Beyond Bug-Finding: Sound Program Analysis for Linux

Zachary Anderson,[1] Eric Brewer,[1] Jeremy Condit,[1] Robert Ennals,[2]
David Gay,[2] Matthew Harren,[1] George C. Necula,[1] Feng Zhou[1]

[1] *University of California, Berkeley* [2] *Intel Research Berkeley*

{zra,brewer,jcondit,matth,necula,zf}@cs.berkeley.edu {robert.ennals,david.e.gay}@intel.com

Abstract

It is time for us to focus on *sound analyses* for our critical systems software—that is, we must focus on analyses that ensure the *absence* of defects of particular known types, rather than best-effort bug-finding tools. This paper presents three sample analyses for Linux that are aimed at eliminating bugs relating to type safety, deallocation, and blocking. These analyses rely on lightweight programmer annotations and run-time checks in order to make them practical and scalable. Sound analyses of this sort can check a wide variety of properties and will ultimately yield more reliable code than bug-finding alone.

1 Introduction

The strength of the systems community has long been *optimization*, historically for performance. This “quantitative approach” promotes common metrics and benchmarks for key properties that in turn become metrics of success for new work. But for properties such as security or privacy or safety, mere optimization is not sufficient. For such properties we need *guarantees*, and we believe that in many cases, it is feasible for the programmer and the compiler to provide these guarantees. For example, type checkers for strong type systems routinely prove deep facts about large programs without imposing an unnecessary burden on the programmer or the compiler. Guarantees for many important higher-level system properties can be obtained by similarly practical techniques, even for existing systems.

In the realm of static analysis for systems code, the systems community’s focus on optimization has been manifested as a focus on *bug-finding*, where the metric of performance is the number of bugs found. Although this work has been very successful at revealing specific flaws in existing software, what we really want is a guarantee that no bugs of a specific type can occur—in other words, we want *sound static analysis*.

The systems community has historically felt that soundness requires too much programmer effort to be practical, either because it requires the programmer to write complex proofs, or because it requires large-scale rewriting of software in higher-level languages. In this paper, we argue that it *is* practical to provide many important soundness properties for large-scale systems software, even when written in C, and indeed that this is the approach that the community should be taking.

To demonstrate the feasibility of this approach, we present three soundness tools that we have developed for use with the Linux kernel. First, *Deputy* checks that a pointer always points to valid data of the correct type, even in presence of pointer arithmetic. Second, *CCount* checks that objects are only freed when there are no dangling references to them. Finally, *BlockStop* checks that the kernel does not call blocking functions while interrupts are disabled.

These tools have several properties that we believe are essential to making soundness practical for large-scale systems software such as the Linux kernel:

- **Lightweight, untrusted annotations:** Some annotations to existing source code may be required, but they are minimal, and they extend the type declarations to express simple ideas that should make sense to normal programmers. These annotations are not trusted by the compiler, so errors in the annotations will be caught along with errors in the code.

- **Incremental porting:** It is not necessary to annotate an entire program at once in order to gain any benefit. The system can be made safe one file or even one line at a time, with increasing levels of safety as more code is annotated.

- **Hybrid checking:** Most operations are checked statically, and the rest are checked at run time. Although detecting bugs at compile time is preferable, run-time checks are often necessary for practicality.

- **Erasure semantics:** The tools check, but do not otherwise modify, the behavior of a program. Annotations are written such that they can be ignored ("erased") by the traditional build process. The program is thus not locked into the tool.

- **Trusted code:** Sometimes the behavior of a particular code fragment is too complex for a practical tool to be able to guarantee soundness. In this case, the programmer should be allowed to mark the code as trusted for the purpose of analysis, thus raising its priority for code reviews and testing.

In addition, since we have written and inferred many annotations in the course of this work, we propose a shared repository of annotations and properties inferred for the Linux kernel. This repository, discussed in Section 3, would allow researchers to better collaborate when building such sound analyses in the future.

The major contribution of this paper is the idea that sound static analysis is a feasible and desirable alternative to bug-finding. In support of this idea, we present three analyses that we have used successfully on a working version of the Linux kernel, showing that it is possible to apply sound static analysis tools at a large scale. In addition, we present the basic principles of these tools that have allowed us to achieve this scalability.

Of course, bug-finding tools still have their place in the systems community. Heuristically checking complex properties of systems is often much easier than designing a sound static analysis, and in some sense, it can be viewed as a precursor to a sound analysis. However, we urge the community to focus on such sound analyses whenever possible, since guarantees provide more lasting benefits than bug-finding alone.

In the following section, we discuss each of our three analyses in more detail. Then, we discuss some future directions for research in sound analyses. Finally, we discuss related work and conclude.

2 Sound Analyses for Linux

In this section, we discuss three analyses we have applied to the Linux kernel thus far. This work was performed on a stripped-down version of the Linux 2.6.15.5 kernel, which consists of 435,000 lines of code. This code includes the basic kernel (`kernel/`, `mm/`, and `arch/i386/`), the networking stack (`ipv4`), several file systems (e.g., `ext2` and `procfs`), and several device drivers. The resulting kernel is fully functional in VMware as well as on several machines that the authors use. More details about our experimental kernel are available at `http://ivy.cs.berkeley.edu/`.

We focused on this stripped-down kernel in order to get a working system in place as fast as possible; however, with sufficient manpower, there is no reason to believe that these results could not be extended to the full Linux distribution. In other words, we omitted parts of the Linux kernel for manpower reasons, not for technical reasons.

2.1 Type Safety: Deputy

One major source of errors in Linux is the lack of type safety in C programs. Although a significant portion of C code is type safe, there are a number of language features and programming idioms whose safety cannot be verified by the C compiler. For example, verifying the correctness of array indices, union field references, and type casts is the sole responsibility of the programmer.

Our approach to handling this problem in the Linux kernel is to use the Deputy type system [CHA$^+$07]. Deputy allows programmers to annotate pointer types with bounds information written in terms of other variables in the environment. Deputy also allows annotations for unions, null-terminated sequences, and polymorphic data. In return, Deputy is able to enforce the memory and type safety of the program using a combination of static checking and run-time checks. Note that these annotations are *not* trusted by the compiler, so if the programmer introduces an erroneous annotation, that error will be caught along with any errors in the code itself. In cases where Deputy's type system is insufficient to annotate the code properly, Deputy allows the programmer to explicitly mark code that should be trusted by our tools.

Overall, Deputy guarantees that, at run time, the value of every program expression corresponds to its compile-time type, and in doing so, Deputy prevents out-of-bounds array accesses and misuse of unions. Deputy assumes that trusted code is correct and that code outside the current module conforms to the provided annotations. Since all safety guarantees are relative to these assumptions about trusted or external code, we attempt to use Deputy on as much of the source code as possible.

Unlike other safe C variants such as Cyclone [JMG$^+$02] and CCured [NCH$^+$05], Deputy is incremental and thread safe. That is, programmers are free to add annotations and modify code function-by-function. This is possible because Deputy does not change the representation of the data visible across function boundaries, which allows "deputized" modules to interoperate with standard modules. While the initial version of the file may contain several blocks of trusted code, subsequent versions will gradually eliminate this trusted code in favor of fully annotated and checked code. The same holds of run-time checks: programmers can gradually modify the code to reduce the number of checks that must be deferred until run time. This approach provides an incremental path towards a fully-annotated and type-safe Linux kernel.

In order to convert code to use Deputy, we replace `gcc` with `deputy` in the kernel makefiles. When Deputy is invoked on a C source file, it prints errors for any code that is considered illegal in its type system, such as casts between pointers with different base types. In order to resolve such errors, the programmer must add annotations (such as information about bounds or polymorphism), alter the code, or tell Deputy to trust the code. Once Deputy accepts the code, it will insert any necessary runtime checks and then compile with `gcc`. Booting the new code typically results in a number of warning messages due to incorrect or incomplete annotations; once these are revised, Linux runs as expected.

In previous work [ZCA^{+}06], we described our experience using Deputy on Linux device drivers alone; however, we have now applied Deputy to the full 435,000-line kernel described above. We added standard Deputy annotations to approximately 2627 lines (about 0.6%), and we trusted approximately 3273 lines (less than 0.8%). (Note, however, that the significance of trusting a given line of code varies greatly according to how each line of code is used in the program text and how often it is executed at run time.) This conversion required approximately 7 person-weeks, but the conversion speed increased significantly in the later part of the process due to improvements in the tool and improvements in our ability to identify and apply common annotation patterns.

The `dep` columns of Table 2.1 show the *relative* performance of a 1.6 GHz Pentium M system with the Deputy-enabled kernel compared to the original Linux kernel, measured with the `hbench` [BS97] suite of benchmarks. Benchmarks in the `bw_` column are bandwidth tests (larger is better) and those in the `lat_` column are latency tests (smaller is better). Most tests show that Deputy incurs relatively small overhead. The worst cases are a maximum slowdown of 17% for the local TCP bandwidth test, and a 48% of latency increase for the local UDP latency test. We believe that these results are promising, since they suggest that type safety can be achieved at a modest performance cost.

2.2 Deallocation: CCount

Memory management bugs are a significant cause of software failures and vulnerabilities in C programs. If an object is freed when references to it still exist, then subsequent accesses to the freed object may actually access a new object that has been allocated in the same space, resulting in crashes, security vulnerabilities, and violations of type-safety properties assumed by other analyses.

bw_...	Rel. Perf.		lat_...	Rel. Perf.	
tests	dep	mem	tests	dep	mem
bzero	1.01	1.00	connect	1.10	2.10
file_rd	0.98	0.99	ctx	1.15	1.17
mem_cp	1.00	1.00	ctx2	1.35	1.13
mem_rd	1.00	1.00	fs	1.35	2.73
mem_wr	1.06	0.99	fslayer	1.04	0.98
mmap_rd	0.85	0.93	mmap	1.41	1.21
pipe	0.98	0.97	pipe	1.14	1.12
tcp	0.83	0.66	proc	1.29	1.30
			rpc	1.37	2.01
			sig	1.31	1.23
			syscall	0.74	1.22
			tcp	1.41	2.55
			udp	1.48	1.82

Table 1: Relative performance of the Linux kernels modified for Deputy (dep) and CCount (mem) on the `hbench` benchmark suite. Networking results measured with localhost.

The standard way to avoid memory management problems is to use garbage collection, where objects are automatically freed when they are no longer referenced. However, while previous work shows that it is possible to build an operating system kernel that uses garbage collection for memory management [HLA^{+}05, Mit96, WP97], we believe that retrofitting a garbage collector onto a large legacy kernel such as Linux would be extremely difficult since it would require making significant changes to the way the kernel manages memory.

To address this issue, we have designed CCount, a C-to-C compiler and runtime system that uses reference counting to check the correctness of a C program's existing manual memory management. CCount's compiler modifies all pointer writes to maintain an 8-bit reference count on each 16-byte chunk of memory (a 6.25% space overhead), and the runtime system uses this data to check that frees are safe. Bad frees of objects with $k*256$ references will be missed by such a system, but we expect this case to occur very infrequently in non-malicious code. For total safety, an overflow check could be used.

Using CCount for the Linux kernel required two significant changes. First, we modified Linux's memory management routines to check reference counts and to zero all allocated storage (necessary to avoid decrementing random reference counts when initializing pointers).[1] On failure, we log an error and (optionally) leak the object to guarantee soundness. Second, CCount rewrites pointer writes such as '`*a = b`' to '`RC(b)++, RC(*a)--, *a = b`', where `RC` accesses the reference count of a pointer.[2] To support concurrent code, we must increment and decrement reference counts using atomic operations, and we must ensure that the increment happens before the decrement to avoid transitory zero reference counts. In contrast, we assume that all pointer writes are already protected by appropriate locks and so we do not translate the write itself into an atomic operation. In the future, we plan to check that this assumption does indeed hold by using an additional static analysis tool.

CCount requires accurate type information when objects are freed, copied (memcpy), or cleared (memset). This information is generally similar to information needed by Deputy, so we expect to reuse Deputy annotations in the future. However, we currently have to provide some of this information "by hand". On our small kernel, we had to describe the layout of 32 types, use explicit runtime type information in 27 places, and change 50 uses of memset and memcpy to type-aware versions. We believe that such modifications are acceptable as long as they are made directly by the programmer at the source level, where the programmer can consider their consequences for performance and correctness, as opposed to being made automatically by the compiler.

With these changes, CCount will boot and run our small Linux kernel, but it reports many bad frees. We fix these bad frees by setting breakpoints at the bad free report statement and tracking down the cause of the bad free using our debugging facilities. Fixes to bad frees involve nulling out some extra pointers, usually around the time the corresponding object is freed (27 instances so far[3]) and adding *delayed free scopes* (26 so far). A delayed free scope simply delays all frees (and the associated reference count check) that happen inside it until its end, greatly simplifying the checks for complex or cyclical data structures. We have spent approximately 6 person-weeks porting CCount and making these changes to the Linux kernel, and we can now verify the correctness of all of the ∼107k frees that occur from boot time until the login prompt is available. And after running the hbench benchmark suite, our modified kernel reports 677 "bad" frees from ∼100M free calls. We are confident that we can eliminate these remaining bad frees.

The relative performance of the CCount-kernel vs the original Linux kernel is shown in the mem columns of Table 2.1. These numbers are measured on a 2.33GHz Intel® Xeon® 5140, running a uniprocessor kernel. The performance degradation is reasonable ($< 30\%$) except on the networking and file system tests. Additionally, we estimated the performance impact of CCount with an SMP kernel by changing the increment and decrement operations for reference count updates to use "locked" instructions. This leads to significant degradations in performance: bw_tcp decreases to 0.50, lat_tcp increases to 3.60, lat_ctx increases to 1.81, etc. Clearly these results need improving, which we hope to achieve through a combination of manual tuning, compiler optimization and a better runtime system.

2.3 Call Graph Analysis: BlockStop

There are many other invariants beyond type and memory safety that must be enforced in the kernel. One tool that is useful for several of these invariants is a call graph. Once we know which functions can be called where, we can begin to analyze important control-flow properties.

BlockStop is a whole-program analysis to enforce the requirement that the kernel does not call any functions that may block while interrupts are disabled, such as while holding a spinlock or handling an interrupt. Once we've run this analysis, we can emit an annotation for each function (and function pointer) that might eventually call a blocking function. Not only is this information useful for humans trying to understand the code, but the annotations can be checked incrementally whenever a file is changed, which preserves separate compilation.

A call graph is a directed graph where each node corresponds to a function and each outgoing edge represents the functions that it might call. The major challenge is to account for calls through function pointers. We use a whole-program *points-to analysis* to determine which functions a given pointer could refer to. Thanks to the type safety provided by Deputy and CCount, this points-to analysis is sound, except that we do not currently detect function calls made within inline assembly.

To find which functions might block, we annotate certain functions with a new blocking attribute, such as copy_to/from_user, wait_for_completion, etc. Allocators such as kmalloc have a special annotation to denote the fact that they may block if they are called with the GFP_WAIT flag. We then propagate this information backwards through the call graph to get a sound approximation of the set of functions that might block.

We ran this analysis on our test kernel and found one bug. We also encountered false positives, mostly due to the overly-conservative points-to analysis of function pointers. Replacing our simple points-to analysis with one that is field- and context- sensitive would improve the results, but to resolve these false positives quickly, we turned to run-time checks. We defined a special function that panics if interrupts are disabled, and we manually inserted calls to this function in 15 places in the kernel. For example, read_chan is a blocking function that BlockStop's points-to analysis incorrectly believes can be called by flush_to_ldisk while interrupts are disabled. Adding this run-time check to the start of read_chan reflects our assertion that this function will not actually be called by flush_to_ldisk. By ruling out these infeasible control-flow paths via runtime checks, we allow BlockStop to verify the desired property for our experimental kernel without any warnings or false positives.

3 Looking Forward

We believe that the three previous analyses represent only the tip of the iceberg in terms of sound analyses that can be used effectively on systems such as Linux. Here we discuss proposals for future analyses as well as ideas for making the results of these analyses widely available.

3.1 Future Analyses

There are many other opportunities to create sound analyses with the properties we discussed.

First, we are in the early stages of designing a hybrid checking tool for verifying *lock safety* in Linux. In addition to checking that deadlocks are impossible by verifying that the code uses a consistent locking order, this analysis will check Linux-specific invariants such as the requirement that the same spinlock is not acquired in interrupts and in process context with interrupts turned on. Light annotations will be used to name the locks, and run-time checks will be used when static checking does not suffice. We rely upon type and memory safety guarantees provided by Deputy and CCount.

Second, the call graph built for BlockStop can be used to prevent *stack overflow*. Given a sound call graph and information about the size of each stack frame, as in the Capriccio thread package [BCZ$^+$03], we can ensure that every possible chain of function calls stays within its allotted 4 or 8 kB of stack space. Stack space annotations on each function will enable incremental verification. For recursive calls, run-time checks will be needed.

As a third example, it is possible to create a simple analysis for ensuring that *error codes* are properly checked at call sites. Programmers can annotate each functions with the set of codes that the function could return, or the programmer could simply indicate to the compiler that negative constant return values are error codes. Then a flow-sensitive analysis at call sites could verify that each of the error codes are accounted for, either together or separately. Calls through function pointers could use a merged list of codes from the functions that the pointer may alias.

Further examples include user/kernel pointers, tainted data flow, and concurrency issues such as identifying shared and thread-local data. All of these properties can be checked by analyses that follow the framework outlined here: lightweight annotations with run-time checks and trusted code where necessary.

3.2 Collaboration

A consequence of applying our tools to the Linux kernel is that we have generated a large amount of information about functions and types in the Linux kernel in a form that is usable by the compiler. Some of this information was generated manually by reading comments and code, while other properties were inferred by our tools. In order to make this information available to other researchers and programmers, we propose the creation of a *collaborative database* of source code information that would allow different researchers and tools to share and reuse information about publicly available source code such as the Linux kernel.

For example, this database could provide pointer alias information and bounds information for function arguments and global variables within Linux. This information is required by both Deputy and CCount, and it will almost certainly be of use to future analyses. We can also store information about blocking functions, error codes, and so on. In addition to aiding researchers, this information would also provide a useful reference for programmers who wish to see additional invariants that are not specified directly in the code or comments. Indeed, with the wide variety of possible analyses that we propose, it may be useful for the programmer to store this information on the side instead of cluttering up the code directly. In addition, the ability to incorporate this information from an external source may be useful when trying to maintain this information across multiple versions of the kernel.

We have seeded this repository of annotations at our web page: `http://ivy.cs.berkeley.edu/`. Our Linux annotations are available here, and we encourage other researchers to help us in expanding the scope of these annotations.

4 Related Work

We previously described the Deputy type system [CHA$^+$07], and the application of Deputy to Linux device drivers in SafeDrive [ZCA$^+$06]. Some of us worked on on CCured [NCH$^+$05], a predecessor to Deputy. Here, we present our experience applying Deputy to a complete, bootable Linux kernel, and discuss the basic principles of both Deputy and our other tools that allow us to scale these analyses to large programs.

CSSV [DRS03], Saber-C [KLP88], and a number of other projects [ABS94, SV98] are capable of verifying some soundness properties for C programs. However, we are not aware of any previous attempt to apply such tools to a program as large and complex as the Linux kernel.

In addition, there exist several safe C variants, such as CCured [NCH$^+$05] and Cyclone [JMG$^+$02], which attempt to impose a stricter typing discipline on C programs. However, these systems require changes to data structures that make an incremental transition to these C variants difficult.

Eau Claire [Che02], MC [HCXE02], and MECA [YKXE03] are three examples of bug-finding tools for systems software. While these tools find many important bugs, they do not guarantee that no more bugs exist, and they do not prevent bug reintroduction. Also, each run of these tools requires a programmer to sift through false positives every time the tool is used, whereas our approach yields a modified program that checks cleanly after the initial programmer effort.

Projects such as Melange [MHD+07], JavaOS [Mit96], and Inferno [WP97] have attempted to write systems code in safe languages. However, when legacy code already exists in C, we believe it would be easier to apply our soundness tools to this legacy code than to rewrite the code in a safe language. Our approach focuses on incremental tools that allow programmers to preserve their investment in existing code while improving its reliability.

5 Conclusion

It is estimated that software vulnerabilities cost $13 billion in 2001, $30 billion in 2002, and $55 billion in 2003 [New04]. While bug-finding tools can be very helpful in finding some of these defects, sound static analyses allow us to guarantee their absence.

In this paper, we have discussed our experience thus far in applying soundness tools to the Linux kernel. Our results are encouraging: we were able to rule out many type errors and buffer overruns in 435,000 lines of kernel code with only 7 weeks of effort, and we were able to verify 98% of the deallocations in this code with only 6 weeks of effort. We thus have reason to believe that it is both practical and wise to focus on making systems software completely safe against such defects.

Notes

[1] So far we have only modified `kmalloc`, `kfree`, and the slab allocators, but extending this support to `vmalloc`, `vfree`, and `alloc_page` should be straightforward.

[2] At the time of writing, the kernel version of CCount does not track references from local variables; however, we expect to have this feature implemented soon.

[3] We also null out the pointer passed to free functions.

References

[ABS94] T. Austin, S. Breach, and G. Sohi. Efficient detection of all pointer and array access errors. In *PLDI*, 1994.

[BCZ+03] R. von Behren, J. Condit, F. Zhou, G. Necula, and E. Brewer. Capriccio: Scalable threads for internet services. In *SOSP*, 2003.

[BS97] A. Brown and M. Seltzer. Operating system benchmarking in the wake of Lmbench: A case study of the performance of NetBSD on the Intel x86 architecture. In *SIGMETRICS*, 1997.

[CHA+07] J. Condit, M. Harren, Z. Anderson, D. Gay, and G. Necula. Dependent types for low lovel programming. In *European Symposium on Programming*, 2007.

[Che02] B. Chess. Improving computer security using extended static checking. In *IEEE Security and Privacy*, 2002.

[DRS03] N. Dor, M. Rodeh, and M. Sagiv. CSSV: Towards a realistic tool for statically detecting all buffer overflows in C. In *PLDI*, 2003.

[HCXE02] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *PLDI*, 2002.

[HLA+05] G. Hunt, J. Larus, M. Abadi, M. Aiken, P. Barham, M. Fahndrich, C. Hawblitzel, O. Hodson, S. Levi, N. Murphy, B. Steensgaard, D. Tarditi, T. Wobber, and B. Zill. An Overview of the Singularity Project. Technical Report MSR-TR-2005-135, Microsoft Research, 2005.

[JMG+02] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*, 2002.

[KLP88] S. Kaufer, R. Lopez, and S. Pratap. Saber-C: an interpreter-based programming environment for the C language. In *USENIX Summer Conference*, 1988.

[MHD+07] A. Madhavapeddy, A. Ho, T. Deegan, D. Scott, and R. Sohan. Melange: Creating a ”functional” internet. In *EuroSys*, 2007.

[Mit96] J. Mitchell. JavaOS: Back to the future (abstract). In *OSDI*, 1996.

[NCH+05] G. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems*, 27(3), May 2005.

[New04] ZDNet News. PC viruses spawn $55 billion loss in 2003, Jan 2004.

[SV98] G. Smith and D. Volpano. A sound polymorphic type system for a dialect of C. *Science of Computer Programming*, 32(1–3):49–72, 1998.

[WP97] P. Winterbottom and R. Pike. The design of the Inferno virtual machine. In *IEEE Compcon*, 1997.

[YKXE03] J. Yang, T. Kremenek, Y. Xie, and D. Engler. MECA: an extensible, expressive system and language for statically checking security properties. In *ACM Computer and Communications Security*, 2003.

[ZCA+06] F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harren, G. Necula, and E. Brewer. SafeDrive: Safe and recoverable extensions using language-based techniques. In *OSDI*, 2006.