USENIX Association

Proceedings of the 13th USENIX Security Symposium

San Diego, CA, USA
August 9–13, 2004

Finding User/Kernel Pointer Bugs With Type Inference

Rob Johnson David Wagner
University of California at Berkeley

Abstract

Today's operating systems struggle with vulnerabilities from careless handling of user space pointers. User/kernel pointer bugs have serious consequences for security: a malicious user could exploit a user/kernel pointer bug to gain elevated privileges, read sensitive data, or crash the system. We show how to detect user/kernel pointer bugs using type-qualifier inference, and we apply this method to the Linux kernel using CQUAL, a type-qualifier inference tool. We extend the basic type-inference capabilities of CQUAL to support context-sensitivity and greater precision when analyzing structures so that CQUAL requires fewer annotations and generates fewer false positives. With these enhancements, we were able to use CQUAL to find 17 exploitable user/kernel pointer bugs in the Linux kernel. Several of the bugs we found were missed by careful hand audits, other program analysis tools, or both.

1 Introduction

Security critical programs must handle data from untrusted sources, and mishandling of this data can lead to security vulnerabilities. Safe data-management is particularly crucial in operating systems, where a single bug can expose the entire system to attack. Pointers passed as arguments to system calls are a common type of untrusted data in OS kernels and have been the cause of many security vulnerabilities. Such user pointers occur in many system calls, including, for example, `read`, `write`, `ioctl`, and `statfs`. These user pointers must be handled very carefully: since the user program and operating system kernel reside in conceptually different address spaces, the kernel must not directly dereference pointers passed from user space, otherwise security holes can result. By exploiting a user/kernel bug, a malicious user could take control of the operating system by overwriting kernel data structures, read sensitive data out of kernel memory, or simply crash the machine by corrupting kernel data.

Kernel version	Bugs found
Linux 2.4.20	11
Linux 2.4.23	10

Table 1: User/kernel bugs found by CQUAL. Each of these bugs represents an exploitable security vulnerability. Four bugs were common to both 2.4.20 and 2.4.23, for a total of 17 unique bugs. Eight of the bugs in Linux 2.4.23 were also in Linux 2.5.63.

User/kernel pointer bugs are unfortunately all too common. In an attempt to avoid these bugs, the Linux programmers have created several easy-to-use functions for accessing user pointers. As long as programmers use these functions correctly, the kernel is safe. Unfortunately, almost every device driver must use these functions, creating thousands of opportunities for error, and as a result, user/kernel pointer bugs are endemic. This class of bugs is not unique to Linux. Every version of Unix and Windows must deal with user pointers inside the OS kernel, so a method for automatically checking an OS kernel for correct user pointer handling would be a big step in developing a provably secure and dependable operating system.

We introduce type-based analyses to detect and eliminate user/kernel pointer bugs. In particular, we augment the C type system with type qualifiers to track the provenance of all pointers, and then we use type inference to automatically find unsafe uses of user pointers. Type qualifier inference provides a principled and semantically sound way of reasoning about user/kernel pointer bugs.

We implemented our analyses by extending CQUAL[7], a program verification tool that performs type qualifier inference. With our tool, we discovered several previously unknown user/kernel pointer bugs in the Linux kernel. In our experiments, we discovered 11 user/kernel pointer bugs in Linux kernel 2.4.20 and 10 such bugs in Linux 2.4.23. Four bugs were common to 2.4.20 and 2.4.23, for a total of 17 different bugs, and eight of these 17 were still present in the 2.5 development series. We have confirmed all but two of the bugs with kernel developers. All the bugs were exploitable.

We needed to make several significant improvements to CQUAL in order to reduce the number of false positives it reports. First, we added a context-sensitive analysis to CQUAL, which has reduced the number of false positives and the number of annotations required from the programmer. Second, we improved CQUAL's handling of C structures by allowing fields of different instances of a structure to have different types. Finally, we improved CQUAL's analysis of casts between pointers and integers. Without these improvements, CQUAL reported far too many false positives. These two improvements reduce the number of warnings 20-fold and make the task of using CQUAL on the Linux kernel manageable.

Our principled approach to finding user/kernel pointer bugs contrasts with the ad-hoc methods used in MECA[15], a prior tool that has also been used to find user/kernel pointer bugs. MECA aims for a very low false positive rate, possibly at the cost of missing bugs; in contrast, CQUAL aims to catch all bugs, at the cost of more false positives. CQUAL's semantic analysis provides a solid foundation that may, with further research, enable the possibility of formal verification of the absence of user/kernel pointer bugs in real OS's.

All program analysis tools have false positives, but we show that programmers can substantially reduce the number of false positives in their programs by making a few small stylistic changes to their coding style. By following a few simple rules, programmers can write code that is efficient and easy to read, but can be automatically checked for security violations. These rules reduce the likelihood of getting spurious warnings from program verification or bug-finding tools like CQUAL. These rules are not specific to CQUAL and almost always have the benefit of making programs simpler and easier for the programmer to understand.

In summary, our main contributions are

- We introduce a semantically sound method for analyzing user/kernel security bugs.
- We identify 17 new user/kernel bugs in several different versions of the Linux kernel.
- We show how to reduce false positives by an order of magnitude, and thereby make type-based analysis of user/kernel bugs practical, by enhancing existing type inference algorithms in several ways. These improvements are applicable to any dataflow oriented program analysis tool.

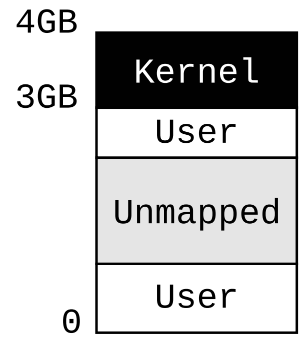

Figure 1: The Linux virtual memory layout on 32-bit architectures.

- We develop guidelines that programmers can follow to further reduce the number of false positives when using program verification tools.

An extended version of this paper is available from the authors' web pages.

We begin by describing user/kernel pointer bugs in Section 2. We then describe type qualifier inference, and our refinements to this technique, in Section 3. Our experimental setup and results are presented in Sections 4 and 5, respectively. Section 6 discusses our false positive analysis and programming guidelines. We consider other approaches in Section 7. Finally, we summarize our results and give several directions for future work in Section 8.

2 User/kernel Pointer Bugs

All Unix and Windows operating systems are susceptible to user pointer bugs, but we'll explain them in the context of Linux. On 32-bit computers, Linux divides the virtual address space seen by user processes into two sections, as illustrated in Figure 1. The virtual memory space from 0 to 3GB is available to the user process. The kernel executable code and data structures are mapped into the upper 1GB of the process' address space. In order to protect the integrity and secrecy of the kernel code and data, the user process is not permitted to read or write the top 1GB of its virtual memory. When a user process makes a system call, the kernel doesn't need to change VM mappings, it just needs to enable read and write access to the top 1GB of virtual memory. It disables access to the top 1GB before returning control to the user process.

This provides a conceptually clean way to prevent user processes from accessing kernel memory directly, but it imposes certain obligations on kernel programmers. We will illustrate this with a toy example: suppose we want to implement two new system calls, `setint` and `getint`: [1]

```
int x;
void sys_setint(int *p)
{
  memcpy(&x, p, sizeof(x)); // BAD!
}
void sys_getint(int *p)
{
  memcpy(p, &x, sizeof(x)); // BAD!
}
```

Imagine a user program which makes the system call

```
getint(buf);
```

In a well-behaved program, the pointer, `buf`, points to a valid region of memory in the user process' address space and the kernel fills the memory pointed to by `buf` with the value of `x`.

However, this toy example is insecure. The problem is that a malicious process may try to pass an invalid `buf` to the kernel. There are two ways `buf` can be invalid.

First, `buf` may point to unmapped memory in the user process' address space. In this case, the virtual address, `buf`, has no corresponding physical address. If the kernel attempts to copy `x` to the location pointed to by `buf`, then the processor will generate a page fault. In some circumstances, the kernel might recover. However, if the kernel has disabled interrupts, then the page fault handler will not run and, at this point, the whole computer locks up. Hence the toy kernel code shown above is susceptible to denial-of-service attacks.

Alternatively, an attacker may attempt to pass a `buf` that points into the kernel's region of memory. The user process cannot read or write to this region of memory, but the kernel can. If the kernel blindly copies data to `buf`, then several different attacks are possible:

- By setting `buf` to point to the kernel executable code, the attacker can make the kernel overwrite its own code with the contents of `x`. Since the user can also set the value of `x` via legitimate calls to `setint`, she can use this to overwrite the kernel code with any new code of her choice. For example, she could eliminate permission checking code in order to elevate her privileges.
- The attacker can set `buf` to point to kernel data structures that store her user id. By overwriting these with all 0s, the attacker can gain root privileges.
- By passing in random values for `buf` the attacker can cause the kernel to crash.

The above examples show the importance of validating a buffer pointer passed from user space before copying data into that buffer. If the kernel forgets to perform this check, then a malicious user gains control of the system. In most cases, an attacker can exploit reads from unchecked pointers, too. Imagine an attacker making the system call

```
setint(buf);
```

The kernel will copy 4 bytes from `buf` into `x`. An attacker could point `buf` at kernel file buffers, and the kernel would copy the contents of those file buffers into `x`. At this point, the attacker can read the contents of the file buffer out of `x` via a legitimate call to `getint`. With a little luck, the user can use this attack to learn the contents of `/etc/shadow`, or even the secret TLS key of the local web server.

User/kernel pointer bugs are hard to detect during testing because, in most cases, they succeed silently. As long as user programs pass valid pointers to system calls, a buggy system call implementation will work correctly. Only a malicious program will uncover the bug.

The `setint` and `getint` functions shown above may seem contrived, but two of the bugs we found effectively implemented these two system calls (albeit not under these names).

In order to avoid these errors, the Linux kernel contains several user pointer access functions that kernel developers are supposed to use instead of `memcpy` or dereferencing user pointers directly. The two most prominent of these functions are `copy_from_user` and `copy_to_user`, which behave like `memcpy` but perform the required safety checks on their user pointer arguments. Correct implementations of `setint` and `getint` would look like

```
int x;
void sys_setint(int *p)
{
```

```
  copy_from_user(&x, p, sizeof(x));
}
void sys_getint(int *p)
{
  copy_to_user(p, &x, sizeof(x));
}
```

As long as the user pointer access functions like copy_from_user and copy_to_user are used correctly, the kernel is safe. Unfortunately, Linux 2.4.20 has 129 system calls accepting pointers from user space as arguments. Making matters worse, the design of some system calls, like ioctl, require every device driver to handle user pointers directly, as opposed to having the system call interface sanitize the user pointers as soon as they enter the kernel. Thus the Linux kernel has hundreds of sources of user pointers and thousands of consumers, all of which must be checked for correctness, making manual auditing impossible.

This problem is not unique to Linux. For example, FreeBSD has similar user buffer access functions. Even though we have presented the problem in the context of the Linux kernel VM setup, the same problem would arise in other VM architectures, e.g. if the kernel was direct mapped and processes lived in virtual memory.

The above discussion makes it clear that there are essentially two disjoint kinds of pointers in the kernel:

User pointers: A pointer variable whose value is under user control and hence untrustworthy.

Kernel pointers: A pointer variable whose value is under kernel control and guaranteed by the kernel to always point into the kernel's memory space, and hence is trustworthy.

User pointers should always be verified to refer to user-level memory before being dereferenced. In contrast, kernel pointers do not need to be verified before being dereferenced.

It is easy for programmers to make user pointer errors because user pointers look just like kernel pointers—they're both of type "void *". If user pointers had a completely different type from kernel pointers, say

```
typedef struct {
  void *p;
} user_pointer_t;
```

then it would be much easier for programmers to distinguish user and kernel pointers. Even better, if this type were opaque, then the compiler could check that the programmer never accidentally dereferenced a user pointer. We could thus think of user pointers as an abstract data type (ADT) where the only permitted operations are copy_{to,from}_user, and then the type system would enforce that user pointers must never be dereferenced. This would prevent user/kernel pointer bugs in a clean and principled way. The downside of such an approach is that programmers can no longer do simple, safe operations, like p++, on user pointers.

Fortunately, we can have all the advantages of typed pointers without the inflexibility if we tweak the concept slightly. All that's really needed is a *qualifier* on pointer types to indicate whether they were passed from user space or not. Consider, for example, the following code:

```
int copy_from_user(void * kernel to,
                   void * user from,
                   int len);
int memcpy(void * kernel to,
           void * kernel from,
           int len);
int x;
void sys_setint(int * user p)
{
  copy_from_user(&x, p, sizeof(x));
}
void sys_getint(int * user p)
{
  memcpy(p, &x, sizeof(x));
}
```

In this example, kernel and user modify the basic void * type to make explicit whether the pointer is from user or kernel space. Notice that in the function sys_setint, all the type qualifiers match. For instance, the user pointer p is passed into the user argument from of copy_from_user. In contrast, the function sys_getint has a type error, since the user pointer p is passed to memcpy, which expects a kernel pointer instead. In this case, this type error indicates an exploitable user/kernel bug.

In this paper, we use CQUAL, which allows programmers to add user-defined qualifiers to the C programming language. We create *user* and *kernel* type qualifiers and we use CQUAL to type-check the kernel. We have analyzed several different versions of the Linux kernel for user/kernel bugs, finding a total of 17 different exploitable user/kernel pointer bugs.

3 Type Qualifier Inference

We begin with a review of type qualifier inference. The C programming language supports a few basic types, like `int`, `float`, and `char`. Programmers can construct types such as pointers, or references, to any type. For example, in our notation, `ref` (`int`) denotes a reference to a memory location of type `int`, or, in other words, a pointer of type `int *`. The C language also contains a few type qualifiers, like *const*, that can be applied to any of the basic or constructed types.

CQUAL allows programmers to create new, user-defined qualifiers that modify the standard C types, just like *const*. In our case, we use CQUAL to define qualifiers *user* and *kernel*. The intended meaning is as follows: a *user* `int` is an `int` whose value is possibly under user control and hence is untrustworthy; if τ is any type, a *user* τ is a value of type τ that is possibly under user control; and likewise, a *kernel* τ is a value of type τ that is under kernel control. For instance, a *user* `ref` (`int`) is a reference to an `int` that is stored in user space; its value is an address in the mapped portion of user memory, and dereferencing it yields an `int`. In C, a pointer `p` of this type would be declared by the code `int *` *user* `p;`, and the `int` typically would be stored in user space, while the pointer to the `int` is stored in kernel space. We refer to a C type, together with its qualifiers, as a *qualified type*.

Note that qualifiers can modify each level of a standard type. The C type `int *` *user* is different from `int` *user* `*`; in the former case, it is the pointer (i.e., address) whose value is under user control, while in the latter case, it is the integer whose value is under user control. As another example, the programmer could declare a variable of C type `int *` *user* `*` *kernel*, which corresponds in our notation to *kernel* `ref` (*user* `ref` (`int`)); this would refer to a pointer, whose value came from the kernel, that points to a pointer, whose value originally came from user space, to an integer.

In general, the invariant we maintain is that every pointer of type *kernel* `ref` $(\cdots)$ has a value referring to an address in kernel space and cannot be controlled by any user process. Pointers of type *user* `ref` $(\cdots)$ may contain any address whatsoever. Normally, when the system is not under attack, user pointers refer to mapped memory within user space, but in the presence of an adversary, this cannot be relied upon. Thus a pointer of type *kernel* `ref` $(\cdots)$ is safe to dereference directly; *user* `ref` $(\cdots)$ types are not.

The type qualifier inference approach to program analysis has several advantages. First, type qualifier inference requires programmers to add relatively few annotations to their programs. Programmers demand tools with low overhead, and type qualifier inference tools certainly meet those demands. Second, type qualifiers enable programmers to find bugs at compile time, before an application becomes widely distributed and impossible to fix. Third, type qualifiers are sound; if a sound analysis reports no errors in a source program, then it is *guaranteed* to be free of the class of bugs being checked. Soundness is critical for verifying security-relevant programs; a single missed security bug compromises the entire program.

Like standard C types and type qualifiers, CQUAL is flow-insensitive. This means that each program expression must have one qualified type that will be valid throughout the entire execution of the program. For example, just as C doesn't allow a local variable to sometimes be used as an `int` and sometimes as a `struct`, CQUAL does not permit a pointer to sometimes have type *user* `ref` (`int`) and sometimes have type *kernel* `ref` (`int`).

Programmers can use these qualifiers to express specifications in their programs. As an example, Figure 2 shows type qualifier annotations for `copy_from_user` and `copy_to_user`. With these annotations in place, if a programmer ever calls one of these functions with, say, a *user* pointer where a *kernel* pointer is expected, CQUAL will report a type error. Figure 2 also shows CQUAL's syntax for annotating built-in C operators. The `__op_deref` annotation prohibits dereferencing *user* pointers. This annotation applies to all dereferences, including the C "`*`" and "`->`" operators, array indexing, and implicit dereferences of references to local variables.

In certain cases, Linux allows *kernel* pointers to be treated as if they were *user* pointers. This is analogous to the standard C rule that a *nonconst* [2] variable can be passed to a function expecting a *const* argument, and is an example of qualifier subtyping. The notion of subtyping should be intuitively familiar from the world of object-oriented programming. In Java, for instance, if A is a subclass of B, then an object of class A can be used wherever an object of class B is expected, hence A can be thought of as a subtype of B (written $A < B$).

```
int copy_from_user(void user * kernel kto,
                   void * user ufrom,
                   int len);
int copy_to_user(void * user uto,
                 void * kernel kfrom,
                 int len);
α __op_deref(α * kernel p);
```

Figure 2: Annotations for the two basic user space access functions in the Linux kernel. The first argument to `copy_from_user` must be a pointer to kernel space, but after the copy, its contents will be under user control. The `__op_deref` annotation declares that the C dereference operator, "`*`", takes a *kernel* pointer to any type, α, and returns a value of type α.

CQUAL supports subtyping relations on user-defined qualifiers, so we can declare that *kernel* is a subtype of *user*, written as *kernel* $<$ *user*. CQUAL then extends qualifier subtyping relationships to qualified-type subtyping rules as follows. First, we declare that *kernel* `int` $<$ *user* `int`, because any `int` under kernel control can be treated as a `int` possibly under user control. The general rule is[3]

$$\frac{Q \leq Q'}{Q\ \texttt{int} \leq Q'\ \texttt{int}}$$

This notation states that if qualifier Q is a subtype of qualifier Q', then Q `int` is a subtype of Q' `int`, or in other words, any value of type Q `int` can be safely used whereever a Q' `int` is expected. For example, if a function expects a *const* `int`, then it may be called with a *nonconst* `int` because *nonconst* $<$ *const*, and therefore *nonconst* `int` $<$ *const* `int`.

The rule for subtyping of pointers is slightly more complicated.

$$\frac{Q \leq Q' \qquad \tau = \tau'}{Q\ \texttt{ref}\ (\tau) \leq Q'\ \texttt{ref}\ (\tau')}$$

Notice that this rule requires that the referent types, τ and τ', be equal, not just that $\tau \leq \tau'$. This is a well-known typing rule that is required for soundness. This rule captures CQUAL's sound handling of aliasing, a problem that has plagued other bug-finding tools.

So far, we have described the basis for a type-checking analysis. If we were willing to manually insert a *user* or *kernel* qualifier at every level of every type declaration in the Linux kernel, we would be able to detect user/pointer bugs by running standard type-checking algorithms. However, the annotation burden of marking up the entire Linux kernel in this way would be immense, and so we need some way to reduce the workload on the programmer.

We reduce the annotation burden using *type inference*. The key observation is that the vast majority of type qualifier annotations would be redundant, and could be inferred from a few base annotations, like those in Figure 2. Type qualifier inference provides a way to infer these redundant annotations: it checks whether there is any way to extend the source code annotations to make the result type-check. CQUAL implements type qualifier inference. For example, this allows CQUAL to infer from the code

```
int bad_ioctl(void * user badp)
{
  char badbuf[8];
  void *badq = badp;
  copy_to_user(badbuf, badq, 8);
}
```

that `badq` must be a *user* pointer (from the assignment `badq = badp`), but it is used as a *kernel* pointer (since `badq` is passed to `copy_from_user`). This is a type error. In this case, the type error indicates a bona fide security hole.

Notice that, in this example, the programmer didn't have to write an annotation for the type of `badq`—instead, it was inferred from other annotations. Inference can dramatically reduce the number of annotations required from the programmer. In our experiments with Linux, we needed less than 300 annotations for the whole kernel; everything else was inferred by CQUAL's type inference algorithm.

3.1 Soundness

As mentioned before, the theoretical underpinnings of type inference are sound, but C contains several constructs that can be used in unsound ways. Here we explain how CQUAL deals with these constructs.

No memory safety. CQUAL assumes programs are memory safe, i.e. that they contain no buffer overflows. Type qualifiers cannot detect buffer overflows, but other tools, such as BOON[14] or CCured[10], do address memory safety. In conjunction with these tools, CQUAL forms a powerful system for verifying security properties.

Unions. CQUAL assumes programmers use unions safely, i.e. that the programmer does not write to one field of a union and read from a different one. Like memory-safety, type qualifiers cannot detect invalid uses of unions, but union-safety could plausibly be checked by another program analysis tool. Programmers could use CQUAL together with such a tool if it seems unrealistic to assume that programmers always use unions safely.

Separate Compilation. Type qualifier inference works from a few base annotations, but if the annotations are incomplete or incorrect, then the results may not be sound. In legacy systems like the Linux kernel, each source module provides one interface and makes use of many others, but none of these interfaces are annotated. Thus any analysis of one source file in isolation will be unsound. To get sound results, a whole-program analysis is required.

Type casts. C allows programmers to cast values to arbitrary types. We had to extend CQUAL slightly to handle some obscure cases. With these enhancements, our experience is that CQUAL just "does the right thing" in all cases we've encountered. For example, if the programmer casts from one type of struct to another, then CQUAL matches up the corresponding fields and flows qualifiers appropriately.

Inline assembly. CQUAL ignores inline assembly, which may cause it to miss some type errors. Analyzing inline assembly would require detailed knowledge of the instruction set and instruction semantics of a specific processor. Inline assembly is rare in most programs, and programmers can obtain sound analysis results by annotating functions containing inline assembly. Alternatively, programmers could provide C implementations of inline assembly blocks. The C implementations would not only benefit CQUAL, they would serve to document the corresponding assembly code.

3.2 Our Analysis Refinements

We made several enhancements to CQUAL to support our user/kernel analysis. The challenge was to improve the analysis' precision and reduce the number of false positives without sacrificing scalability or soundness. One of the contributions of this work is that we have developed a number of refinements to CQUAL that meet this challenge. These refinements may be generally useful in other applications as well, so our techniques may be of independent interest. However, because the technical details require some programming language background to explain precisely, we leave the details to to the extended version of this paper and we only summarize our improvements here.

Context-Sensitivity. Context-sensitivity enables CQUAL to match up function calls and returns. Without context-sensitivity, type constraints at one call site to a function `f` will "flow" to other call sites. Context-sensitivity simultaneously reduces the number of annotations programmers must write and the number of false positives CQUAL generates. Experiments performed with Percent-S, a CQUAL-based tool for detecting format string bugs, found that context-sensitivity could reduce the false positive rate by over 90%, depending on the application[11].

Field-sensitivity. Field-sensitivity enables CQUAL to distinguish different instances of structures. Without field-sensitivity, every variable of type `struct foo` shares one qualified type, so a type constraint on field `x` of one instance flows to field `x` of every other instance. Without this enhancement, CQUAL was effectively unable to provide any useful results on the Linux kernel because the kernel uses structures so heavily. In our early experiments, the field-insensitive analysis produced a false positive for almost every call to `copy_from_user`, `copy_to_user`, etc. With our more precise analysis of structures and fields, CQUAL produces only a few hundred warnings.

Well-formedness Constraints. Well-formedness constraints enable CQUAL to enforce special type rules related to structures and pointers. We used this feature to encode rules like, "If a structure was copied from user space (and hence is under user control), then so were all its fields." Without support for well-formedness constraints, CQUAL would miss some user/kernel bugs (see, e.g., Figure 4). Well-formedness constraints require no additional annotations; they are optional properties that are enabled or disabled in the configuration file that describes the type system used for an analysis.

Sound and Precise Pointer/Integer Casts. CQUAL now analyzes casts between pointers and integers soundly. Our improvement to CQUAL's cast handling simultaneously fixes a soundness bug and improves CQUAL's precision.

Together, these refinements dramatically reduce CQUAL's false positive rate. Before we made these improvements, CQUAL reported type errors (almost all of which were false positives) in almost every kernel source file. Now CQUAL finds type errors in only about 5% of the kernel source files, a 20-fold reduction in the number of false positives.

3.3 Error Reporting

In addition to developing new refinements to type qualifier inference, we also created a heuristic that dramatically increases the "signal-to-noise" ratio of type inference error reports. We implemented this heuristic in CQUAL, but it may be applicable to other program analysis tools as well.

Before explaining our heuristic, we first need to explain how CQUAL detects type errors. When CQUAL analyzes a source program, it creates a qualifier constraint graph representing all the type constraints it discovers. A typing error occurs whenever there is a valid path [4] from qualifier Q to qualifier Q' where the user-specified type system requires that $Q \not\leq Q'$. In the user/kernel example, CQUAL looks for valid paths from *user* to *kernel*. Since each edge in an error path is derived from a specific line of code, given an error path, CQUAL can walk the user through the sequence of source code statements that gave rise to the error, as is shown in Figure 3. This allows at least rudimentary error reporting, and it is what was implemented in CQUAL prior to our work.

Unfortunately, though, such a simple approach is totally inadequate for a system as large as the Linux kernel. Because typing errors tend to "leak out" over the rest of the program, one programming mistake can lead to thousands of error paths. Presenting all these paths to the user, as CQUAL used to do, is overwhelming: it is unlikely that any user will have the patience to sort through thousands of redundant warning messages. Our heuristic enables CQUAL to select a few canonical paths that capture the fundamental programming errors so the user can correct them.

Many program analyses reduce finding errors in the input program to finding invalid paths through a graph, so a scheme for selecting error paths for display to the user could benefit a variety of program analyses.

To understand the idea behind our heuristic, imagine an ideal error reporting algorithm. This algorithm would pick out a small set, S, of statements in the original source code that break the type-correctness of the program. These statements may or may not be bugs, so we refer to them simply as untypable statements. The algorithm should select these statements such that, if the programmer fixed these lines of code, then the program would type-check. The ideal algorithm would then look at each error path and decide which statement in S is the "cause" of this error path. After bucketing the error paths by their causal statement, the ideal algorithm would select one representive error path from each bucket and display it to the user.

Implementing the ideal algorithm is impossible, so we approximate it as best we can. The goal of our approximation is to print out a small number of error traces from each of the ideal buckets. When the approximation succeeds, each of the untypable statements from the ideal algorithm will be represented, enabling the programmer to address all his mistakes.

Another way to understand our heuristic is that it tries to eliminate "derivative" and "redundant" errors, i.e., errors caused by one type mismatch leaking out into the rest of the program, as well as multiple error paths that only differ in some minor, inconsequential way.

The heuristic works as follows. First, CQUAL sorts all the error paths in order of increasing length. It is obviously easier for the programmer to understand shorter paths than longer ones, so those will be printed first. It is not enough to just print the shortest path, though, since the program may have two or more unrelated errors.

Instead, let E be the set of all qualifier variables that trigger type errors. To eliminate derivative errors we require that, for each qualifier $Q \in E$, CQUAL prints out *at most one* path passing through Q. To see why this rule works, imagine a local variable that is used as both a *user* and *kernel* pointer. This variable causes a type error, and the error may spread to other variables through assignments, return statements, etc. When using our heuristic, these other, derivative errors will not be printed because they necessarily will have longer error paths. After printing the path of the original error, the qualifier variable with the type error will be marked, suppressing any extraneous error reports. Thus this heuristic has the additional benefit of selecting the error path that is most likely to highlight the actual programming bug that caused the er-

```
buf.win_info.handle: $kernel $user
   proto-noderef.cq:66     $kernel == _op_deref_arg1@66@1208
               cs.c:1208           == &win->magic
               cs.c:1199           == *win
               ds.c:809            == *pcmcia_get_first_window_arg1@809
               ds.c:809            == buf.win_info.handle
include/pcmcia/ds.h:76             == buf.win_info
               ds.c:716            == buf
               ds.c:748            == *cast
               ds.c:748            == *__generic_copy_from_user_arg1@748
               ds.c:748            == *__generic_copy_from_user_arg1
   proto-noderef.cq:27             == $user
```

Figure 3: The CQUAL error report for a bug in the PCMCIA system of Linux 2.4.5 through 2.6.0. We shortened file names for formatting. By convention, CQUAL type qualifiers all begin with "$".

ror. The heuristic will also clearly eliminate redundant errors since if two paths differ only in minor, inconsequential ways, they will still share some qualifier variable with a type error. In essence, our heuristic approximates the buckets of the ideal algorithm by using qualifier variables as buckets instead.

Before we implemented this heuristic, CQUAL often reported over 1000 type errors per file, in the kernel source files we analyzed. Now, CQUAL usually emits one or two error paths, and occasionally as many as 20. Furthermore, in our experience with CQUAL, this error reporting strategy accomplishes the main goals of the idealized algorithm described above: it reports just enough type errors to cover all the untypable statements in the original program.

4 Experiment Setup

We performed experiments with three separate goals. First, we wanted to verify that CQUAL is effective at finding user/kernel pointer bugs. Second, we wanted to demonstrate that our advanced type qualifier inference algorithms scale to huge programs like the Linux kernel. Third, we wanted to construct a Linux kernel provably free of user/kernel pointer bugs.

To begin, we annotated all the user pointer accessor functions and the dereference operator, as shown in Figure 2. We also annotated the kernel memory management routines, `kmalloc` and `kfree`, to indicate they return and accept *kernel* pointers. These annotations were not strictly necessary, but they are a good sanity check on our results. Since CQUAL ignores inline assembly code, we annotated several common functions implemented in pure assembly, such as `memset` and `strlen`. Finally, we annotated all the Linux system calls as accepting *user* arguments. There are 221 system calls in Linux 2.4.20, so these formed the bulk of our annotations. All told, we created 287 annotations. Adding all the annotations took about half a day. The extended version of this paper lists all the functions we annotated.

The Linux kernel can be configured with a variety of features and drivers. We used two different configurations in our experiments. In the file-by-file experiments we configured the kernel to enable as many drivers and features as possible. We call this the "full" configuration. For the whole-kernel analyses, we used the default configuration as shipped with kernels on kernel.org.

CQUAL can be used to perform two types of analyses: file-by-file or whole-program. A file-by-file analysis looks at each source file in isolation. As mentioned earlier, this type of analysis is not sound, but it is very convenient. A whole-program analysis is sound, but takes more time and memory. Some of our experiments are file-by-file and some are whole-program, depending on the goal of the experiment.

To validate CQUAL as a bug-finding tool we performed file-by-file analyses of Linux kernels 2.4.20 and 2.4.23 and recorded the number of bugs CQUAL found. We also analyzed the warning reports to determine what programmers can do to avoid false positives. Finally, we made a subjective evaluation of our error reporting heuristics to determine how effective they are at eliminating redundant warnings.

We chose to analyze each kernel source file in isolation because programmers depend on separate compilation,

Version	Configuration	Mode	Raw Warnings	Unique Warnings	Exploitable Bugs
2.4.20	Full	File	512	275	7
2.4.23	Full	File	571	264	6
2.4.23	Default	File	171	76	1
2.4.23	Default	Whole	227	53	4

Table 2: Experimental results. A full configuration enables as many drivers and features as possible. The default configuration is as shipped with kernels on kernel.org. A file-by-file analysis is unsound, but represents how programmers will actually use program auditing tools. A whole kernel analysis requires more resources, but is sound and can be used for software verification. The raw warning count is the total number of warnings emitted by CQUAL. We discovered in our experiments that many of these warnings were redundant, so the unique warning count more accurately represents the effort of investigating CQUAL's output.

so this model best approximates how programmers actually use static analysis tools in practice. As described in Section 3, analyzing one file at a time is not sound. To partially compensate for this, we disabled the subtyping relation *kernel* $<$ *user*. In the context of single-file analysis, disabling subtyping enables CQUAL to detect inconsistent use of pointers, which is likely to represent a programming error. The following example illustrates a common coding mistake in the Linux kernel:

```
void dev_ioctl(int cmd, char *p)
{
  char buf[10];
  if (cmd == 0)
    copy_from_user(buf, p, 10);
  else
    *p = 0;
}
```

The parameter, p, is not explicitly annotated as a *user* pointer, but it almost certainly is intended to be used as a *user* pointer, so dereferencing it in the "else" clause is probably a serious, exploitable bug. If we allow subtyping, i.e. if we assume *kernel* pointers can be used where *user* pointers are expected, then CQUAL will just conclude that p must be a *kernel* pointer. Since CQUAL doesn't see the entire kernel at once, it can't see that dev_ioctl is called with user pointers, so it can't detect the error. With subtyping disabled, CQUAL will enforce consistent usage of p: either always as a *user* pointer or always as a *kernel* pointer. The dev_ioctl function will therefor fail to typecheck.

In addition, we separately performed a whole kernel analysis on Linux kernel 2.4.23. We enabled subtyping for this experiment since, for whole kernel analyses, subtyping precisely captures the semantics of user and kernel pointers.

We had two goals with these whole-kernel experiments. First, we wanted to verify that CQUAL's new type qualifier inference algorithms scale to large programs, so we measured the time and memory used while performing the analysis. We then used the output of CQUAL to measure how difficult it would be to develop a version of the Linux kernel provably free of user/kernel pointer bugs. As we shall see, this study uncovered new research directions in automated security analysis.

5 Experimental Results

All our experimental results are summarized in Table 2.

Error reporting. We quickly noticed that although our error clustering algorithm substantially improved CQUAL's output, it still reported many redundant warning messages. Each warning is accompanied by an error path that explains the source of the user pointer and the line of code that dereferences it, as shown in Figure 3. Based on our experience reviewing the warnings, they can further be clustered by the line of code from which the user pointer originates. In our experiments, we performed this additional clustering (according to the source of the user pointer) manually. Table 2 presents both the raw and manually clustered warning counts. We refer only to the clustered error counts throughout the rest of this paper.

Bug finding with CQUAL. Our first experiment analyzed each source file separately in the full configuration of Linux kernel 2.4.20. CQUAL generated 275 unique warnings in 117 of the 2312 source files in this version of the kernel. Seven warnings corresponded to

```
 1: int i2cdev_ioctl (struct inode *inode, struct file *file, unsigned int cmd,
 2:                   unsigned long arg)
 3: {
 4: ...
 5:         case I2C_RDWR:
 6:                 if (copy_from_user(&rdwr_arg,
 7:                                    (struct i2c_rdwr_ioctl_data *)arg,
 8:                                    sizeof(rdwr_arg)))
 9:                         return -EFAULT;
10: ...
11:                 for( i=0; i<rdwr_arg.nmsgs; i++ )
12:                 {
13: ...
14:                         if(copy_from_user(rdwr_pa[i].buf,
15:                                 rdwr_arg.msgs[i].buf,
16:                                 rdwr_pa[i].len))
17:                         {
18:                                 res = -EFAULT;
19:                                 break;
20:                         }
21:                 }
22: ...
```

Figure 4: An example bug we found in Linux 2.4.20. The `arg` parameter is a *user* pointer. The bug is subtle because the expression `rdwr_arg.msgs[i].buf` on line 15 dereferences the *user* pointer `rdwr_arg.msgs`, but it looks safe since it is an argument to `copy_from_user`. Kernel developers had recently audited this code for user/kernel bugs when we found this error.

real bugs. Figure 4 shows one of the subtler bugs we found in 2.4.20. Kernel maintainers had fixed all but one of these bugs in Linux 2.4.22, and we confirmed the remaining bug with kernel developers. Because of this, we repeated the experiment when Linux kernel 2.4.23 became available.

When we performed the same experiment on Linux 2.4.23, CQual generated 264 unique warnings in 155 files. Six warnings were real bugs, and 258 were false positives. We have confirmed 4 of the bugs with kernel developers. Figure 5 shows a simple user/kernel bug that an adversary could easily exploit to gain root privileges or crash the system.

We also did a detailed analysis of the false positives generated in this experiment and attempted to change the kernel source code to eliminate the causes of the spurious warnings; see Section 6.

Scalability of Type Qualifier Inference. To verify the scalability of CQual's type inference algorithms, we performed a whole-kernel analysis on Linux kernel 2.4.23 with the default configuration. Since the default configuration includes support for only a subset of the drivers, this comprises about 700 source files containing 300KLOC. We ran the analysis on an 800MHz Itanium computer, and it required 10GB of RAM and 90 minutes to complete. Since CQual's data-structures consist almost entirely of pointers, it uses nearly twice as much memory on 64-bit computers as on 32-bit machines; also, 800MHz Itaniums are not very fast. Therefore we expect that CQual can analyze large programs on typical developer workstations in use today.

Software Verification. Finally, we took a first step towards developing an OS kernel that is provably free of user/kernel pointer bugs. We performed a brief review of the warnings generated during our whole-kernel analysis of Linux 2.4.23. This review uncovered an additional four bugs and a total of 49 unique false positives. We can draw two conclusions from this experiment. First, our error reporting algorithms may occasionally cause one bug to be masked by another bug or false positive. This is obvious from the fact that the bug discovered in our file-by-file analysis is not reported in the whole-program analysis. On the other hand, a whole-kernel analysis with CQual does not result in many more warnings

```
static int
w9968cf_do_ioctl(struct w9968cf_device* cam, unsigned cmd, void* arg)
{
...
    case VIDIOCGFBUF:
    {
            struct video_buffer* buffer = (struct video_buffer*)arg;

            memset(buffer, 0, sizeof(struct video_buffer));
```

Figure 5: A bug from Linux 2.4.23. Since `arg` is a *user* pointer, an attacker could easily exploit this bug to gain root privileges or crash the system.

than a file-by-file analysis. This suggests that we only need to reduce CQUAL's false positive rate by an order of magnitude to be able to develop a kernel provably free of user/kernel pointer bugs.

Observations. We can draw several conclusions from these experiments. First, type qualifier inference is an effective way of finding bugs in large software systems. All total, we found 17 different user/kernel bugs, several of which were present in many different versions of the Linux kernel and had presumably gone undiscovered for years.

Second, soundness matters. For example, Yang, et al. used their unsound bug-finding tool, MECA, to search for user/kernel bugs in Linux 2.5.63. We can't make a direct comparison between CQUAL and MECA since we didn't analyze 2.5.63. However, of the 10 bugs we found in Linux 2.4.23, 8 were still present in 2.5.63, so we can compare MECA and CQUAL on these 8 bugs. MECA missed 6 of these bugs, so while MECA is a valuable bug-finding tool, it cannot be trusted by security software developers to find all bugs.

Our attempt to create a verified version of Linux 2.4.23 suggests future research directions. The main obstacles to developing a verifiable kernel are false positives due to field unification and field updates, which are described in the extended version of this paper. A sound method for analyzing these programming idioms would open the door to verifiably secure operating systems.

Bugs and warnings are not distributed evenly throughout the kernel. Of the eleven bugs we found in Linux 2.4.23, all but two are in device drivers. Since there are about 1500KLOC in drivers and 700KLOC in the rest of the kernel, this represents a defect rate of about one bug per 200KLOC for driver code and about one bug per 400KLOC for the rest of the kernel. (Caveat: These numbers must be taken with a grain of salt, because the sample size is very small.) This suggests that the core kernel code is more carefully vetted than device driver code. On the other hand, the bugs we found are not just in "obscure" device drivers: we found four bugs in the core of the widely used PCMCIA driver subsystem. Warnings are also more common in drivers. In our file-by-file experiment with 2.4.23, 196 of the 264 unique warnings were in driver files.

Finally, we discovered a significant amount of bug turnover. Between Linux kernels 2.4.20 and 2.4.23, 7 user/kernel security bugs were fixed and 5 more introduced. This suggests that even stable, mature, slowly changing software systems may have large numbers of undiscovered security holes waiting to be exploited.

6 False Positives

We analyzed the false positives from our experiment with Linux kernel 2.4.23. This investigation serves two purposes.

First, since it is impossible to build a program verification tool that is simultaneously sound and complete,[5] any system for developing provably secure software must depend on both program analysis tools and programmer discipline. We propose two simple rules, based on our false positive analysis, that will help software developers write verifiably secure code.

Second, our false positive analysis can guide future reasearch in program verification tools. Our detailed classification shows tool developers the programming idioms that they will encounter in real code, and which ones are crucial for a precise and useful analysis.

Source	Frequency	Useful	Fix
User flag	50	Maybe	Pass two pointers instead of `from_user` flag
Address of array	24	Yes	Don't take address of arrays
Non-subtyping	20	No	Enable subtyping
C type misuse	19	Yes	Declare explicit, detailed types
Field unification	18	No	None
Field update	15	No	None
Open structure	5	Yes	Use C99 open structure support
Temporary variable	4	Yes	Don't re-use temporary variables
User-kernel assignment	3	Yes	Set `user` pointers to NULL instead
Device buffer access	2	Maybe	None
FS Tricks	2	Maybe	None

Table 3: The types of false positives CQUAL generated and the number of times each false positive occurred. We consider a false positive useful if it tends to indicate source code that could be simplified, clarified, or otherwise improved. Where possible, we list a simple rule for preventing each kind of false positive.

Our methodology was as follows. To determine the cause of each warning, we attempted to modify the kernel source code to eliminate the warning while preserving the functionality of the code. We kept careful notes on the nature of our changes, and their effect on CQUAL's output. Table 3 shows the different false positive sources we identified, the frequency with which they occurred, and whether each type of false positives tended to indicate code that could be simplified or made more robust. The total number of false positives here is less than 264 because fixing one false positive can eliminate several others simultaneously. The extended version of this paper explains each type of false positive, and how to avoid it, in detail.

Based on our experiences analyzing these false positives, we have developed two simple rules that can help future programmers write verifiably secure code. These rules are not specific to CQUAL. Following these rules should reduce the false positive rate of any data-flow oriented program analysis tool.

Rule 1 *Give separate names to separate logical entities.*

Rule 2 *Declare objects with C types that closely reflect their conceptual types.*

As an example of Rule 1, if a temporary variable sometimes holds a *user* pointer and sometimes holds *kernel* pointer, then replace it with two temporary variables, one for each logical use of the original variable. This will make the code clearer to other programmers and, with a recent compiler, will not use any additional memory. [6] Reusing temporary variables may have improved performance in the past, but now it just makes code more confusing and harder to verify automatically.

As an example of the second rule, if a variable is conceptually a pointer, then declare it as a pointer, not a `long` or `unsigned int`. We actually saw code that declared a local variable as an `unsigned long`, but cast it to a pointer *every time the variable was used.* This is an extreme example, but subtler applications of these rules are presented in the extended version of this paper.

Following these rules is easy and has almost no impact on performance, but can dramatically reduce the number of false positives that program analysis tools like CQUAL generate. From Table 3, kernel programmers could eliminate all but 37 of the false positives we saw (a factor of 4 reduction) by making a few simple changes to their code.

7 Related Work

CQUAL has been used to check security properties in programs before. Shankar, et al., used CQUAL to find format string bugs in security critical programs[11], and Zhang, et al., used CQUAL to verify the placement of authorization hooks in the Linux kernel[16]. Broadwell, et al. used CQUAL in their Scrash system for eliminating sensitive private data from crash reports[2]. Elsman, et al. used CQUAL to check many other non-security applications, such as Y2K bugs[4] and Foster, et al. checked correct use of garbage collected "`__init`" data in the Linux kernel[6].

Linus Torvalds' program checker, Sparse, also uses type qualifiers to find user/kernel pointer bugs[12]. Sparse doesn't support polymorphism or type inference, though, so programmers have to write hundreds or even thousands of annotations. Since Sparse requires programmers to write so many annotations before yielding any payoff, it has seen little use in the Linux kernel. As of kernel 2.6.0-test6, only 181 files contain Sparse user/kernel pointer annotations. Sparse also requires extensive use of type qualifier casts that render its results completely unsound. Before Sparse, programmers had to be careful to ensure their code was correct. After Sparse, programmers have to be careful that their casts are also correct. This is an improvement, but as we saw in Section 5, bugs can easily slip through.

Yang, et al. developed MECA[15], a program checking tool carefully designed to have a low false positive rate. They showed how to use MECA to find dozens of user-kernel pointer bugs in the Linux kernel. The essential difference between MECA and CQUAL is their perspective on false positives: MECA aims for a very low false positive, even at the cost of missing bugs, while CQUAL aims to detect all bugs, even at the cost of increasing the false positive rate. Thus, the designers of MECA ignored any C features they felt cause too many false positives, and consequently MECA is unsound: it makes no attempt to deal with pointer aliasing, and completely ignores multiply-indirected pointers. MECA uses many advanced program analysis features, such as flow-sensitivity and a limited form of predicated types. MECA can also be used for other kinds of security analyses and is not restricted to user/kernel bugs. This results in a great bug-finding tool, but MECA can not be relied upon to find all bugs. In comparison, CQUAL uses principled, semantic-based analysis techniques that are sound and that may prove a first step towards formal verification of the entire kernel, though CQUAL's false alarm rate is noticeably higher.

CQUAL only considers the data-flow in the program being analyzed, completely ignoring the control-flow aspects of the program. There are many other tools that are good at analyzing control-flow, but because the user/kernel property is primarily about data-flow, control-flow oriented tools are not a good match for finding user/kernel bugs. For instance, model checkers like MOPS[3], SLAM[1], and BLAST[8] look primarily at the control-flow structure of the program being analyzed and thus are excellent tools for verifying that security critical operations are performed in the right order, but they are incapable of reasoning about data values in the program. Conversely, it would be impossible to check ordering properties with CQUAL. Thus tools like CQUAL and MOPS complement each other.

There are several other ad-hoc bug-finding tools that use simple lexical and/or local analysis techniques. Examples include RATS[9], ITS4[13], and LCLint[5]. These tools are unsound, since they don't deal with pointer aliasing or any other deep structure of the program. Also, they tend to produce many false positives, since they don't support polymorphism, flow-sensitivity, or other advanced program analysis features.

8 Conclusion

We have shown that type qualifier inference is an effective technique for finding user/kernel bugs, but it has the potential to do much more. Because type qualifier inference is sound, it may lead to techniques for formally verifying the security properties of security critical software. We have also described several refinements to the basic type inference methodology. These refinements dramatically reduce the number of false positives generated by our type inference engine, CQUAL, enabling it to analyze complex software systems like the Linux kernel. We have also described a heuristic that improves error reports from CQUAL. All of our enhancements can be applied to other data-flow oriented program analysis tools. We have shown that formal software analysis methods can scale to large software systems. Finally, we have analyzed the false positives generated by CQUAL and developed simple rules programmers can follow to write verifiable code. These rules also apply to other program analysis tools.

Our research suggests many directions for future research. First, our false positive analysis highlights several shortcomings in current program analysis techniques. Advances in structure-handling would have a dramatic effect on the usability of current program analysis tools, and could enable the development of verified security software. Several of the classes of false positives derive from the flow-insensitive analysis we use. Adding flow-sensitivity may further reduce the false-positive rate, although no theory of simultaneously flow-, field- and context-sensitive type qualifiers currently exists. Alternatively, researchers could investigate alternative programming idioms that enable programmers to write clear code that is easy to verify correct. Currently, annotating the source code requires domain-specific knowledge, so some annotations may accidentally be omitted. Methods for checking or automatically deriving annotations could improve analysis results.

Our results on Linux 2.4.20 and 2.4.23 suggest that widely deployed, mature systems may have even more latent security holes than previously believed. With sound tools like CQUAL, researchers have a tool to measure the number of bugs in software. Statistics on bug counts in different software projects could identify development habits that produce exceptionally buggy or exceptionally secure software, and could help users evaluate the risks of deploying software.

Availability

The extended version of this paper is available from
`http://www.cs.berkeley.edu/~rtjohnso/`.

CQUAL is open source software hosted on SourceForge, and is available from
`http://www.cs.umd.edu/~jfoster/cqual/`.

Acknowledgements

We thank Jeff Foster for creating CQUAL and helping us use and improve it. We thank John Kodumal for implementing an early version of polymorphism in CQUAL and for helping us with the theory behind many of the improvements we made to CQUAL. We thank the anonyous reviewers for many helpful suggestions.

Notes

[1] In Linux, the system call `foo` is implemented in the kernel by a function `sys_foo`.

[2] In C, the *nonconst* qualifier is an implicit default.

[3] This is standard deductive inference notation. The notation

$$\frac{A_1 \quad A_2 \quad \cdots \quad A_n}{B}$$

means that, if $A_1, A_2, \ldots A_n$ are all true, then B is true.

[4] Although it's not important for this discussion, the definition of a valid path is given in the extended version of this paper.

[5] This is a corollary of Rice's Theorem.

[6] The variables can share the same stack slot.

References

[1] Thomas Ball and Sriram K. Rajamani. The SLAM Project: Debugging System Software via Static Analysis. In *Proceedings of the 29th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–3, Portland, Oregon, January 2002.

[2] Pete Broadwell, Matt Harren, and Naveen Sastry. Scrash: A System for Generating Secure Crash Information. In *Proceedings of the 12th Usenix Security Symposium*, Washington, DC, August 2003.

[3] Hao Chen and David Wagner. MOPS: an infrastructure for examining security properties of software. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 235–244, Washington, DC, November 18–22, 2002.

[4] Martin Elsman, Jeffrey S. Foster, and Alexander Aiken. Carillon—A System to Find Y2K Problems in C Programs, 1999. `http://bane.cs.berkeley.edu/carillon`.

[5] David Evans. *LCLint User's Guide*, February 1996.

[6] Jeff Foster, Rob Johnson, John Kodumal, and Alex Aiken. Flow-Insensitive Type Qualifiers. *ACM Transactions on Programming Languages and Systems*. Submitted for publication.

[7] Jeffrey Scott Foster. *Type Qualifiers: Lightweight Specifications to Improve Software Quality*. PhD thesis, University of California, Berkeley, December 2002.

[8] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy abstraction. In *Symposium on Principles of Programming Languages*, pages 58–70, 2002.

[9] Secure Software Inc. RATS download page. `http://www.securesw.com/auditing_tools_download.htm`.

[10] George Necula, Scott McPeak, and Westley Weimer. CCured: Type-Safe Retrofitting of Legacy Code. In *Proceedings of the 29th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 128–139, Portland, Oregon, January 2002.

[11] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting Format String Vulnerabilities with Type Qualifiers. In *Proceedings of*

the 10th Usenix Security Symposium, Washington, D.C., August 2001.

[12] Linus Torvalds. Managing kernel development, November 2003. `http://www.linuxjournal.com/article.php?sid=7272`.

[13] John Viega, J.T. Bloch, Tadayoshi Kohno, and Gary McGraw. ITS4: A Static Vulnerability Scanner for C and C++ Code. In *16th Annual Computer Security Applications Conference*, December 2000. `http://www.acsac.org`.

[14] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. In *Networking and Distributed System Security Symposium 2000*, San Diego, California, February 2000.

[15] Junfeng Yang, Ted Kremenek, Yichen Xie, and Dawson Engler. MECA: an extensible, expressive system and language for statically checking security properties. In *Proceedings of the 10th ACM conference on Computer and communication security*, pages 321–334. ACM Press, 2003.

[16] Xiaolan Zhang, Antony Edwards, and Trent Jaeger. Using CQUAL for Static Analysis of Authorization Hook Placement. In *Proceedings of the 11th Usenix Security Symposium*, San Francisco, CA, August 2002.