool. The second half points out a bug in the 8250 serial port driver. The lock `.uart_port.lock` (the one protecting each port), is used in both ISRs ("IN sites", in total 5 sites) and with interrupts enabled ("EN sites", 2 sites). The first half shows details about the function `receive_chars()`, which is among the sites acquiring the lock. It shows two back traces. One not in ISR and with IRQ enabled ("IN-,EN+"), the other in ISR and with IRQ enabled. The first back trace shows that the call originates from `__run_timers()`, and the reason why IRQ is on is because in that function, `raw_local_irq_enable()` is called, and so on.

With this output, it is relatively easy to find the reason why the analysis thinks

```
receive_chars()
  Summary: =
  Flags:
  Preconditions:
    IN-,EN+ (1) :
      serial8250_handle_port() at drivers/serial/8250.c:1346
      serial8250_backup_timeout() at drivers/serial/8250.c:1526
      __run_timers() at kernel/timer.c:577, and call to
          raw_local_irq_enable() at kernel/timer.c:574
    IN+,EN+ (1) :
      serial8250_handle_port() at drivers/serial/8250.c:1346
      serial8250_interrupt() at drivers/serial/8250.c:1387
      handle_IRQ_event() at kernel/irq/handle.c:141, and call to
         raw_local_irq_enable() at kernel/irq/handle.c:138

...

.uart_port.lock : IN:5 EN:2
  IN sites:
    * uart_start() at drivers/serial/serial_core.c:114
    * uart_stop() at drivers/serial/serial_core.c:93
    * serial8250_console_write() at drivers/serial/8250.c:2369
    * receive_chars() at drivers/serial/8250.c:1269
    * serial8250_handle_port() at drivers/serial/8250.c:1339
  EN sites:
    * receive_chars() at drivers/serial/8250.c:1269
    * serial8250_handle_port() at drivers/serial/8250.c:1339

Unsafe: lock .uart_port.lock used in both ISRs and with IRQ enabled.
See above for more info.
```

Figure 2.4: Example output snippet of Ctxcheck

there are problems, and to tell if they are false positives and real bugs in the code.

2.5 Evaluation

The current implementation of Ctxcheck is written in about 3000 lines of OCaml, on top of the CIL compiler framework [Necula *et al.*, 2002]. In this section, we discuss results of applying Ctxcheck to the Linux kernel.

2.5.1 Checking the Linux Kernel

We work with a modified version of Linux kernel 2.6.20.7. In our evaluation we focus on several parts of the kernel, including all indispensible sub-systems (boot, scheduling, virtual memory and etc), the networking stack, 32 network card drivers (18 10M/100M cards and 14 1000M cards) and several other miscellaneous drivers. Apart from these, the kernel is minimally configured. For example, we did not compile in a file system.

Because our analysis requires whole program analysis, we modified the Linux build scripts so that when asked to compile and link files, they call the CIL merger to merge preprocessed source code together instead of compiling them into object code. The pre-processed and merged single-file kernel as configured above is 855K lines of code (26MB in size).

We did the following modifications to the kernel for the analysis to better understand the code and do checking.

- The kernel contains some explicit runtime checks of contexts. We annotate the checking functions once and take advantage of all the checks immediately. For example, `might_sleep()` dictates that the current function might sleep and should not be used in atomic contexts, including hardware interrupt contexts.

We annotate it as `NOT_IN_IRQ`. Thus Ctxcheck should report a problem if code that could execute in interrupt context calls `might_sleep()`.

- We annotate many "entry" functions with proper context information, so that they can propagate context to other parts of the kernel. In particular, all system call functions and module init/exit functions are annotated as `IRQ_ON, NOT_IN_IRQ`. And the function that calls all ISRs, `handle_IRQ_event()` is annotated as `IN_IRQ, IRQ_OFF`.

- We annotate some commonly used functions that have context requirements. This task requires knowledge about the internals of the kernel. We rely on our understanding, kernel documentation and existing usage scenarios in the kernel to figure out what functions need to be annotated and how to annotate them. For example, mutexes and semaphores can not be used in interrupt context. So we annotate these as `NOT_IN_IRQ`. Another example is that `smp_call_function()` is annotated to be `IRQ_ON` because it requires IRQ to be enabled when called.

- For a few cases where the analysis still cannot proceed, as workarounds, we add assertions to dictate the state and behavior of the code to the analysis. For example, the function `lock_task_sighand()` behaves as `IRQ_SAVE` when it succeeds (and returns non-zero), but does not turn IRQ off when it fails (and returns 0). We still annotate the function as `IRQ_SAVE` and add `ASSERT_ACT_UNCHANGED()` at the call site along the failures path to dictate the correct behavior.

Annotation	Number
IRQ_ON	546
IRQ_OFF	466
IN_IRQ	64
NOT_IN_IRQ	44
IRQ_SAVE	9
IRQ_RESTORE	8
ACT_ENABLED	1
ACT_DISABLED	1
ACT_UNCHANGED	25
LOCK_NAME	97
UNSAFE	4
CALL3	12

Table 2.3: Number of Ctxcheck annotations added to Linux 2.6.20.7

2.5.2 Annotation Burden

We measured the amount of annotations needed for the tool to yield useful results. In total we changed 1792 lines in the code (0.2% of the merged kernel code). Most of the changes are addition of annotations. Table 2.3 is a break-down of the annotations we added.

We estimate that the above changes are about 3 days of work. We believe this is mostly a one-time cost, because the contexts of functions should not change very often as the code evolves. As we can see, most of the annotations are `IRQ_ON` and `IRQ_OFF` ones. Part of the reason is that we do not currently do inference for these flags yet. So we still need these annotations for many functions, for example, all system call entry functions. We think the addition of inference for them will reduce the total number of annotations needed significantly.

Note that it is not easy to tell when there are *enough* annotations in the code. One way to measure the "coverage" of the annotations is to measure the percentage of *unknown* flags in the results. This gives a rough gauge of how much knowledge the analysis has about the code. Because we do not take unknown flags into account

Entity	Number
Functions	12180
Orphan functions	209 (1.7%)
Callsite traces	101975
Unknown inIrq calls	5617 (5.5%)
Unknown irqEnable calls	22434 (22.0%)

Table 2.4: Ctxcheck annotation coverage statistics

for error checking, unknown flags lead to missed errors. Therefore this also gives a measure of how much "unsoundness" there is.

With the above annotations, the statistics are shown in 2.4. The first part shows that 1.7% of all functions are *orphans*, that is we do not know anything about the contexts they could execute in. Inspection shows that most of them are code unreachable in our configuration of the kernel. Ctxcheck tries its best to remove unreachable code for the purpose of the analysis under the close-world assumption [2]. However it does this conservatively and there are cases where unreachable code is left untouched. The remaining orphans are due to valid entry functions that do not have definite context settings. Therefore we cannot annotate them with our current set of annotations. Both these cases could lead to false negatives. The second half of the table shows how these orphan functions "pollute" the rest of the kernel. We can see that the percentage of unknown `irqEnable` calls is still quite large. Removing dead code more effectively, and add more expressive annotations to describe the indefinite functions, are both future work.

2.5.3 Analysis Performance

We timed the analysis on a dual-processor Pentium 4 Xeon 2.8Ghz server. A full merge of the kernel took 10 minutes 18 seconds. When only a few files are modified,

[2]Note that we may remove code that is needed by dynamically loaded kernel modules at runtime. But that is okay for the sake of the analysis.

Analysis step	Duration
CIL parsing	39.7s
Points-to analysis & call graph gen.	3.2s
Propagation of must_in_irq flags	0.7s
Main analysis	27.2s
Printing final report	6.3s
Others	6.0s

Table 2.5: Timing statistics for analyzing Linux 2.6.20.7 using Ctxcheck

the merge process is much faster, generally taking 1-2 minutes.

Running the Ctxcheck tool itself on the merged kernel took 83 seconds. This is quite fast considering the size of the kernel. Table 2.5 shows the break-down of the time. Note that nearly half of the time is spent in CIL parsing and preprocessing. The printing of the report took 5 seconds, most of which is spent searching for back traces explaining context flags. We think this could be made faster by caching or precomputing partial back traces as a lot of them have common segments. This is not implemented yet because currently it is not a performance problem.

We would like to point out that the performance of the analysis is very sensitive to the precision of the points-to analysis. And this is actually the reason why we use a specialized points-to analysis tailored to function pointer usage in the kernel. Before that we tried the One-Level Flow algorithm [Das, 2000]. Although performance of the points-to analysis is okay (about 1 minute for the same kernel). It does not work well for function pointers in the kernel, generating so many false paths that the main analysis cannot complete within 30 minutes.

2.5.4 Bugs Found

A run of Ctxcheck on the merged kernel yields 81 warnings. Among these, 21 are about locks that are used in both `in_irq` and `irq_enabled` contexts. The remaining 60 are context-related warnings. We do not expect a lot of these to be real bugs, as this kernel version is considered "stable" and as we will discuss in the related work section, the kernel has integrated a runtime checking tool covering the bugs Ctxcheck checks for over half a year.

Nevertheless after we examined the results, we found four cases that are definitely bugs. In fact, when we checked the latest kernel version, we found that these are already fixed in 2.6.21. All four are similar problems in a network card driver `qla3xxx.c`. One example is in `ql_link_state_machine()`,

```
spin_lock_irqsave(&qdev->hw_lock, hw_flags);
...
if (test_bit(QL_RESET_ACTIVE,&qdev->flags)) {
  if (netif_msg_link(qdev))
    printk(KERN_INFO PFX
      "%s: Reset in progress, skip processing link "
      "state.\n", qdev->ndev->name);
      return;
}
...
spin_unlock_irqrestore(&qdev->hw_lock, hw_flags);
```

The problem is that when the second `if` tests true, the function returns without unlocking `&qdev->hw_lock`, and also fails to restore the IRQ mask. The analysis found that the function returns with inconsistent IRQ mask. So it reports a potential bug. We believe this bug slipped through testing and the runtime tool because these error paths are hard to exercise for both methods. This shows Ctxcheck, as a static tool, has better coverage than runtime checking tools.

There are another 5 warnings that we think are probably bugs. We are in the process of confirming these with kernel developers. For example, one of them is that the function `kmap_skb_frag()` could be reached while processing segmented packets in ISRs, but the function is clearly marked as "should not be called in IRQ".

To further evaluate the usefulness of Ctxcheck in finding IRQ context and locking related bugs. We browsed the Linux kernel mailing for relevant bug reports and fixes. We found 5 of them that are recent enough that we can "port" to our kernel. Of the 5 bugs, Ctxcheck successfully finds 2 of them [3]. Both of them are locks usage bugs. Of the remaining three, one requires understanding of software IRQs, which the analysis does not have. The other two are related to multi-lock deadlocks, also not covered by the analysis yet.

Although these results are still preliminary, we can see that Ctxcheck is effective in finding real and non-obvious bugs in a large code base like the Linux kernel. What has not been shown by this evaluation is that, we believe tools like Ctxcheck would be very useful in the early development phase of new code for the kernel, like new device drivers. Code in this stage tends to be much buggier than code already in the mainline tree. Ctxcheck can help the developer find bugs without actually running the code. Since interrupt related bugs and deadlocks are always hard to debug at runtime, we expect the ability to find bugs statically to be very helpful.

2.6 Discussion

Here we discuss a couple of interesting issues related to the current Ctxcheck implementation and future work.

Currently Ctxcheck is a whole-program analysis, mostly because context is a piece of global state. This may be a problem for very large code bases. For example if we configure in all Linux device drivers, the merging and analysis may take too long.

[3]Original bug reports: `http://groups.google.com/group/linux.kernel/browse_thread/thread/66ec98c14c879ce3/2b539acafe962d6b`, `http://groups.google.com/group/linux.kernel/browse_thread/thread/b143b4281c9f9a54/188cc2c69ca8510c`

One way around this is to divide the system into a number of large modules, and analyze one module at a time, using annotations and summaries on the interface functions between the modules, and output inferred annotations and summaries for analysis of other modules. One particular set-up is to divide the kernel into basic kernel subsystems as a large module, and each device driver as a module. This way, the large kernel subsystems only need to be analyzed once, and the analysis output saved. Then for each driver, the analysis only needs to work on a very small program using the saved results, which should make the analysis very fast. This would make Ctxcheck a useful tool for device driver developers, especially those new to Linux driver development. In addition, the annotations and summaries of kernel API function should make useful documentation of the kernel API, whose usage conventions regarding contexts are often non-obvious.

Another issue regards the result presentation of the tool. Currently in the result report, we normally list a single back trace as the reason for a function to have a particular flag. However, this trace may be a false positive, leaving the user wondering whether there is another *valid* path leading to this. Currently there are two ways you can handle this. First you can annotate the code to remove the false path, and rerun the analysis. This process will probably take a couple of minutes and thus tedious if there are a lot of error cases to examine. Another way is to rerun the tool and tell it to output multiple traces for each flag. This is easier sometimes. But it makes the analysis take longer and may generate a lot of clutter. Moreover, it is often hard to say how many traces per flags is enough. Thus it still gets tedious sometimes. We think a better way would be to provide an interactive interface for examining the results right after the analysis is done. When activated, this would allow the user to tell the tool to “explain” a particular flag with arbitrary number of traces. This would make examining the results easier than it currently is.

The process context and hardware interrupt context that Ctxcheck tracks are not exhaustive. There are more similar concepts in the kernel that could be analyzed. For example, *softirq*, or so-called "software interrupts" are a commonly used context somewhat in the middle of process context and hardware interrupt context. Ctxcheck currently treats softirq context as hardware interrupt context, and this generates false positives, as there are some combinations of states that are actually allowed but the analysis treats as illegal. Adding softirq support involves adding more state bits to the function summaries and dataflow state. It should be a relatively straight-forward addition.

2.7 Related Work

`Lockdep` [Molnar and van de Ven, 2007] is a set of concurrency related runtime checks for the Linux kernel. It has been integrated into the mainline Linux kernel and is considered to be very useful by Linux kernel developers. `Lockdep` includes interrupt related checks that are quite similar in nature to those performed by Ctxcheck. Actually Ctxcheck is inspired partly by `locdep`. However `lockdep` checks are done at runtime. So it incurs significant overhead and can only find errors in code that are actually exercised at runtime. Note that `lockdep` does not need the actual error condition to happen, but only needs to observe enough information that makes a potential error *possible*, just like Ctxcheck. But still Ctxcheck, being a static analysis, achieves more coverage than `lockdep`. Currently `lockdep` includes many more checks than Ctxcheck does, for example it checks for multi-lock dead locks.

One direction for future work is to add more checks to Ctxcheck and see how many of the checks that `lockdep` does, which have been deemed very useful by the Linux kernel community, could be made static. The biggest missing piece seems to be the tracking of what locks are acquired at each location. With that we should be able to check for multi-lock deadlocks and context problems involving multiple locks.

Another possible direction is to combine `lockdep` and Ctxcheck to create a *hybrid* concurrency checking tool. With this tool, Ctxcheck is run on the code during the build process. And for all the locks and portion of code that are verified to be safe with regard to a set of checks, they do not need to be checked at runtime again. For the rest of locks and code, runtime checks are inserted so that their safety can be verified. This guarantees safety while having the benefit of lower overhead than a full-runtime checking tool, and also finds many problems during the build process. Depending on how low the overhead could be, this could potentially make the tool deployable on production systems, which would help a lot in detecting hard-to-reproduce bugs.

SLAM [Ball *et al.*, 2001], BLAST [Henzinger *et al.*, 2002] and MAGIC [Chaki *et al.*, 2004] represent a family of recently-developed software analysis engines based-on the *abstract-check-refine*, or iterative refinement, paradigm. They build an abstract model of the program, check certain properties, refine the model if it is not precise enough, and then start over again. In this way they can get precise results with a model of the program that has only enough details for the properties to be checked. Microsoft's SDV [Ball *et al.*, 2006] uses SLAM to statically analyze Windows device drivers looking for temporal API usage problems, like acquiring a spin lock twice. The problems that SDV checks include concurrency-related ones that bear similarity with those Ctxcheck checks. However, these tools have different focuses. SDV focuses on device drivers (up to a few hundred of thousand lines according to [Ball *et al.*, 2006]) with a relative fixed API interface. In contrast, Ctxcheck checks the entire kernel using simpler analyses and more domain-specific techniques, and scale to millions of lines of code easily.

Sparse [Torvalds and Triplett, 2003] is a static analysis tool developed by the Linux kernel community specifically for the Linux kernel. It supports simple checks like user/kernel pointers and unpaired lock/unlock calls. It does not currently understand interrupt contexts though. Compared to sparse, Ctxcheck is built on a more flexible and analyzable framework (CIL), and uses more sophisticated techniques and thus understands the code better. The checks already in sparse should also be doable in Ctxcheck too.

2.8 Contexts and Locks Conclusion

In this chapter we have presented Ctxcheck, a tool that statically analyzes the execution contexts and spin lock usage in the Linux kernel and their interactions. With the help of some added annotations (about 0.2% of the kernel code changed). the tool can analyze the the whole kernel in a few minutes and find useful bugs in these aspects.

Chapter 3

Type Safe and Recoverable Device Drivers

3.1 Introduction

Operating systems, and other large and general systems like web servers, often provide an extensibility mechanism that allows the behavior of the system to be customized for a particular usage scenario. For example, device drivers adapt the behavior of an operating system to a particular hardware configuration, and web server modules adapt the behavior of the web server to the content or performance needs of a particular web site. However, such extensions are often responsible for a disproportionately large number of bugs in the system [Chou *et al.*, 2001; Swift *et al.*, 2003], and bugs in an extension can often cause the entire system to fail. In this chapter our goal is to improve the reliability of extensible systems without requiring significant changes to the core of the system. To do so, we must *isolate* existing extensions, preferably with little modification, *restore system invariants* when they fail, *restart* them automatically for availability, and (ideally) *restore active sessions.*

In this chapter, we present techniques to improve the dependability of commodity operating systems by making device driver more reliable. Previous systems have attempted to address this problem using some form of lightweight protection domain for extensions. For example, the Nooks project [Swift *et al.*, 2004; Swift *et al.*, 2003] runs Linux device drivers in an isolated portion of the kernel address space, modifying kernel API calls to move data into and out of the extension. This approach prevents drivers from overwriting kernel memory at the cost of relatively expensive driver/kernel boundary crossings.

Our system, SafeDrive, takes a different approach to improving extension reliability. Instead of using hardware to enforce isolation, SafeDrive uses language-based techniques similar to those used in type-safe languages such as Java. Specifically, SafeDrive adds type-based checking and restart capabilities to *existing* device drivers written in C without hardware support or major OS changes (i.e., without adding a new protection domain mechanism). We have four primary goals for SafeDrive:

- **Fine-grained type-based isolation**: We detect memory and type errors on a per-pointer basis, whereas previous work has only attempted to provide per-extension memory safety. SafeDrive ensures that data of the correct type is used in kernel API calls and in shared data structures. This advantage is critical, because it means that SafeDrive can catch memory and type violations before they corrupt data, even for violations that occur entirely within the driver. Thus, we can prevent the kernel or devices from receiving incorrect data for these cases. SafeDrive can also catch more memory-related bugs than hardware-based approaches; specifically, SafeDrive can catch errors that violate type safety but do not trigger VM faults. In addition, because errors are caught as they occur, SafeDrive can provide fine-grained error reports for debugging.

- **Lower overhead for isolation**: SafeDrive exhibits lower overhead in general, particularly for extensions with many crossings (for which Nooks admits it is a poor fit [Swift *et al.*, 2003, Sec. 6.4]). Compared with SafeDrive, hardware-enforced isolation incurs additional overhead due to domain changes, page table updates, and data copying. Moreover, these techniques change the semantics of various operations subtlely, especially for code dealing with complex data strucutres requiring deep copying and sharing. When concurrency between the kernel and the extensions is involved, this again poses new challenges as there may be multiple copies of a data structure at a certain point of time. Also, the stronger type invariants that SafeDrive maintains makes it possible to check many pointer operations statically.

- **Non-intrusive evolutionary design.** SafeDrive provides type safety without changing the structure of the host system (e.g., the OS kernel) significantly and without rewriting extensions to use a new language or API. Moreover the binary interface between the host system and the extension is unchanged. Therefore the system is able to use both SafeDrive-enabled extensions and non-SafeDrive-enabled extensions. This way extensions can be migrated to use SafeDrive gradually over time.

- **Protection against buggy (but not malicious) extensions.** In rare cases where true type safety would require significant changes to the extension or to the host API, we prefer to trust individual operations whose safety we cannot verify and gradually migrate to more complete isolation over time. For example, our current implementation does not yet attempt to verify memory allocation, deallocation, and mapping operations. In addition, we make no attempt to protect the system from extensions that abuse CPU, memory, or other resources (unlike OKE [Bos and Samwel, 2002] and Singularity [Patel *et al.*, 2003]). As a consequence, SafeDrive is able to guard against mistakes made by the author of the extension but does not attempt to protect against a malicious adversary capable of exploiting specific behaviors of our system.

C and its variants have a number of important constructs that can be used to cause violations. In addition to the most obvious issue of out-of-bounds array accesses, C also has fundamental problems due to unions, null-terminated strings, and other constructs. To transform a driver written in C (which includes all Linux drivers) into one that obeys stricter type safety requirements, we must fix all of these flaws without requiring extensive rewrites and ideally without requiring modifications to the kernel.

The existing approaches to type safety for C involve the use of “fat” pointers, which contain both the pointer and its bounds information. CCured [Necula *et al.*, 2005], for example, can make a legacy C program memory-safe by converting most of its pointers into fat pointers and then inserting run-time checks to enforce bounds constraints. However, this approach is not realistic for drivers or for the kernel, since it modifies the layout of every structure containing pointers as well as every kernel API function that uses pointers. To use CCured effectively in this context, we would need to “cure” the entire kernel and all of its drivers together, which is impractical.

Instead, SafeDrive employs a novel type system for pointers called Deputy, developed by Condit et al [Condit *et al.*, 2007]. Deputy is a dependent type system designed for low-level programming. It enforces memory safety for most programs without resorting to the use of fat pointers and thus without requiring changes to the layout of data or the driver API. The key benefit Deputy provides is that it exploits the insight that most of the required pointer bounds information is already present in the driver code or in the API—just not in a form that the compiler currently understands. Deputy relies on a set of type annotations to identify where this information is in places where it already exists. In SafeDrive, annotations are added to Linux kernel header files and driver source files. Therefore operations within the driver, including function calls and accesses to data structures are checked for type safety. And all calls to kernel functions and accesses to exported kernel data structures are also checked. But any operation inside the kernel is trusted to preserve type safety.

Similar to Ctxcheck as we discussed in Chapter 2, there is legitimate concern that adding annotations to kernel headers or driver code may be a burden. However, there are many reasons why this approach is a practical one for type safety. First, the required annotations are typically very simple, allowing programmers to easily express known relationships between variables and fields (e.g., "`p` points to an array of length `n`"). Second, the cost of annotating kernel headers is a one-time cost; once the headers are annotated, the marginal cost of annotating additional drivers is much smaller. Third, annotations are only mandatory for the driver-kernel interface, while the unannotated data structures internal to the driver can use fat pointers. About 600 Deputy annotations were added to kernel headers for the 6 drivers we tested (Section 3.6.3).

Any solution based on run-time enforcement of isolation must provide a mechanism for dealing with violations. In SafeDrive we assume that extensions are restartable provided that certain system invariants are restored. In the case of Linux device drivers, invariant restoration consists of releasing a number of resources allocated by the driver and unregistering any name space entries registered by the driver (e.g., new device entries or file system entries), both of which we will refer to as *updates*. In order to undo updates during fault recovery, we track them using wrappers for the relevant API calls. Because SafeDrive allows an extension to operate safely in the kernel address space without the use of a hardware-enforced protection domain, the task of managing and recovering kernel resources is greatly simplified.

As with Nooks, SafeDrive cannot prevent every extension-related crash, nor can it guarantee that malicious code cannot abuse the machine or the device. However, because SafeDrive can catch errors that corrupt the driver itself without corrupting other parts of the system, we expect it to catch more errors, and we expect it to catch them earlier.

We have implemented SafeDrive for Linux device drivers, and we have used the system on network drivers, sound drivers, and video drivers, among others. Our experiments indicate that SafeDrive provides safety and recovery with significantly less run-time overhead than Nooks, especially when many calls to/from the driver are made (e.g., 12% vs. 111% overhead in one benchmark, and 4% vs. 46% in another). The main conclusion to draw from these experiments is that language-based techniques can provide fine-grained error detection at significantly lower cost by eliminating the need for expensive kernel/extension boundary crossings.

In Section 3.2, we present an overview of the SafeDrive system, including the compile-time and run-time components. In Section 3.3, we provide a brief overview of the Deputy type system and compiler. Then, in Section 3.4 and Section 3.5, we describe in detail the *deputization* of Linux device drivers and the implementation of the recovery system. Section 3.6 describes our experiments. Finally, Section 3.7 and Section 3.8 discuss related work and our conclusions.

3.2 SafeDrive Overview

In SafeDrive, isolated recoverable extensions require support from the programmer, the compiler, and the runtime system. The programmer is responsible for inserting type annotations that describe pointer bounds, and the compiler uses these annotations to insert appropriate run-time checks. The runtime system contains the implementation of the recovery subsystem.

The compiler is implemented as a source-to-source transformation. Given the annotated C code, the Deputy source-to-source compiler produces an instrumented version of this source code that contains the appropriate checks. This instrumented

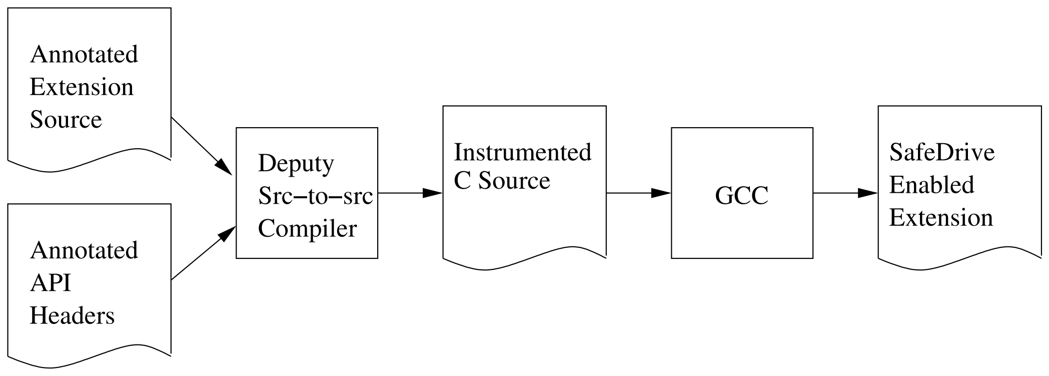

Annotated Extension Source
Annotated API Headers
Deputy Src–to–src Compiler
Instrumented C Source
GCC
SafeDrive Enabled Extension

Figure 3.1: Compilation of an extension in SafeDrive

code is then compiled with GCC. Most of the annotations required for this process are found in the system header files, where they provide bounds information for API calls in addition to the recovery system interface. The compilation process is illustrated in Figure 3.1.

At run time, a SafeDrive-enabled extension is loaded into the same address space as the host system and is linked to both the host system and the SafeDrive runtime system. Because all code runs in the same address space, no special handling of calls between the host system and the extension is required, apart from the run-time checks that the Deputy compiler inserts based on each function's annotations. The extension can read and write shared data structures directly, again using the Deputy-inserted run-time checks to verify the safety of these operations. Note that the host system does not need to be annotated or compiled with the Deputy compiler. Another consequence of this binary compatibility property is that non-SafeDrive extensions can be used with a SafeDrive-enabled host system as-is.

The SafeDrive runtime system is responsible for "update tracking" and recovery. It maintains data structures tracking the list of updates the extension has done to kernel state. Whenever a Deputy check fails, the control is passed to the SafeDrive runtime system, which attempts to cancel out these updates and restart the failed

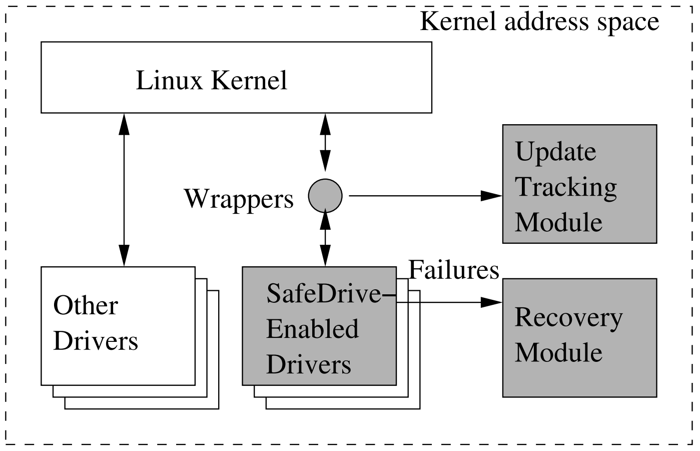


Figure 3.2: Block diagram of SafeDrive for Linux device drivers. Gray boxes indicate new or changed components.

extension. Figure 3.2 shows the structure of the SafeDrive runtime system for the Linux kernel.

Although this chapter focuses on Linux device drivers, we believe that the principles used in SafeDrive could be applied to a wide range of other extensible systems without a large amount of additional effort.

3.2.1 Example

Figure 3.3 shows some sample code adapted from a Linux device driver. Here the programmer has added the Deputy annotation "`count(info_count)`" to the `buffer_info` field, which indicates that `info_count` stores the number of elements in this array. The original code contained the same information in comments only; thus, the existence of this relationship between structure fields was previously hidden to the compiler.

Code in italics shows run-time checks that SafeDrive inserts into the instrumented version of the file (see Figure 3.1) to enforce isolation and to support recovery. The first and third checks are assertions that enforce memory safety. The first check

```
struct e1000_tx_ring {
  ...
  unsigned int info_count;
  struct e1000_buffer * count(info_count)
      buffer_info;
  ...
};

static boolean_t
e1000_clean_tx_irq(
    struct e1000_adapter *adapter,
    struct e1000_tx_ring *tx_ring)
{
  ...
  assert(tx_ring != NULL);
  spin_lock(&tx_ring->tx_lock);
  track_spin_lock(&tx_ring->tx_lock);
  ...
  i = tx_ring->next_to_clean;
  assert(0 <= i && i < tx_ring->info_count);
  eop = tx_ring->buffer_info[i]
                    .next_to_watch;
  ...
  track_spin_unlock(&tx_ring->tx_lock);
  spin_unlock(&tx_ring->tx_lock);
}
```

Figure 3.3: SafeDriveexample adapted from the Linux e1000 network card driver. The one programmer-inserted annotation is underlined. The italic code shows Deputy run-time checks that are inserted into the instrumented version of the file.

ensures that `tx_ring` is non-null, which is required because Deputy assumes that unannotated function arguments are either null or point to a single element of the declared type, much like references in a type-safe language. The third check in the example enforces the bounds declared for the `buffer_info` field before it is accessed. Note that a simple optimization ensures that we do not insert redundant checks every time `tx_ring` is dereferenced; indeed, the low overhead of SafeDrive is in part due to the fact that most pointer accesses require very simple checks or no checks at all.

If either of these assertions fail, SafeDrive invokes the recovery subsystem. To support recovery, SafeDrive inserts code to track the invariants that must be restored, as seen in the second and fourth italic statements in this example.

This example shows that SafeDrive preserves the binary interface with the rest of the kernel, inserts relatively few checks (tracked resource allocation is rare, and most pointers point to a single object), at the expense of a few annotations that capture simple invariants. In the next three sections, we describe the Deputy type system, discuss how we apply it to Linux and present the associated recovery system.

3.3 The Deputy Type System and Compiler

Deputy is developed by Condit et al. [Condit *et al.*, 2007], with SafeDrive [Zhou *et al.*, 2006] as a main application. As Deputy is quite new, and for the completeness of discussion on SafeDrive, we include here an overview of the Deputy type system and the corresponding compiler in this section. Readers interested in technical details of the Deputy type system and its soundness argument are referred to [Condit *et al.*, 2007].

The goal of the Deputy type system is to prevent pointer bounds errors through a combination of compile-time and run-time checking. Adding these checks is a challenging task because the C language provides no simple way to determine at run time whether a pointer points to an allocated object of the appropriate type. Previous systems, such as CCured [Necula *et al.*, 2005], addressed this problem by replacing pointers with multi-word pointers, known as "fat" pointers, which indicate the bounds for the pointer in question. Unfortunately, this approach changes the layout of data in the program, making it very difficult to "cure" one module or extension independently of the rest of the system.

In contrast, Deputy's types allow the programmer to specify pointer bounds in terms of other variables or structure fields in the program. Deputy's annotations also allow the programmer to identify null-terminated arrays and tagged unions. These annotations are flexible enough to accommodate a wide range of existing strategies for tracking pointer bounds. The key insight is that *most pointers' bounds are present in the program in some form—just not a form that is obvious to the compiler.* By adding appropriate annotations, we allow the compiler to insert memory safety checks throughout the program without changing the layout of the program's data structures.

Deputy is implemented as a source-to-source transformation that runs immediately after preprocessing, before the code is fed to the native compiler, GCC in our case. It is built on top of the CIL compiler framework [Necula *et al.*, 2002], and the current prototype contains about 20,000 lines of OCaml code. Deputy has three phases:

1. *Inference.* First, Deputy infers bounds annotations for every unannotated pointer in the program. For unannotated local variables, Deputy inserts an annotation that refers to new local variables that explicitly track the unannotated pointer's bounds; this approach is essentially a variant of the fat pointer approach. For globals, structure fields, and function parameters and results, Deputy assumes a default pointer type. We use the inference module from CCured [Necula *et al.*, 2005] to improve local variable annotations and to iden-

tify global variables that may require manual annotation.

2. *Type Checking.* Once all pointers have Deputy annotations, Deputy checks the code itself. It emits errors and run-time checks where appropriate.

3. *Optimization.* Since the type checking phase generates a large number of run-time checks, a Deputy-specific optimization phase is used to eliminate checks that will never fail at run-time and to identify checks that will definitely fail at run time.

3.3.1 Deputy Type Annotations

Here we discuss Deputy's type annotations and their associated run-time checks. These type annotations allow Deputy to verify code that uses unsafe C features, such as pointer arithmetic, null-terminated strings, and union types. Deputy annotations represent a reasonably large range of common C programming practices. They are simple enough to allow the type system to reason about them effectively, and yet they are expressive enough to be usable in real-world C programs. Note that annotations are *not* trusted by the compiler. Deputy checks when assigning a value to a variable that the value has the appropriate type; when using a variable, the type checker assumes that it contains a value of the declared type. In other words, these annotations function much like the underlying C types. Of course, Deputy only checks one compilation unit at a time, and therefore it must assume that other compilation units adhere to the restrictions of any Deputy annotations on global variables and functions, even if those other compilation units are not compiled by Deputy.

3.3.1.1 Buffers

Buffers are the most important construct that Deputy provides safety for. Deputy allows the user to specify the bounds of a pointer by using one of four type annotations: `safe`, `sentinel`, `count(n)`, and `bound(lo, hi)`. In these annotations, `n`, `lo`, and `hi` stand for expressions that can refer to other variable or field names in the immediately enclosing scope. For example, annotations on local variables can refer to other local variables in the same function, and annotations on structure fields can refer to other fields of the same structure. These annotations can be written after any pointer type in the program; for example, a variable named `buf` could be declared with the syntax `int * count(len) buf`, which means that the variable `len` holds the number of elements in `buf`. The meanings of these annotations are as follows:

- The `safe` annotation indicates that the pointer is either null or points to a single element of the base type. Such pointers are the most common kind of pointer in C programs, and they typically require only a null check at dereference.
- The `sentinel` annotation indicates that a pointer is useful only for comparisons and not for dereference. This annotation is typically used for pointers that point immediately after an allocated area, as permitted by the ANSI C standard.
- The `count(n)` annotation indicates that the pointer is either null or points to an array of at least `n` elements. When accessing an element of this array, Deputy will insert a run-time check verifying that the pointer is non-null and the index is between zero and `n`.
- The `bound(lo, hi)` annotation indicates that the pointer is either null or points into an array with lower and upper bounds `lo` and `hi`. When accessing this array or applying pointer arithmetic to this pointer, Deputy will verify at run time that the pointer remains within the stated bounds.

Deputy allows casts between pointers with these annotations, inserting run-time checks as appropriate. For example, when casting a `count(n)` pointer to a `safe` pointer, Deputy will check that `n >= 1`. Similarly, when casting a pointer `p` with annotation `count(n)` to a pointer with annotation `bound(lo, hi)`, Deputy will verify that `lo <= p` and that `p + n <= hi`.

The fourth annotation, `bound(lo, hi)`, is quite general and can be used to describe a wide range of pointer bound invariants. For example, if `p` points to an array whose upper bound is given by a sentinel pointer `e`, we could annotate `p` with `bound(p, e)`, which indicates that `p` points to an area bounded by itself and `e`. In fact, the `sentinel`, `safe`, and `count(n)` annotations are actually special cases of `bound(lo, hi)`; when used on a pointer named `p`, they are equivalent to `bound(p, p)`, `bound(p, p + 1)`, and `bound(p, p + n)`, respectively. Thus, when checking these annotations, it suffices to consider the `bound(lo, hi)` annotation alone.

The invariant maintained by Deputy is as follows. At run time, any pointer whose type is annotated with `bound(lo, hi)` must either be null or have a value between `lo` and `hi`, inclusive. That is, for a pointer `p` of this type, we require that `p == 0 || (lo <= p && p <= hi)`. (Note that ANSI C allows `p == hi` as long as `p` is not dereferenced.) Furthermore, all of these pointers must be aligned properly with respect to the base type of this pointer. Given this invariant, we can verify the safety of a dereference operation by checking that `p != 0 && p < hi`. In order to ensure that this invariant is maintained throughout the program's execution, Deputy inserts run-time checks before any operation that could potentially break this invariant, which includes changing the pointer itself or any other variable that appears in `lo` or `hi`. Since this process generates a large number of run-time checks, many of which are trivial (e.g., `p <= p`), Deputy provides an optimizer that is specifically designed to remove the statically verifiable checks that were generated by this process.

```
int * safe find(int * count(len) buf,
                int len) {
  assert(buf != 0);
  int * sentinel end = buf + len;
  int * bound(cur, end) cur = buf;
  while (cur < end) {
    assert(cur != 0 && cur < end);
    if (*cur == 0) return cur;
    cur++;
  }
  return NULL;
}
```

Figure 3.4: Example usage of Deputy bounds annotations. Underlined code indicates programmer-inserted annotations; italic code indicates checks performed by Deputy at run time.

Figure 3.4 presents an example of Deputy-annotated code. This function finds and returns a pointer to the first null element in an array, if one exists. This example shows several annotations at work. The first argument, `buf`, is annotated with `count(len)`, which indicates that `len` stores the length of this buffer. The return type of this function, `int * safe`, indicates that we return a pointer to a single element (or null). (This annotation is not strictly necessary since `safe` is the default annotation on most unannotated pointers.) In the body of the function, we use a sentinel type for `end`, indicating that it cannot be dereferenced. Also, we use `cur` and `end` for the bounds of `cur`, which allows us to increment `cur` until it reaches `end` without breaking `cur`'s bounds invariant.

The italic code in Figure 3.4 indicates the checks that were inserted automatically by Deputy based on the programmer-supplied annotations. When applying arithmetic to `buf`, we verify that `buf` is not null, since Deputy disallows arithmetic on null pointers. When dereferencing, incrementing, or returning `cur`, we verify that there is at least one element remaining in the array, which is a requirement for all of these operations. In many cases, Deputy's optimizer has removed checks that it determined to be unnecessary. For example, the result of `buf + len` is required to stay in bounds, so Deputy checks that `buf <= buf + len <= buf + len`; however, since `len` is known to be the length of `buf`, Deputy's optimizer has removed this check. In addition, the `cur++` operation requires the same check as the dereference in the previous statement; in this example, Deputy's optimizer has removed the duplicate check.

3.3.1.2 Other C features.

Apart from bounded buffers. Deputy also provides safety for several other C language features that are common sources of type-safety bugs.

Null termination. Deputy provides a `nullterm` annotation that can be used in conjunction with any one of the above bounds annotations. This annotation indicates that the elements *beyond* the upper bound described by the bound annotation are a null-terminated sequence; that is, the bound annotation describes a subset of a larger null-terminated sequence. For example, `count(5) nullterm` indicates that a pointer points to an array of five elements followed by a null-terminated sequence. Note that `count(0) nullterm` indicates that a pointer points to an empty array followed by a null-terminated sequence—that is, it is a standard null-terminated sequence.

The checks inserted for null-terminated sequences are a straightforward extension of the checks for bounds annotations. For example, when dereferencing a null-terminated pointer, we verify only that the pointer is non-null. When incrementing a null-terminated pointer, we check that the pointer stays within the declared bounds, and if not, we check that it is not incremented past the null element. Note that the `nullterm` annotation can be safely dropped during a cast, but it cannot be safely added to a pointer that is not null-terminated.

Tagged unions. Deputy also provides support for tagged unions. From the perspective of memory safety, C unions are a form of cast, allowing data of one type to be reinterpreted as data of another type if used improperly. C programmers usually use a tag field in the enclosing structure to determine which field of the union is currently in use, but the compiler cannot verify proper use of this tag. As with pointer bounds, Deputy allows the programmer to declare the conditions under which each field of the union is used so that these conditions can be verified at run time when one of the union's fields is accessed. To do so, the programmer adds the annotation "`when(p)`" to each field of the union, where `p` is a predicate that can refer to variable or field names in the immediately enclosing scope, and can use arbitrary side-effect-free arithmetic and logical operators.

Deputy also supports a number of other common C features. For example, Deputy will use format strings to determine the types of the arguments in `printf`-style functions. Also, Deputy allows the program to annotate open arrays (i.e., structures that end with a variable-length array whose length is stored in another field of the structure). Finally, Deputy provides special annotations for functions like `memcpy()` and `memset()`, which require additional checks beyond those used for the core Deputy annotations.

3.3.2 Inferring Annotations

The Deputy type system expects to see a pointer bound annotation on every pointer variable in the program in order to insert checks. However, since adding annotations to every pointer type would be difficult and would clutter the code, Deputy provides a simple inference mechanism. For any local variable of pointer type and for any cast to a pointer type, Deputy will insert two new variables that explicitly hold the bounds for this type. The type can then be automatically annotated with `bound(b, e)`, where `b` and `e` are the two new variables. Whenever this variable is updated, Deputy inserts code to update `b` and `e` to hold the bounds of the right-hand side of the assignment. Although this instrumentation inserts many additional assignments and variables, the trivial ones will be eliminated by the Deputy-specific optimizer.

Deputy also employs the inference algorithm from CCured [Necula *et al.*, 2005] to avoid inserting unnecessary local variables. For example, if a local variable is incremented but not decremented, we insert a new variable for the upper bound only. This inference algorithm is also useful for inferring `nullterm` annotations.

The Deputy compiler works by compiling each C file separately, just as a normal C compiler does. Therefore for pointer types other than those found in local variables or casts, Deputy requires the programmer to insert an annotation. Such pointer types include function prototypes, structure fields, and global variables. Since Deputy cannot instrument external function arguments or structures fields the same way it can instrument local variables, the programmer must explicitly supply bounds annotations. Fortunately, most of these annotations appear in header files that are shared among many compilation units, which means that each additional annotation will benefit a large number of modules in the program. Also, Deputy can optionally use a default value for unannotated pointers in the interface (usually `safe`); however, such annotations are unreliable and should eventually be replaced by an explicit annotation. The inference algorithm assists in this process as well by identifying global variables, functions, and structure fields that require explicit annotations.

3.4 Applying Deputy to the Linux Kernel

The last section discussed the annotations and inferences supported by Deputy. In this section, we discuss how we applied Deputy to the Linux kernel to provide type safety to device drivers in the SafeDrive system.

3.4.1 Adding Annotations

Deputy annotations are added to the kernel API interface exported to the drivers, and in the drivers themselves. The basic process is like this.

- First we try compiling the unannotated device driver with Deputy. Note that at this time some kernel headers used by the driver may already be annotated because they are used by other drivers we have already annotated. But there may be other non-annotated pointers that actually need non-default annotations. The compiler looks for these cases and issues warnings. For example, indexing into unannotated buffers (like `*(p+2)`) will trigger a warning.

- We then go examine the code and see if we can come up with legitimate bounds and other annotations for the pointers. For data structures defined by the kernel, we modify the kernel header files. In order to maintain binary compatibility, we can only add annotations and use other existing variables as bounds, but are not allowed to add new members to kernel data structures. This is possible most of the time. For the few cases where precise annotations are not possible, we add conservative bounds or just trust the operations. For data structures defined by the driver, we modify the drivers source or header files. Here we are allowed to change the definition of driver-only data structures, although we still use annotations most of the time.

- When relevant compile-time warnings are all fixed, we try loading the Deputy-enabled driver in the kernel. Sometimes there are annotations missing (or wrong) that the compile-time checks did not notice. They often trigger spurious runtime errors. We fix these until they are also gone, or find actual problems in the driver.

```
size_t strlcpy(
      char * count(size-1) nullterm dst,
      const char * count(0) nullterm src,
      size_t size);
```

Figure 3.5: Deputy annotations for the prototype of `strlcpy()`

In Section 3.6.3 we will present a breakdown of different types of annotations we actually added, and estimates the amount of work involved. Now let us discuss a few examples of annotated code to see how these annotations are used for actual code.

Figure 3.5 is the annotated prototype of the standard `strlcpy` string manipulation function. It illustrates the use of bounds and nullterm annotations on kernel function prototypes. The argument `dst` is annotated as `char * count(size-1) nullterm`, meaning that it has an least `size-1` bytes of real data followed by a null-terminated area (typically just a zero byte).[1] The `src` argument is `const char * count(0) nullterm`, which just means that it is a standard null-terminated string with unknown minimum length. These annotations demonstrate the flexibility of the `nullterm` annotation, since Deputy is capable of describing both a standard null-terminated pointer as well as a null-terminated array with some known minimum size.

Figure 3.6 shows some annotations used by the Linux device driver `e1000`, illustrating the use of tagged unions and dependence upon other fields in the same structure. In this example, the `type` field indicates which field of the union is currently selected. When using Deputy, the programmer can place the `when` annotations to indicate the correspondence between the tag and the choice of union field so that it can be checked when the union is accessed. For example, when reading the field `range`, Deputy will check that `type == range_option`. When the user wishes to change the tag field (or any field on which the `when` predicates depend), Deputy

[1]This requirement is slightly different from the official `strlcpy` specification, since `dst` is required to be null-terminated on entry as well as on exit. However, in practice, establishing this invariant on entry should not require much, if any, additional effort on the part of client code.

```
struct e1000_option {
  enum {range_option, list_option} type;
  union {
    struct {
        int min, max;
    } range when(type == range_option);
    struct {
        int nr;
        struct e1000_opt_list {...} *p;
    } list  when(type == list_option);
  } arg;
};
```

Figure 3.6: Example usage of the `when` annotation, adapted from Linux's `e1000` driver.

verifies that pointers in the newly selected union field are null and therefore valid.

There are two important restrictions that Deputy imposes on the use of tagged unions. First, they must be embedded in structures that contain one or more fields that can be used as tags for the union. Second, one cannot take the address of a union field, although it is possible to take the address of the structure in which it is embedded.

3.4.2 Caveats and Deputy Limitations

The most significant safety issue that is ignored by Deputy currently is the issue of memory deallocation. Deputy is designed to detect safety violations due to bounds violations, but it does not check for dangling pointers, since detecting such violations would require more extensive run-time checking or run-time support than we currently provide. For now we simply trust that the program's deallocation behavior is correct. We could potentially use a conservative garbage collector.

In addition, there are some cases in which Deputy's type system is insufficient. First, Deputy's type system is sometimes incapable of expressing an existing dependency; for example, programmers sometimes store the length of an array in a structure that is separate from the array itself. Second, Deputy's type system sometimes fails to understand a type cast where there are significant differences between the pointer's base type before and after the cast. Based on our experience (see Section 3.6), such casts occur at about 1% of the assignments. In both of these cases, we use Deputy's trusted cast mechanism to suppress any Deputy errors or run-time checks for a particular cast or assignment. For example, the `container_of` macro, which subtracts from a pointer to a field to get a pointer to the containing object, is not typable in Deputy and we have to resort to trusted casts in the macro definition for now. This means that we are not able to check that this macro is used properly.

In particular, pointers of `void*` type sometimes pose challenges. Downcast from `void *` to a specific type is currently unsound if the type contains pointers. This is used a lot for storing private data of device drivers. So trusted casts need to be used. In some of the cases, we can annotate the `void*` pointer as `POLY`, which means it corresponds to one particular type in this driver and Deputy should assume it will not be changed outside the driver to something of other types. Although this is unsound to be exact, it saves a lot of trusted casts while does not produce many false negatives.

3.4.3 API Changes for Soundness

Although some cases where trusted casts are needed are due to Deputy limitations, there are many of them where a failure in Deputy's type system indicates a lack of robustness in the code itself. Thus, the process of annotating code can help identify places where the interface between modules can be improved. Here are a few examples of these.

We encountered a few cases where the required bounds information is not readily available or not expressible in our type system. Although currently we use trusted code to work around these problems, we could introduce kernel API changes so that soundness can be ensured. For example, the ipw2100 driver defines a number of functions of type `iw_handler`, which are called by the kernel itself.

```
int (*iw_handler)(struct net_device *dev,
                  struct iw_request_info *info,
                  union iwreq_data *wrqu,
                  char *extra)
```

The `extra` parameter points to an array of characters whose bounds are determined by various fields of `wrqu`, depending on the particular `iw_handler` being invoked. We believe that this API should be changed so that the length of `extra` is passed as a fifth parameter, allowing easy verification of this function. In fact, it is quite easy for the clients of this function to pass the appropriate parameter; for example, the following code from `wireless.c` shows that the length of `extra` is readily available at the call site:

```
   ...
   extra_size = descr->max_tokens * descr->token_size;
   extra = kmalloc(extra_size, GFP_KERNEL);
   handler(dev, &info, &(iwr->u), extra);
   ...
```

Overall, this API change required the modification of roughly 10 lines in kernel source and header files, along with 1 find/replace command on the ipw2100 driver source that modified about 40 lines. We believe that such API changes are useful even in the absence of a tool like SafeDrive, since they help the programmer to reason about safety at the time the code is written.

3.5 Recovery System

This section describes how SafeDrive tracks updates from the driver and recovers from driver failures. We also compare our recovery mechanisms to that of Nooks.

3.5.1 Update Tracking

As mentioned briefly in Section 3.2, the update tracking module maintains a linked list of all updates a driver has made to kernel state that should be undone if the driver fails. For each update, the list stores a *compensation operation* [Weimer and Necula, 2004], which is a pointer to a function that can undo the original update, along with any data needed by this compensating action. For example, in Linux device drivers, the compensation function for `kmalloc()` is `kfree()`. This list is also indexed by a hash table, which allows compensations to be removed from the list if the driver manually reverses the update (e.g., if an allocated block is freed). SafeDrive provides wrappers for all functions in the kernel API that require update tracking, allowing drivers to use this feature with minimal changes to their source code.

In a few cases, we need to modify the kernel to handle changes to the list of updates that are not explicitly performed by the driver. For example, timers are removed from the list after the corresponding timer function executes.

Updates recorded by the tracking module are divided into two separate pools, one associated with the driver itself (long-term updates) and the other associated with the current CPU and control path (short-term updates). The latter pool holds updates like spinlock-acquires, which have state associated with the local CPU and must be undone atomically (without blocking) on the same CPU.

3.5.2 Failure Handling

Failures are detected by the run-time checks inserted by the Deputy compiler. When a run-time checks fails, it invokes the SafeDrive recovery system. First, a description of the error is printed for debugging purposes. For problems due to memory safety bugs, this error report pinpoints the actual location where a pointer leaves its designated bounds or is dereferenced inappropriately.

Then SafeDrive goes through a series of steps to clean up the driver module itself, restoring kernel state and optionally restarting the driver, while at the same time allowing other parts of the system to continue running. We maintain two invariants during the recovery process, both vital to the success of recovery:

- *Invariant 1: No driver code is executed from the point when a failure is detected until recovery is complete.* This invariant is required because the driver is already corrupt; executing driver code could easily trigger more failures or corrupt kernel state.

- *Invariant 2: No kernel function is forcefully stopped and returned early.* Forceful returns from kernel functions would corrupt kernel state. On the other hand, the driver function that fails is always stopped forcefully, in order to maintain the first invariant.

3.5.2.1 Returning gracefully from a failed driver.

The basic mechanism for "forceful return" from a driver function is a `setjmp()/longjump()` variant that unwinds the stack and jumps directly to the next instruction after `setjmp()`. SafeDrive requires the programmer to identify driver entry points, and add calls to SafeDrive macros at these entry points to generate wrapper stubs that call `setjmp()` to store the context in the current task structure (Linux's kernel thread data structure). When a failure occurs, the failure-handling code will call `longjmp()` to return to the wrapper, which then returns control to the kernel with a pre-specified return value, often indicating a temporary error or busy condition. The failure-handling code also sets a flag that indicates that the driver is in the "failed" state. This flag is checked at each wrapper stub to ensure that any future calls to the driver will return immediately, thus preserving Invariant 1. This approach allows the kernel to continue normally when the driver fails.

Typically, identifying driver entry points is a simple task; to a first approximation, we can look for all driver functions whose address is taken. Determining the appropriate return value for a failure can be more difficult, since returning an inappropriate value can cause the kernel to hang. Fortunately, the appropriate return value is relatively consistent across each class of drivers in Linux. The Nooks authors provide a more extensive study of possible strategies for selecting proper return values [Swift *et al.*, 2004].

The recovery process is more complicated in cases where the driver calls back into the kernel, which then again calls back into the driver, resulting in a stack that contains interleaved kernel and driver stack frames. If we jump to the initial driver entry point, we skip important kernel code in the interleaved stack frame, violating Invariant 2. Conversely, if we jump to the closest kernel stack frame, we must ensure that Invariant 1 is maintained when we return to the failed driver. To solve this problem, SafeDrive records context information at each re-entry point into the driver. A counter tracks the level of re-entries into the driver, which is incremented whenever the driver calls a kernel function that *may* call back into the driver. After the driver fails, control jumps directly to re-entry points when it returns from these kernel functions. Essentially, we finish executing any kernel code still on the stack while skipping any driver code still on the stack. This technique is similar to the handling of domain termination in Lightweight RPCs [Bershad *et al.*, 1989], although in our case, both the kernel and the module run in the same protection domain.

3.5.2.2 Restoring kernel state and restarting driver.

During the recovery process, all updates associated with the failed driver are undone in LIFO order by calling the stored compensation functions. These compensations undo all state changes the driver has made to the kernel so far, similar to exception handling in languages like C++ and Java. The main difference between our approach and C++/Java exceptions is that the compensation code does not contain any code from the extension itself, thus preserving Invariant 1. As a result, the extension will not have an opportunity to restore any local invariants; however, because the extension will be completely restarted, we are only concerned with restoring the appropriate kernel invariants. Note that this unwinding task is complicated by the fact that it is executed in parallel with other kernel code and by the fact that the failure could have happened in inconvenient places, such as an interrupt handler or timer callback. Thus, after CPU-local compensations such as lock releases are undone in the current context, all other compensations are deferred to be released in a separate kernel thread (in `event/*`).

After compensations have been performed, the driver's module is unloaded. As mentioned above, we do not call the driver's deinitialization function; however, because we track all state-changing functions provided by the kernel, including any function that registers new devices or other services, calling the driver's deinitialization function should not be necessary. After this process is complete, depending on user settings, the driver can be restarted automatically from a clean slate. The restart process follows the normal module initialization process.

The current SafeDrive update tracking and recovery code is a patch to the Linux kernel changing 1084 lines, tracking in total 21 types of Linux kernel resources for the recovery of four drivers in three classes: two network card drivers, one sound card driver, and one USB device driver (see Section 3.6 for details). The resources tracked include 4 types of locks, 4 PCI-related resources, and 4 network-related resources, among others.

3.5.3 Discussion

The recovery system of SafeDrive resembles that of Nooks [Swift *et al.*, 2003] in principle. Both track different types of kernel updates and undo them at failure time. However, the fact that all code using SafeDrive runs in the same protection domain makes SafeDrive's recovery system significantly simpler and less intrusive. In fact, the SafeDrive kernel patch is less than one tenth the size of Nooks'. Update-tracking wrappers and compensation functions are simpler mainly because there is no need for code to copy objects in and out of drivers, to manage the life cycles of separate protection domains, or to perform cross-domain calls. In addition, implementing the cross-domain calls efficiently can complicate the system design significantly. For example, Nooks uses complicated page table tricks to enable fast writes of large amount of data from the device driver to the kernel. Nooks also gives each domain its own memory allocation pool, which requires even more changes to the kernel. We believe that simpler recovery code adds significantly to the trustworthiness of the whole mechanism since testing is less effective for such code than for the normal execution path.

The fact that SafeDrive lets drivers modify kernel data structures directly has some ramifications for kernel consistency, which contrasts with Nooks' call-by-value-result object passing. For each driver call, Nooks first copies all objects the driver may modify, then lets the driver work on the copies, and finally copies the results back. This approach provides atomic updates of kernel objects and thus should provide more consistency. However, we have found there are caveats to this approach. First, it has problems on SMP systems because other processors see the updates to kernel data structures at a different time than they would ordinarily see the updates; the original Nooks paper suggests that SMP is not supported yet [Swift *et al.*, 2003, end of Sec. 4.1]. Second, many kernel data structure updates are not done through this mechanism but through cross-domain calls to kernel functions. The interaction of these two mechanisms becomes complicated quickly.

Our recovery scheme is orthogonal to the Deputy type system and can also work with other languages, including type-safe languages such as Java. As far as we know, current type-safe languages and runtimes do not provide a general mechanism for recovering buggy extensions. Our technique for gracefully returning from a failed extension should apply here. More broadly, a recovery module based on compensation stacks [Weimer and Necula, 2004] would be a valuable asset for these systems.

State and session restoration for failed drivers is not yet implemented in our prototype. Thus, the driver will be in an initial empty state after recovery. We believe the shadow driver approach proposed by the Nooks project [Swift *et al.*, 2004] should apply to our system. Its implementation should also be simpler because of the absence of multiple hardware protection domains.

3.6 Evaluation

In this section, the recovery mechanism is first evaluated in terms of successful recoveries in the face of randomly injected faults with two experiments. Then we measure two distinct types of overhead of using SafeDrive: (1) the one-time overhead of annotating APIs and adding wrapper functions to a particular extension, and (2) runtime overhead due to additional checks inserted by the Deputy type system and due to update tracking for recovery. We quantify the first one by noting how much of the kernel API and wrappers needed alteration to support a handful of drivers that we chose

Driver	Description
e1000	Intel PRO/1000 network card driver
tg3	Broadcom Tigon3 network card driver
usb-storage	USB mass-storage driver
intel8x0	Intel 8x0 builtin sound driver
emu10k1	Creative Audigy 2 sound driver
nvidia	NVidia video driver

Table 3.1: Drivers used in SafeDrive experiments

Fault Category	**Code Transformation**
Loop fault	Make loop count larger by: 1 with 0.5 prob, 2–1024 with 0.44 prob., and 2K–4K with 0.06 prob.
Scan overrun	Make size parameter to `memset`-like funcs larger as the line above
Off-by-one	Change < to <=, etc., in boolean expressions
Flipped condition	Negate conditions in if statements
Missing assignment	Remove assignments and initialization of local vars
Corrupt parameter	Replace a pointer parameter with null, or a numeric parameter with a random number
Missing call	Remove calls to funcs and return a random result as the line above

Table 3.2: Categories of faults injected in SafeDrive recovery experiments

from different subsystems of the kernel. We quantify the second source of overhead with traditional performance benchmarks.

Our experiments are done with six device drivers in four categories for the Linux 2.6.15.5 kernel, as shown in Table 3.1. All drivers are annotated with Deputy annotations to detect type-safety errors. However, due to time constraints, we only added update tracking and recovery support to the four drivers with names shown in bold.

3.6.1 Error Detection and Recovery Rate

Here we evaluate how well SafeDrive is able to handle various faults in drivers. We use *compile-time software-implemented fault injection* to inject random faults into the driver, and then we run the driver with and without SafeDrive. We wrote a compile-time fault injection tool as an optional phase in the Deputy compiler. The tool injects

	Correct		**Incorrect**	
SafeDrive	Works	Innocuous Errors	Crashes	Mal-functions
Off	75	*n/a*	44	21
On	113	8	0	19

Table 3.3: Results of 140 runs of `e1000` with injected faults, with SafeDrive off and on. "Correct" means that the driver behaved as expected, possibly with the help of SafeDrive's recovery subsystem and assuming the bugs are transient.

into C code seven categories of faults, shown in Table 3.2, following two empirical studies on kernel code [Christmansson and Chillarege, 1996; Sullivan and Chillarege, 1991]. We did not use an existing binary-level fault injection tool, such as that used by the Rio file cache [Ng and Chen, 1999] or Nooks [Swift *et al.*, 2005], because all of Deputy's checks are inserted at compilation time, which means that Deputy does not have a chance to catch errors introduced via binary fault injection. Compile-time fault injection allows us to evaluate Deputy's error detection fairly, and it also has the benefit of being able to introduce more realistic programming errors.

The fault-injection experiments were done with `e1000` driver. For each experiment, 5 random faults of the same category were inserted into the driver code during the compilation process. A script exercised the driver by loading it, configuring networking, downloading a large (89MB) file, checksumming the file, and finally unloading the driver. Then the same script was run with the unmodified `e1000` driver to check whether the system was still functioning. This test was performed with and without SafeDrive recovery on. When SafeDrive recovery was off, Deputy checks were still performed, but they did not trigger any action when failures were detected. Thus the driver should have behaved exactly the same as the original one with faults injected.

Table 3.3 shows the results of all 140 runs, with 20 runs per fault category. When SafeDrive is disabled, 44 of these runs resulted in crashes and 21 resulted in the driver

Detection	Crashes	Malfunc.	Innocuous	Total
Static	10	0	3	13 (24%)
Dynamic	34	2	5	41 (76%)
Total	44	2	8	54

Table 3.4: Breakdown for the 54 cases in which SafeDrive detected errors

malfunctioning. When SafeDrive was enabled, SafeDrive successfully prevented all 44 crashes, and it invoked the recovery subsystem on 2 of the 21 non-crash driver malfunctions. In addition, there were 8 runs where SafeDrive successfully detected apparently innocuous type-safety errors that did not trigger crashes or malfunctions with SafeDrive disabled, but failed Deputy checks when SafeDrive was on.

A closer look at the results reveals that, among the 7 categories of faults injected, all categories except “off-by-one” and “flipped condition” resulted in crashes when SafeDrive was off. The fact that SafeDrive prevented these crashes indicates that these crashes were actually due to type-safety errors, not other serious errors such as incorrect interrupt commands. On the other hand, the malfunctioning runs were due to a variety of reasons, including setting hardware registers to bad values and incorrect return value checking. Type-safety checks alone cannot always detect these errors.

Overall, SafeDrive detected problems in 54 of the runs. Table 3.4 shows that 24% of the problems were detected statically by the Deputy compiler. These are errors that were *not* detected by GCC, and in fact 10 of them led to crashes. These compile-time errors include passing an incorrect constant size argument to `memcpy` and dereferencing an uninitialized pointer, among others.

Of the innocuous errors, five were detected dynamically, causing SafeDrive to invoke recovery and successfully restart the driver. Although these errors appear to be innocuous, they are likely latent bugs; thus, invoking recovery seems to be a reasonable response. Of course, SafeDrive has no way to know which errors will be truly benign.

These results show that SafeDrive is effective in detecting and recovering from typical memory and type-safety errors in drivers. In addition, a significant portion of these errors are caught at compile time, before the driver is even run. Finally, it seems unlikely that SafeDrive will always prevent *all* crashes, as it did with the `e1000`, due to the limits of type checking.

3.6.2 Another Recovery Rate Test

We did another set of experiments to futher test the recovery ability of SafeDrive for failures happening at any location in device drivers. We use a simpler form of fault injection this time: We alter Deputy's checks so that they fail after a random number N of checks are executed. By choosing a large and random N, we induce failure at a given check with a probability proportional to the frequency with which that check is executed. We use a script to repeatedly inject failure to the e1000 driver and reconfigure the network card after it recovers, as we do not do session restoration in kernel yet. At the same time, we test network connectivity by logging in from another machine to the test machine through ssh. Note that Linux preserves TCP connections through network card reconfiguration, so a single ssh session persists through recoveries if they do not fail.

We injected 50 faults in total. In 48 of these the ssh session is preserved, while in the other 2 the machine crashed. Both crashing faults (and 3/4 of all faults) are in interrupts handlers. We were not yet able to find out the exact reasons for the two crashes. Suspects are buggy wrappers, or some untracked resources.

Although this test does not introduce actual bugs into the driver, it still shows that SafeDrive can successfully recover from a large portion of memory-safety faults at random positions in the driver, allowing the kernel to continue its work.

3.6.3 Annotation Burden

Table 3.5 shows the number of lines of code we changed in order to use SafeDrive on these drivers. The third column shows all Deputy-related changes, and the next four columns show the number of lines containing each type of Deputy annotation. These numbers do not add up to the previous number because not every changed line contains a Deputy annotation. These changes are essential to driver safety, and they amount to approximately 1–4% of the total number of lines in a given driver. The last column shows additional recovery-related changes currently needed in the driver code by the SafeDrive prototype. These changes are boilerplate code placed at driver entry points in the driver source files. These code should be easy to generate with an automated tool, making the effort needed minimal.

The other set of changes required are changes to the kernel headers. This set of changes includes Deputy annotations for the kernel API as well as wrappers for functions that are tracked by the SafeDrive runtime system. Both types of changes must be written by hand; however, these changes are a one-time cost for drivers of a given class. For the 4 classes of drivers we worked on, a total of 1,866 lines in 112 header files were changed. Casual inspection reveals that about half of the lines are Deputy-related and that the rest are recovery-related. In particular, there are 187 lines with bounds annotations, 260 lines with nullterm, 8 lines with tagged unions, and 140 lines with trusted code annotations.

Driver	Original LOC	Deputy Changes					Recovery Changes
		LOC Modified	Bounds	Nullterm	Tagged Unions	Trusted Code	
e1000	17011	260	146	15	2	47	270
tg3	13270	359	78	9	0	64	156
usb-storage	13252	136	16	11	0	21	118
intel8x0	2897	124	31	2	0	8	167
emu10k1	11080	441	66	11	0	23	n/a
nvidia	10126	224	42	35	0	27	n/a

Table 3.5: Number of lines changed in order to enable SafeDrive for each driver. All numbers are lines of code. "LOC Modified" in "Deputy Changes" are number of lines with Deputy annotations and related changes. The next four columns show numbers of lines with each category of annotations. "Recovery Changes" shows the number of lines where trivial wrappers were added for recovery.

Benchmark	Driver	Native Throughput	SafeDrive Throughput	Native CPU %	SafeDrive CPU %
TCP Receive	e1000	936Mb/s	936Mb/s	47.2	49.1 (+4%)
UDP Receive	e1000	20.9Mb/s	17.4Mb/s (-17%)	50.0	50.0
TCP Send	e1000	936Mb/s	936Mb/s	20.1	22.5 (+12%)
UDP Send	e1000	33.7Mb/s	30.0Mb/s (-11%)	45.5	50.0 (+9%)
TCP Receive	tg3	917Mb/s	905Mb/s (-1.3%)	25.4	27.4 (+8%)
TCP Send	tg3	913Mb/s	903Mb/s (-1.1%)	18.0	20.4 (+13%)
Untar	usb-storage	1.64MB/s	1.64MB/s	5.5	6.8 (+23%)
Aplay	emu10k1	n/a	n/a	9.10	9.64 (+6%)
Aplay	intel8x0	n/a	n/a	3.79	4.33 (+14%)
Xinit	nvidia	n/a	n/a	12.13	12.59 (+4%)

Table 3.6: Benchmarks measuring SafeDrive overhead. Utilization numbers are kernel CPU utilization.

3.6.4 Performance

The run-time overhead of SafeDrive is composed of several parts, including run-time checks inserted by the Deputy compiler, update tracking cost, and context saving cost. We measure the performance overhead of SafeDrive by running several benchmarks with native and SafeDrive-enabled drivers.

Table 3.6 shows results for these benchmarks on a dual Xeon 2.4Ghz. In "TCP Receive", `netperf` [Jones, 1995] is run on another host and sends TCP streaming data to the testing server (the `TCP_STREAM` test). The socket buffers are 256KB on both the sending and receiving side, and 32KB messages are sent. "TCP Send" works the other way around, with the testing server running `netperf` and sending traffic. In "UDP Receive" and "UDP Send", UDP packets of 16 bytes are received/sent (the `UDP_STREAM` test). CPU utilization is measured with the `sar` utility. CPU utilization maxes out at 50%, probably because the driver cannot utilize more than one CPU. The UDP tests show higher overhead probably due to two reasons. First, the packets are much smaller, leading to more Deputy overhead overall. Second, less other kernel code is involved in UDP processing compared to TCP, amplifying SafeDrive's overhead.

We tested the `usb-storage` driver with a 256MB Sandisk USB2.0 Flash drive on a Thinkpad T43p laptop (2.13Ghz Pentium-M CPU). The "Untar" benchmark simply untars a Linux 2.2.26 source code tar ball, which is already on the drive, to the drive itself. The tar file is 82MB in size. After untarring finishes, the drive is immediately unmounted to flush any data in the page cache. CPU utilization is the average value over the whole period. As can be seen from the results, the whole operation finishes in the same amount of time, though SafeDrive's instrumentation increased CPU usage by 23%.

We also benchmarked two sound card drivers: the `intel8x0` sound driver for the built-in sound chip in a Thinkpad T41 (1600Mhz Pentium-M CPU), and the `emu10k1` sound driver for a Creative Audigy 2 card on a Pentium II 450Mhz PC. Both benchmarks used the `oprofile` facility to capture how much time (in percentage of total CPU time) was spent in the kernel on behalf of the sound driver while a 30-minute 44.1Khz wave file was playing. `aplay` and the standard `alsa` sound library were used for playback. Throughput is irrelevant here because the sample rate is fixed.

Finally, we tested the open-source portion of the driver distributed by NVidia for their video cards. These 10,296 lines of open-source code are the interface between the kernel and a larger, proprietary graphics module which we did not process because the source code is not available. We tested the `nvidia` driver on a Pentium 4 2.4 GHz machine with a GeForce4 Ti 4200 graphics card. Table 3.6 shows the CPU usage of this driver while setting up and tearing down an X Window session, as measured by `oprofile`. The instrumentation did not have a measurable effect on the performance of the `x11perf` graphics benchmarking tool, which is limited by hardware performance rather than by the driver.

The above results show that SafeDrive has relatively low performance overhead, even for data-intensive drivers. To compare with Nooks, consider the `e1000` TCP send/receive tests. These tests are similar to experiments discussed in the Nooks journal paper [Swift *et al.*, 2005], where the authors reported relative receive and send CPU overheads of 111% and 46% respectively, both with 3% degradation of throughput. This overhead is nearly an order of magnitude higher than the overhead of SafeDrive (rows 1 and 3 in Table 3.6), suggesting that the overhead of cross-domain calls far outweighs the overhead of Deputy's run-time checks for these benchmarks.

3.7 Related Work

3.7.1 Enforcing Isolation with Hardware

Several projects have used hardware to isolate device drivers from the rest of the system and to allow recovery in the case of failure.

The Nooks project [Swift *et al.*, 2004; Swift *et al.*, 2003] isolates device drivers from the main kernel by placing them in separate hardware protection domains called "nooks". These protection domains share the same address space but have different permission settings for pages. A driver is permitted to read all kernel data but only allowed to write to certain pages. Cross-domain calls replace function calls between the driver and the kernel, although the semantics of the calls remain mostly similar.

Virtual Machine Monitors (VMMs) such as L4 [LeVasseur *et al.*, 2004] and Xen [Barham *et al.*, 2003; Fraser *et al.*, 2004] isolate a driver using hardware protection. However, rather than placing a protection barrier between a driver and the kernel, the VMM runs each driver with its own kernel inside a separate virtual machine. This approach allows device drivers to be used unchanged, and it allows one to use device drivers that depend on different operating systems. Communication between virtual machines is performed using a special-purpose high-performance batched channel interface.

From a pure driver isolation standpoint, SafeDrive is less secure than a VMM since it is possible for a driver to manipulate the kernel via the API calls, privileged instructions, or simply tying up CPU resources by looping. In the VMM case, however, a buggy driver can only corrupt the driver's local, untrusted kernel, which is isolated from the trusted kernel behind a message-passing interface. The SLAM project [Ball *et al.*, 2001] looks at API usage validation, and in theory could close this gap when combined with SafeDrive.

However, this caveat does not mean that SafeDrive is able to catch fewer errors than a VMM. In fact, one advantage of SafeDrive is that its finer-grained checks are able to detect errors within the driver and not just outside it. Although hard to measure, this fine-grained error detection reduces the likelihood of data corruption and is very important in cases involving persistent data. For example, a misbehaving disk driver in any of Xen, Nooks, or SFI can corrupt data on disk before the fault is detected. This exact behavior occurred with Nooks during their fault injection experiments [Swift *et al.*, 2003].

Another advantage of SafeDrive relative to these systems is performance. The additional domain crossings required by the hardware approaches impose additional costs. In all three hardware-based systems, one can generally expect the CPU overhead for data intensive device drivers to be between 40% to 200% [LeVasseur *et al.*, 2004; Menon *et al.*, 2005; Swift *et al.*, 2004; Swift *et al.*, 2005]. This result contrasts with a typical CPU overhead of less than 20% for SafeDrive, which incurs no additional cost for calls into or out of a driver. Of course, it is not guaranteed that SafeDrive will always outperform hardware approaches: if crossings are rare and checks are frequent then a hardware solution is likely to outperform SafeDrive; however, our experiments suggest that SafeDrive performs better in practice and that SafeDrive's performance is likely to improve further as its optimizer improves.

3.7.2 Enforcing Isolation with Binary Instrumentation

Software-enforced Fault Isolation (SFI) [Erlingsson *et al.*, 2006; Small and Seltzer, 1997; Wahbe *et al.*, 1993] instruments extension binaries to ensure that no memory operation can write outside of an extension's designated memory region. The instrumentation can take one of several forms, depending on the desired tradeoff between isolation and performance.

If reads are protected as well as writes (as they are in SafeDrive), then typical performance overhead varies between 17% and 144%, depending on the SFI implementation and the benchmarks being used. Although it is hard to make direct performance comparisons against results obtained from different test programs, we expect SafeDrive to exceed the performance of SFI, since SafeDrive only needs to check memory accesses for which the type checker is unable to verify correctness.

As with the hardware-based approaches, SFI only prevents an extension from corrupting the system, and it does not attempt to prevent a driver from corrupting itself or the device. Furthermore, these approaches require a very clean kernel-driver interface in order to ensure that all data passed between the kernel and the driver can be checked at run time [Erlingsson *et al.*, 2006].

3.7.3 Enforcing Isolation with a Language

A number of research projects have attempted to enforce memory safety in C programs at source level instead of at binary level. CCured [Necula *et al.*, 2005], a predecessor to Deputy, used a whole-program analysis to classify pointers according to their use, and then it altered data structures and code to provide low-cost run-time memory safety checks. Unfortunately, CCured made significant changes to the program's data structures, making it difficult to apply CCured to one module at a time. Another source-level tool is the Cyclone [Jim *et al.*, 2002] language, which is a safe alternative to the C language that has been used to write safe kernel-level code [Anagnostakis *et al.*, 2002]. However, like CCured, Cyclone requires a large amount of manual intervention to port existing drivers when compared to Deputy. Neither CCured nor Cyclone support *a priori* data layouts, which are a prerequisite for extensions with predefined APIs and data structures. Finally, Yong and Horwitz [Yong and Horwitz, 2003] use static analysis to insert efficient buffer overflow checks; however, they do not address memory safety in general, and they provide relatively coarse-grained checks.

The major advantage of the Deputy type system over these other source-level approaches is that Deputy allows the programmer to describe pointer bounds in terms of other variables or fields in the program, and thus Deputy can leave data layout and APIs unchanged. A related project at Microsoft uses the SAL annotation language and the ESPX modular annotation checker in order to find buffer overflows [Hackett *et al.*, 2005]. Although SAL can describe relationships between variables, it cannot describe relationships between structure fields, and it does not support tagged union types. Also, ESPX is a static analysis, which means that problematic code is simply left unverified; in contrast, Deputy inserts run-time checks where static analysis is insufficient.

There are also a number of related projects that allow types to refer to program data, including the Xanadu language [Xi, 2000] and Hickey's very dependent function types [Hickey, 1996]. However, these projects have a number of restrictions on mutable data, which Deputy addresses using run-time checks. Also, Harren and Necula [Harren and Necula, 2005] developed a similar framework for assembly language in which dependencies can occur between registers or between structure fields.

Type qualifiers represent another area of related work. CQual [Foster *et al.*, 1999; Johnson and Wagner, 2004] allows programmers to add custom type qualifiers such as `const` or `user/kernel`, using an inference algorithm to propagate these qualifiers throughout the program. Semantic type qualifiers [Chin *et al.*, 2005] builds on this work by allowing qualifiers to be proved sound in isolation. Compared to this work, Deputy's annotations are more expressive (since annotations can refer to other program data) and correspondingly more difficult to infer. Although Deputy provides a number of features for inferring or guessing annotations, it relies more heavily on programmer-supplied annotations than these other type qualifier tools. Finally, Privtrans [Brumley and Song, 2004] allows the programmer to specify privileged operations and automatically separates a program into a privileged process and non-privileged process, improving security.

There are operating systems built mainly with type-safe languages, such as Singularity [Hunt *et al.*, 2005], JavaOS [Mitchell, 1996], SPIN [Bershad *et al.*, 1995], and the Lisp Machine OS [Stallman *et al.*, 1981]. These operating systems naturally have few memory-safety problems, and as processor speed increases, the performance penalty of these languages become less of a concern. While this approach will be important in building future operating systems, we believe that current commodity operating systems such as Windows and Linux, which are written mainly in C and C++, will be in wide use for many years. SafeDrive will be useful both in improving the reliability of these existing operating systems and in providing a transition to future type-safe operating systems.

3.7.4 Fault Tolerance for Applications

In addition to the issue of how a device driver should be isolated from the rest of the system, there is also the largely orthogonal issue of how the system should recover when a driver failure is detected. Both the original Nooks paper [Swift *et al.*, 2003] and SafeDrive provide isolation, release of resources, and restart of the driver.

A later Nooks paper [Swift *et al.*, 2004] showed how to use *shadow drivers* to restore the session state of applications that were using the driver. We expect the shadow driver technique to work without significant modification with SafeDrive, and thus we do not address this issue further.

Vino [Seltzer *et al.*, 1996], which isolates drivers using SFI, executes each driver request inside a transaction that can be aborted and retried on failure. Another related technique is Microreboot [Candea *et al.*, 2004], which is used to build rebootable components in large enterprise systems. Session restoration is achieved by programming the components against a separate session state storage that persists over component restarts.

3.8 Conclusion

We have presented SafeDrive, a system that uses language-based techniques to detect type safety errors and to recover from such errors in device drivers written in C. The checking in SafeDrive is fine-grained, which is critical because it not only protects the kernel from misbehaving drivers, but also helps prevent the driver from corrupting persistent data or kernel state. SafeDrive requires few changes to the kernel or drivers, and experiments show that SafeDrive incurs low overhead (normally less than 20%) and successfully prevents all 44 crashes due to randomly injected errors for one driver. Overall, we hope that this work shows that we can achieve the safety of high-level, type-safe languages without abandoning existing C code.

Chapter 4

Conclusion

Dependability of commodity operating systems is becoming increasingly vital. This work presents two case studies of applying program analysis to the issue of improving dependability of commodity operating systems. We focus on techniques that can scale to large code bases and are sound in most cases.

First, we presented a domain-specific static analysis tool called Ctxcheck that analyzes the Linux kernel to understand execution contexts and spin locks, two error-prone aspects of the Linux kernel. Then the tool finds related bugs according to a set of rules. The tool employs a flow-sensitive, inter-procedural and context-insensitive analysis that tracks the context state at all locations in the kernel. To achieve enough precision and scalability, we designed a set of annotations to help the analysis. In our evaluation, Ctxcheck successfully found 6 real bugs in Linux 2.6.20.7 while requiring relatively small amount of annotations (about 1000 annotations, changing 0.2% of kernel code lines). And the analysis runs for less than 2 minutes.

Second, we presented SafeDrive, a system enabling type-safe and recoverable device drivers for Linux, without requiring rewriting the existing drivers. SafeDrive employs both static analysis and runtime instrumentation to ensure fine-grained type-safety and recoverability for drivers written in C. SafeDrive uses the Deputy dependent type system, which encodes bounds and other type-safety information with dependent type annotations. Then a runtime recovery system in the kernel tracks operations done by the driver, intercepts any failure in drivers and automatically recovers the system by cleaning up related kernel/driver state and returning the system to a consistent state. Evaluation shows that the number of annotations needed is moderate, at about 1-4% of driver code. Runtime overhead in micro-benchmarks is normally less than 20% and one order of magnitude lower than previous hardware protection-based approaches. And SafeDrive is effective in preventing crashes due to type-safety errors. It successfully recovered from all 44 crashes due to injected faults in one experiment, and 48 of 50 tests in another.

We believe this work represents a first step in the direction of solving the dependability challenge that operating system designers face today. This challenge is posed by ongoing trends such as increasingly complex devices and applications, more parallel hardware architectures and an ever-more networked and hostile environment. The overall approach can be characterized by a few principles: employing powerful sound analyses, using a small number of carefully-designed annotations, combining static and runtime techniques, and exploiting domain knowledge. We hope ultimately this approach, or combined with others, could enable operating system designers to find defects early in the development process, and save the huge cost, in both money and customer losses, of fixing the defects in the field by releasing one patch after another as is common practice today.

Bibliography

[Aho *et al.*, 2007] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, & Tools*. Pearson/Addison Wesley, 2rd edition, 2007.

[Anagnostakis *et al.*, 2002] Kostas G. Anagnostakis, Michael Greenwald, Sotiris Ioannidis, and Stefan Miltchev. Open packet monitoring on FLAME: Safety, performance and applications. In *Proceedings of the 4rd International Working Conference on Active Networks (IWAN 2002)*, 2002.

[Anonymous, 2007] Anonymous. Source lines of code. *Wikipedia article: http://en.wikipedia.org/wiki/Source_lines_of_code*, 2007.

[Asanovic *et al.*, 2006] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[Ball *et al.*, 2001] Thomas Ball, Rupak Majumdar, Todd D. Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *SIGPLAN Conference on Programming Language Design and Implementation*, 2001.

[Ball *et al.*, 2006] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. Thorough static analysis of device drivers. In *EuroSys '06: Proceedings of the 2006 EuroSys conference*, pages 73–85, New York, NY, USA, 2006. ACM Press.

[Barham *et al.*, 2003] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, 2003.

[Bershad *et al.*, 1989] Brian Bershad, Thomas Anderson, Edward Lazowska, and Henry Levy. Lightweight remote procedure call. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, 1989.

[Bershad *et al.*, 1995] Brian N. Bershad, Stefan Savage, Przemyslaw Pardyak, Emin G. Sirer, Marc Fiuczynski, David Becker, Susan Eggers, and Craig Chambers. Extensibility, safety and performance in the SPIN operating system. In *15th Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain, Colorado, 1995.

[Bligh *et al.*, 2007] Martin Bligh, Mathieu Desnoyers, and Rebecca Schultz. Linux kernel debugging on Google-sized clusters. In *Proceedings of the Linux Symposium*, Ottawa, Ontario, Canada, 2007.

[Bos and Samwel, 2002] Herbert Bos and Bart Samwel. Safe kernel programming in the OKE. In *Open Architectures and Network Programming Proceedings*, 2002.

[Bovet and Cesati, 2005] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly & Associates, Inc., 3rd edition, 2005.

[Brumley and Song, 2004] David Brumley and Dawn Song. Privtrans: Automatically partitioning programs for privilege separation. In *Proceedings of the 13th USENIX Security Symposium*, 2004.

[Candea *et al.*, 2004] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. Microreboot — a technique for cheap recovery. In *Symposium on Operating System Design and Implementation*, 2004.

[Chaki *et al.*, 2004] Sagar Chaki, Edmund Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular verification of software components in C. *IEEE Transactions on Software Engineering (TSE)*, 30(6):388–402, June 2004.

[Chin *et al.*, 2005] Brian Chin, Shane Markstrum, and Todd Millstein. Semantic type qualifiers. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 2005.

[Chou *et al.*, 2001] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating system errors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, 2001.

[Christmansson and Chillarege, 1996] Jorgen Christmansson and Ram Chillarege. Generation of an error set that emulates software faults — based on field data. In *Proceedings of the 26th IEEE International Symposium on Fault Tolerant Computing*, 1996.

[Compaq *et al.*, 2001] Compaq, Intel, Microsoft, Phoenix, and Toshiba. ACPI: Advanced configuration and power interface, 2001.

[Condit *et al.*, 2007] Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, and George Necula. Dependent types for low-level programming. In *16th European Symposium on Programming*, Braga, Portugal, 2007.

[Das, 2000] Manuvir Das. Unification-based pointer analysis with directional assignments. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 35–46, New York, NY, USA, 2000. ACM Press.

[Erlingsson *et al.*, 2006] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. XFI: Software guards for system address spaces. In *Symposium on Operating System Design and Implementation*, 2006.

[Flanagan, 2006] Cormac Flanagan. Hybrid type checking. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 245–256, New York, NY, USA, 2006. ACM Press.

[Foster *et al.*, 1999] Jeffrey S. Foster, Manuel Fahndrich, and Alexander Aiken. A theory of type qualifiers. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 1999.

[Fraser *et al.*, 2004] Keir Fraser, Steven Hand, Rolf Neugebauer, Ian Pratt, Andrew Warfield, and mark Williamson. Safe hardware access with the Xen virtual machine monitor. In *Proceedings of the 1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS 2004)*, 2004.

[Hackett *et al.*, 2005] Brian Hackett, Manuvir Das, Daniel Wang, and Zhe Yang. Modular checking for buffer overflows in the large. Technical Report MSR-TR-2005-139, Microsoft Research, 2005.

[Harren and Necula, 2005] Matthew Harren and George C. Necula. Using dependent types to certify the safety of assembly code. In *Procedings of the 12th international Static Analysis Symposium (SAS)*, 2005.

[Henzinger *et al.*, 2002] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 58–70, New York, NY, USA, 2002. ACM Press.

[Hickey, 1996] Jason Hickey. Formal objects in type theory using very dependent types. In *Proceedings of the 3rd International Workshop on Foundations of Object-Oriented Languages*, 1996.

[Hunt and Larus, 2007] Galen C. Hunt and James R. Larus. Singularity: Rethinking the software stack, April 2007.

[Hunt *et al.*, 2005] Galen Hunt, James R. Larus, Martin Abadi, Mark Aiken, Paul Barham, Manuel Fahndrich, Chris Hawblitzel, Orion Hodson, Steven Levi, Nick Murphy, Bjarne Steensgaard, David Tarditi, Ted Wobber, and Brian D. Zill. An overview of the Singularity project. Technical report, Microsoft Research, 2005.

[Jim *et al.*, 2002] Trevor Jim, J. Gregory Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*, 2002.

[Johnson and Wagner, 2004] Rob Johnson and David Wagner. Finding user/kernel pointer bugs with type inference. In *USENIX Security Symposium*, 2004.

[Jones, 1995] Rick Jones. Netperf: A network performance benchmark, 1995. `http://www.netperf.org`.

[Kroah-Hartman, 2007] Greg Kroah-Hartman. Linux kernel development. In *Proceedings of the Linux Symposium*, Ottawa, Ontario, Canada, 2007.

[LeVasseur *et al.*, 2004] Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Gotz. Unmodified device driver reuse and improved system dependability via virtual machines. In *Symposium on Operating System Design and Implementation*, 2004.

[Menon *et al.*, 2005] Aravind Menon, Jose Renato Santos, Yoshio Turner, G. (John) Janakiraman, and Willy Zwaenepoel. Diagnosing performance overheads in the Xen virtual machine environment. In *Proceeding of the 1st ACM/USENIX Conference on Virtual Execution Environments (VEE 2005)*, 2005.

[Mitchell, 1996] James G. Mitchell. JavaOS: Back to the future (abstract). In *Symposium on Operating System Design and Implementation*, 1996.

[Molnar and van de Ven, 2007] Ingo Molnar and Arjan van de Ven. Runtime locking correctness validator. *Linux kernel documentation*, 2007.

[Necula *et al.*, 2002] George C. Necula, Scott McPeak, and Westley Weimer. CIL: Intermediate language and tools for the analysis of C programs. In *International Conference on Compiler Construction*, 2002.

[Necula *et al.*, 2005] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. CCured: Type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems*, 27(3), 2005.

[Ng and Chen, 1999] Wee Teck Ng and Peter M. Chen. The systematic improvement of fault tolerance in the Rio file cache. In *Symposium on Fault-Tolerant Computing*, 1999.

[NIST, 2002] NIST. The economic impacts of inadequate infrastructure for software testing. Technical Report NIST Planning Report 02-3, NIST, May 2002.

[Patel *et al.*, 2003] Parveen Patel, Andrew Whitaker, David Wetherall, Jay Lepreau, and Tim Stack. Upgrading transport protocols using untrusted mobile code. In *SOSP*, 2003.

[Schmid, 2006] Patrick Schmid. 32-core processors: Intel reaches for (the) sun. *News article from Tom's Hardware*, 2006.

[Schneier, 2003] Bruce Schneier. Blaster and the great blackout. *Salon.com*, December 2003.

[Seltzer *et al.*, 1996] Margo I. Seltzer, Yasuhiro Endo, Christopher Small, and Keith A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Symposium on Operating System Design and Implementation*, 1996.

[Small and Seltzer, 1997] Christopher Small and Margo Seltzer. MiSFIT: A tool for constructing safe extensible C++ systems. In *Proceedings of the 3rd USENIX Conference on Object-Oriented Technologies (COOT 1997)*, 1997.

[Stallman *et al.*, 1981] Richard Stallman, Daniel Weinreb, and David Moon. *The Lisp Machine Manual.* Massachusetts Institute of Technology, 1981.

[Sullivan and Chillarege, 1991] M. Sullivan and R. Chillarege. Software defects and their impact on system availability — a study of field failures in operating systems. In *Proceedings of the 21st International Symposium on Fault-Tolerant Computing*, 1991.

[Sutter, 2005] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), 2005.

[Swift *et al.*, 2003] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, 2003.

[Swift *et al.*, 2004] Michael M. Swift, Muthukaruppan Annamalai, Brian N. Bershad, and Henry M. Levy. Recovering device drivers. In *Symposium on Operating System Design and Implementation*, 2004.

[Swift *et al.*, 2005] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. *ACM Transactions Computer Systems*, 23(1), 2005.

[Swinehart *et al.*, 1986] Daniel C. Swinehart, Polle T. Zellweger, Richard J. Beach, and Robert B. Hagmann. A structural view of the cedar programming environment. *ACM Transactions on Programming Languages and Systems*, 8(4):419–490, 1986.

[Torvalds and Triplett, 2003] Linus Torvalds and Josh Triplett. Sparse project website, 2003. `http://www.kernel.org/pub/software/devel/sparse/`.

[Wahbe *et al.*, 1993] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. *ACM SIGOPS Operating Systems Review*, 27(5), 1993.

[Weimer and Necula, 2004] Westley Weimer and George C. Necula. Finding and preventing run-time error handling mistakes. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2004.

[Xi, 2000] Hongwei Xi. Imperative programming with dependent types. In *Proceedings of 15th IEEE Symposium on Logic in Computer Science*, 2000.

[Yong and Horwitz, 2003] Suan Hsi Yong and Susan Horwitz. Protecting C programs from attacks via invalid pointer dereferences. In *Proceedings of the 10th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2003.

[Zhou *et al.*, 2006] Feng Zhou, Jeremy Condit, Zachary Anderson, Ilya Bagrak, Rob Ennals, Matthew Harren, George Necula, and Eric Brewer. Safedrive: safe and recoverable extensions using language-based techniques. In *Proceedings of the 7th conference on USENIX Symposium on Operating Systems Design and Implementation*, pages 4–4, Berkeley, CA, USA, 2006. USENIX Association.