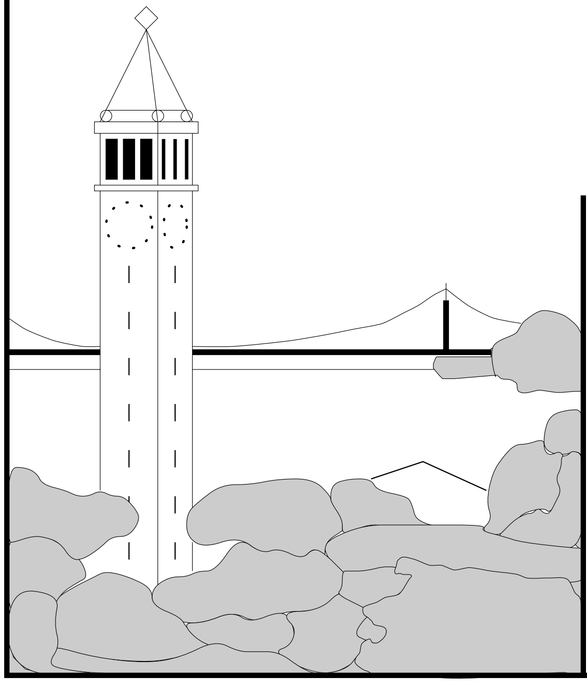
Finding User/Kernel Pointer Bugs With Type Inference

Rob Johnson *David Wagner*



Report No. UCB/CSD-04-1308

March 2, 2004

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Finding User/Kernel Pointer Bugs With Type Inference

Rob Johnson David Wagner

March 2, 2004

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 16 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. One common type of untrusted data in OS kernels that has been implicated in many security vulnerabilities includes pointers passed to system calls from user programs. 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	9

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 16 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 9 such bugs in Linux 2.4.23. Four bugs were common to 2.4.20 and

2.4.23, for a total of 16 different bugs, and eight of these 16 were still present in the 2.5 development series. We have confirmed all but one 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, because 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 16 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 data-flow oriented program analysis tool.
- We develop guidelines that programmers can follow to further reduce the number of false positives when using program verification tools.

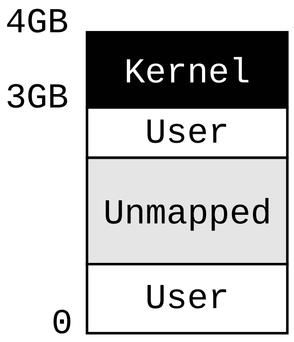


Figure 1: The Linux virtual memory layout on 32-bit architectures.

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.

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

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.

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 16 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.

```
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.

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 an error. Figure 2 also shows CQUAL's syntax for annotating built-in C operators. The `__op_deref` annotation prohibits dereferencing *user* pointers.

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 $A < B$).

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]

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

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.

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

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.

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 250 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[9], 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

[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.

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.

3.2 Our Analysis Refinements

We made several enhancements to CQUAL to support our user/kernel analysis. The challenge was to improve the analysis's 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 Appendix A 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.

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.

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).

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

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

```
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 "$".

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 error. 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 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 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. These annotations are given in Figure 8. We also annotated the kernel memory management routines, `kmalloc` and `kfree`, to indicate they return and accept *kernel* pointers. This annotation wasn't strictly necessary, but we felt it was a good sanity check on our results. 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 245 annotations. Adding all the annotations took about half a day.

To validate CQUAL as a bug-finding tool, we analyzed Linux kernels 2.4.20 and 2.4.23 and recorded the number of bugs CQUAL found. To estimate the programmer effort required to investigate CQUAL's error reports, we also recorded the number of files which had false positives. In our experience, this is the most useful metric for how long it will take the user to evaluate CQUAL's results.

We chose to analyze each kernel source file in isolation because programmers depend on separate compilation, 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, 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 2.4.5.[5] 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 this analysis to take a first step towards developing an OS kernel verifiably free of user/kernel bugs by fixing almost all the warnings CQUAL reported. The remaining warnings point out new research directions in automated security analysis.

[5]A minor bug in our tool chain prevented us from performing a whole kernel analysis on 2.4.20 or 2.4.23.

```
 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.

5 Experimental Results

Bug finding with CQUAL. Our first experiment analyzed each source file in Linux kernel 2.4.20 separately. CQUAL generated warnings for 117 of the 2312 source files in this version of the kernel. Seven warnings corresponded to real bugs, and 110 warnings were false positives. 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 warnings for 143 files. Five warnings were real bugs, 138 were false positives. We have confirmed 4 of the five 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. A discussion of our experience programming within the flow-insensitive type constraints enforced by CQUAL is also presented in 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 2.4.5 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 8.1GB of RAM and 80 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; consequently, on a 32-bit machine, we'd expect memory usage to be about 4GB of RAM. Also, 800MHz Itaniums are not very fast. Therefore we expect that CQUAL can analyze large programs on typical developer workstations in use today.

```
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.

Software Verification. Finally, we took a first step towards developing an OS kernel that is provably free of user/kernel pointer bugs. By making minor modifications to the kernel source code, we eliminated all but 136 warnings from our whole-kernel analysis of Linux 2.4.5. Based on this work, auditors could verify this kernel free of user/kernel bugs by manually checking the 136 remaining warnings. This is not yet entirely satisfactory, but it is an order of magnitude reduction in effort compared to manually checking the 129 system calls and thousands of device drivers that handle user pointers in Linux. Our analysis of the false positives from Linux 2.4.23 also suggest research directions for further reducing the false positive rate of tools like CQUAL. With another order of magnitude reduction in false positives, operating systems that are provably secure against user/kernel attacks will be within reach.

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 16 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 9 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 2.4.5 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 Appendix B. A sound method for analyzing these programming idioms would open the door to verifiably secure operating systems.

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,[6] 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.

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 pre-

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

Source	Frequency	Useful	Fix
User flag	48	Maybe	Pass two pointers instead of `from_user` flag
Address of array	24	Yes	Don't take address of arrays
C type misuse	17	Yes	Declare explicit, detailed types
Field unification	16	No	None
Field update	15	No	None
Non-subtyping	15	No	Enable subtyping
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

Table 2: 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.

serving the functionality of the code. We kept careful notes on the nature of our changes, and their effect on CQUAL's output. Table 2 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 more than 138 because some files had more than one false positive. Appendix B 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. [7] Reusing temporary variables may have improved performance in the past, but now it just makes code more confusing and harder to verify automatically.

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

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 Appendix B.

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 2, kernel programmers could eliminate all but 33 of the false positives we saw (a factor of 4.5 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[10], 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. Alternatively, researchers could investigate alternative programming idioms that enable programmers to write clear code that is easy to verify correct. 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

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.

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] 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.

[10] Inc. Secure Software. Rats download page. `http://www.securesw.com/auditing\_tools\_download.htm`.

[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.

```
 1:  void * helper(void *h)
 2:  {
 3:    assert (h != NULL);
 4:    return h;
 5:  }
 6:
 7:  int good_ioctl(void * user goodp)
 8:  {
 9:    char goodbuf[8];
10:    void *q = helper(goodp);
11:    void *b = helper(goodbuf);
12:
13:    copy_from_user(b, q, 8);
14:  }
```

Figure 6: `good_ioctl` doesn't have a user/kernel pointer bug, but a monomorphic analysis would report that it does.

A CQUAL Improvements

A.1 Context-Sensitivity

Previously, CQUAL's handling of function calls could introduce a large number of false positives, and users had to add numerous annotations to the source program to squelch these warnings. We have improved CQUAL by adding support for context-sensitivity, completely eliminating this source of false-positives and hence the need for extra annotations.

To understand the importance of context-sensitivity, consider the code in Figure 6. When the old, monomorphic version of CQUAL analyzes this code, it generates the type qualifier constraint graph shown in Figure 7 (ignore the labels on the edges for the moment). In the constraint graph, each node represents one type qualifier variable, and an edge $Q_1 \rightarrow Q_2$ means that $Q_1 \leq Q_2$. The nodes kto and $ufrom$ are from the first and second arguments to `copy_from_user` in Figure 2. Any path through the graph corresponds to a transitive sequence of typing constraints. For example, the graph clearly shows the sequence of constraints

$$\textsf{\textit{user}} \leq goodp \leq h \leq helper\_ret \leq b \leq to \leq \textsf{\textit{kernel}}$$

which imply a typing error, but `good_ioctl` doesn't contain any user/kernel pointer bugs. The confusion arises because the `helper` function processes both *user* and *kernel* pointers, but a monomorphic type inference engine cannot distinguish different calls to the same function. Thus, passing a *user* pointer into `helper` on

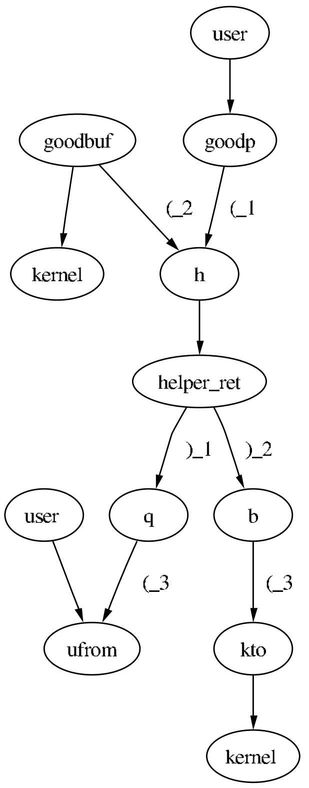


Figure 7: The constraint graph generated from the code in Figure 6. CQUAL prevents bad flow through the `helper` function by ignoring paths with mismatched parentheses. Note that not all the qualifier variables and constraints are shown.

line 10 induces a constraint on the return type of the unrelated call to `helper` on line 11.

To fix this, we add labels to the edges in the type qualifier constraint graph as shown in Figure 7. These labels restrict the set of valid paths though the graph by matching up function calls and returns, because we don't consider paths through the graph with mismatched parentheses. Formally, we require paths to satisfy the following rule:

Rule 3 *A path in the constraint graph is valid if the string of parentheses along the path is a substring of some string of matched parentheses.*

In other words, the path may not have *mis*matched parentheses, but it may have *un*matched parentheses. We say that a qualifier node, Q, in the graph has a type error if there exist qualifiers Q_1 and Q_2 such that the user-defined type qualifier system specifies that $Q_1 \not\leq Q_2$ and there exists a valid path from Q_1 to Q and a valid path from Q to Q_2.

So we have reduced the type inference problem to the problem of checking if there exists a path, subject to Rule 3, from qualifier Q_1 to Q_2 where the user-defined type qualifier system specifies that $Q_1 \not\leq Q_2$. This is an instance of the more general CFL reachability problem. In this case, the reachability query can be answered in time linear in the size of the graph. A discussion of CFL reachability algorithms is outside the scope of this paper.

A.2 Structures

Prior versions of CQUAL were also field-insensitive, which created false positives when analyzing programs that use structures intensively. OS kernels are heavy users of structures, so it is critical that CQUAL be able to distinguish fields of different structures if it is to give good results on problems like user/kernel pointer bugs.

Early experiments showed that full field-sensitivity requires too much memory to be practical. Thus we chose a hybrid approach that preserves soundness, guarantees linear memory usage, and significantly decreases the false positive rate. CQUAL treats structures and unions identically, so all these improvements apply to unions, as well, but for simplicity we only discuss structures below. A small example illustrates the problem.

```
void sys_foo (char * user p)
{
  struct foo x;
  struct foo y;
  x.a = p;
  *(y.a) = 0;
}
```

In a field-insensitive type qualifier inference analysis, one qualified type is assigned to all the "a" fields of all instances of `struct foo`. Thus the type qualifier constraint graph contains the path

$$\textit{user} \rightarrow p \rightarrow a \rightarrow \textit{kernel}.$$

The type qualifier node a applies to both `x.a` and `y.a`, creating the false positive. To eliminate this false positive, we need to disambiguate the fields of different structure instances.

A naive solution to this problem is to simply create separate qualifier variables for the fields of each structure instance. This certainly solves the problem, but it can require memory exponential in the size of the input program, as the following example illustrates

```
struct a { int      x; int      y; };
struct b { struct a x; struct a y; };
struct c { struct b x; struct b y; };
struct c s;
```

We now have to create qualifiers for

```
s.a.a.a  s.a.a.b  s.a.b.a  s.a.b.b
s.b.a.a  s.b.a.b  s.b.b.a  s.b.b.b
```

This wouldn't be a problem if real programs didn't declare deeply nested structures like the above, but they do. OS kernels contain hundreds of large, complex, interconnected data structures that get passed to hundreds, if not thousands, of functions. They don't exhibit the exponential worst case described above, but they are complex enough to make the naive approach completely impractical.

We enhanced CQUAL's structure handling in two ways. First, the fields of different instances of a structure are given separate qualified types. This eliminates many of the false positives, as above. To control memory usage, we create the qualifiers for each field on demand the first time that field is referenced. Since the number of fields referenced by a program must be linear in the size of the program, this ensures that the number of qualifiers created by CQUAL is linear in the size of the input program.

Seocnd, upon analyzing an assignment statement like `x = y`, where `x` and `y` are structures, we unify the types of `x` and `y`, including all their fields. From then on, referencing a field of `x` is equivalent to referencing the corresponding field of `y`. This approach is sound, and it guarantees at worst linear space-complexity.

Because of this trade-off, our approach currently provides no subtyping or polymorphism on structure fields. In our experience, the lack of subtyping on structure fields is of little consequence because most structures are accessed through pointers. Thus, as we saw in Section 3, the type system would require equating the types of structure fields anyway. The lack of polymorphism for structure fields is also relatively benign, but as described in Section 6, when it does cause a false positive, it can be quite difficult to work around.

A.3 Well-formedness Constraints

Old versions of CQUAL only supported one set of semantics for their type qualifier systems, but the user/kernel application requires some non-standard typing rules. These non-standard rules center around the typing relations between structures and their fields and pointers and their referents. A type, τ, that satisfies these requirements is called *well-formed*, written $\vdash_{wf} \tau$. We added support to CQUAL for optionally enforcing well-formedness constraints. Although this feature affects CQUAL's structure typing rules, it is independent of the enhanced structure handling described in the previous section.

The following code illustrates the need for well-formed types:

```
void ioctl(void * user arg)
{
  struct cmd { char* datap; } c;
  copy_from_user(&c, arg, sizeof(c));
  c.datap[0] = 0;
}
```

The annotation for `copy_from_user` in Figure 2 implies that `c` receives a type of the form *user* `struct cmd`, which makes intuitive sense since the structure's contents are copied from user space. In the standard type qualifier semantics, though, the type qualifier on the structure is completely unrelated to the type qualifier on a field field, such as `c.datap`. Thus the dangerous user pointer dereference, `c.datap[0]`, doesn't create a typing error. This means that our basic type system fails to capture a certain class of user/pointer bugs, an obvious shortcoming. To repair this shortcoming, the typing rules need to take into account the relationship between a structure and its contents.

To fix this, we allow qualifiers both on the structure itself as well as on all the fields, and then we introduced *well-formedness constraints* that relate these qualifiers. For instance, in the above example, the variable `c` might receive the type *user* `struct { char *` *kernel* `datap; }`, and this violates the intuitive constraint that if the contents of the structure are under user control, then the value of all of its fields are also under user control.

More formally, this relationship may be expressed by the well-formedness constraint

$$\frac{\forall i \qquad Q \le Q_i \qquad \vdash_{wf} \tau_i}{\vdash_{wf}\ Q\ \texttt{struct}\,(Q_1\,\tau_1, \ldots, Q_n\,\tau_n)}.$$

This rule states that, if $Q \le Q_i$ and if the types τ_i are all well-formed, then the structure with field types $(Q_1\,\tau_1, \ldots, Q_n\,\tau_n)$ and qualifier Q is well-formed. In the implementation, this rule results in the addition of edges from the type qualifier on every structure instance to the qualifiers on each of its fields.

A similar issue arises with pointers. When a pointer is passed from user space, the value it points to is also under user control. The well-formedness constraint

$$\frac{Q \le Q' \qquad \vdash_{wf} \tau}{\vdash_{wf}\ Q\ \texttt{ref}\,(Q'\,\tau)}.$$

captures this rule. Again, the implementation of this rule simply requires adding type qualifier constraint edges from the qualifier on every pointer to the qualifier of its referent.

In our implementation, the application of these rules can be controlled very precisely by the user of CQUAL. For example, in our experiments with the Linux kernel, the two rules given here were enabled only for the the user/kernel analysis. No other qualifiers, including *const*, were affected by these rules. This flexibility is implemented by storing a bit-mask with each constraint edge indicating the qualifiers to which it applies.

A.4 Integer/Pointer Casts

The C code

```
char **p = ...;
int x = (int)p;
```

induces the qualified type constraint

$$p\ \texttt{ref}\,(p'\ \texttt{ref}\,(p''\ \texttt{char})) \le x\ \texttt{int}$$

From this, we know that $p \le x$, but what about p' and p''? Previous versions of CQUAL "collapsed" the type of p by adding the constraints $p'' = p' = p$. This caused many false positives, and it isn't even sound.

Both these problems can be solved by treating every `int` as an implicit `void *`. Then the type constraint becomes

$$p\ \texttt{ref}\,(p'\ \texttt{ref}\,(p''\ \texttt{char})) \le x\ \texttt{ref}\,(x'\ \texttt{void})$$

This reduces to the qualifier constraints $p \leq x$ and $p'' = p' = x'$. This approach still requires collapsing part of the type of p, but it is obviously more precise and, as a bonus, sound.

B False Positive Details

User Flag. Several subsystems in the Linux kernel pass around pointers along with a flag indicating whether the pointer is a user pointer or a kernel pointer. These functions typically look something like

```
void tty_write(void *p,
               int from_user)
{
  char buf[8];
  if (from_user)
    copy_from_user(buf, p, 8);
  else
    memcpy(buf, p, 8);
}
```

Since p is used inconsistently, CQUAL cannot assign a type to p, and hence generates a typing error. The type of p depends on the value of from_user. This idiom, where the value of one variable indicates the type of another, appears in all kinds of code, not just OS kernels, and programmers can easily avoid it. One way to make this code type-safe is to recognize that p serves two different logical roles, so we can convert the program to have two pointers as follows:

```
void tty_write(void *kp, void *up,
               int from_user)
{
  char buf[8];
  if (from_user)
    copy_from_user(buf, up, 8);
  else
    memcpy(buf, kp, 8);
}
```

Now from_user does not indicate the type of another argument to the function. Instead it indicates which argument to use. Note that the from_user flag could be eliminated by testing for up != NULL instead.

Programmers can also fix this problem by viewing it as a lack of context-sensitivity: the type of p depends on the calling context. CQUAL supports context-sensitivity, so we just need to find a way to exploit it. The solution is to encode the accesses to p in the arguments to tty_write:

```
typedef int (*copyfunc)(void *to,
                        void *from,
                        int len);
void tty_write(void *p, copyfunc cp)
{
  char buf[8];
  cp(buf, p, 8);
}
```

Programmers can now call either

```
tty_write(user_pointer, copy_from_user);
tty_write(kernel_pointer, memcpy);.
```

A type inference engine like CQUAL can verify that the arguments are never confused or misused.

Address of Array. In C, the following two code fragments accomplish the same thing:

```
char A[10];
memcpy(A, ...);

char A[10];
memcpy(&A, ...);
```

These two code fragments give the same result because &A is the same as A, i.e. these expressions have the same value. The two expressions have different types, though, and CQUAL is careful to distinguish the types, which can generate false positives when &A is used. The expression &A has type Q_1^* ref $(Q_1$ array $(Q_1'$ char$))$. When this gets coerced to Q_2 ref $(Q_2'$ char$)$ in the call to memcpy, there's an extra level in the type. CQUAL applies the standard type collapsing rule, identifying Q_1 and Q_1'. This can easily lead to false positives.

We could easy modify CQUAL to avoid this source of false positives, but after some thought, we decided that using &A makes code unnecessarily brittle, so programmers just shouldn't use it. This code works because, for arrays, &A=A. If the developer ever changes the declaration of A to "char *A" (so she can dynamically allocate A, for example), then &A and A will differ, and thus memcpy(&A,...) will break. Similarly, if the programmer decides to pass A as a parameter to func, then A will behave as a pointer, also breaking uses of &A.

Because taking the address of an array is so brittle and completely unnecessary, we recommend just not doing it.

C type misuse. Examples of this source of false positives take one of two forms: variables declared with a type that doesn't reflect how they are actually used and variables declared with very little type structure at all. The `long` vs. pointer example given above demonstrates the first form of type misuse, but sometimes programmers provide almost no type information at all. For example, several kernel device drivers would assemble command messages on the stack. These messages had a well-defined format, but there was no corresponding message data structure in the source code. Instead, the messages were assembled in simple `char` arrays:

```
void makemsg(char *buf)
{
  char msg[10];
  msg[0] = READ_REGISTER;
  msg[1] = 5;
  msg[2] = buf;
  ...
```

The following code is not only easier to typecheck, it's much easier to understand: [8]

```
void makemsg(char *buf)
{
  struct msg m;
  m.command   = READ_REGISTER;
  m.register  = 5;
  m.resultbuf = buf;
  ...
```

Declaring program variables with complete and correct types helps both programmers and program analysis tools.

Field Update. Since CQUAL is flow-insensitive, structure fields cannot be updated with values of two different types. The problem occurs most often with code like this:

```
struct msg {
  int type;
  void *body;
}
void msg_from_user(struct msg *m)
{
  struct msg km;
  void *t;
  copy_from_user(&km, m, ...);
  t = km.body;
  km.body = kmalloc(100);
  copy_from_user(km.body, t, ...);
}
```

[8]The developer must declare `struct msg` as "packed" to ensure equivalent behavior. Both gcc and Microsoft Visual C++ support packed structures.

From the initial `copy_from_user`, CQUAL infers that `km` is under user control, and hence `km.body` is a *user* pointer. When `km.body` is updated with a pointer returned by `kmalloc`, it becomes a *kernel* pointer, but a flow-insensitive type-system can only assign one type to `km.body`. Thus there is a type error.

We don't have a good way to program around this source of false positives. This problem can occur whenever one structure instance has a field that serves two conceptual roles. For existing code, fixing this false positive can be a challenge. The approach we used is to copy all the non-updated fields to a new structure instance, and initialize the updated field in the new structure instance instead of updating the field in the original instance. This doesn't produce easily maintained code, since every time a field is added to the structure, the code must be updated to match:

```
struct msg {
  int type;
  void *body;
}
void msg_from_user(struct msg *m)
{
  struct msg tm, km;
  void *t;
  copy_from_user(&tm, m, ...);
  km.type = tm.type;
  // If struct msg had more fields
  // copy those, too.
  km.body = kmalloc(100);
  copy_from_user(km.body, tm.body, ...);
}
```

For new programs, if there is only one field that is used for two different logical purposes, then the code maintainance problem above can be avoided by packaging the rest of the fields in one easily copied sub-structure, like this:

```
struct msg {
  struct {
    int type;
  } md;
  void *body;
}
void msg_from_user(struct msg *m)
{
  struct msg km;
  void *t;
  copy_from_user(&km.md, m, ...);
  copy_from_user(&t, &m->body, ...);
  km.body = kmalloc(100);
  copy_from_user(km.body, t, ...);
}
```

Neither of these solutions is completely satisfactory. We leave it as an open problem to develop simple coding conventions that avoid this type of false positive.

Field Unification. As described in Section A.2, CQUAL uses unification for fields of structures in order to ensure that memory usage is linear. The downside of the this decision is that unification can generate false positives. This is the only source of false positives that we feel is both specific to CQUAL and not useful to the programmer. We hope to find some way to improve CQUAL's handling of structures in the future.

Non-subtyping. CQUAL supports subtyping, but we decided not to use it in our experiments so that we could detect inconsistent uses of pointers without performing a whole-kernel analysis. Since we were checking for a stricter policy than is actually required, this caused a few false positives.

For program properties that genuinely don't need subtyping, this source of false positives will not exist. If an application does require subtyping, we can suggest two alternatives. For small to medium programs, simply turn on subtyping and perform a whole-program analysis. For large programs, thoroughly annotating the interfaces between different program modules will enable a sound analysis in the presence of subtyping without having to perform a whole-program analysis. These annotations will also provide additional documentation to new programmers using those interfaces.

Open Structures. An open structure is a structure with a variable-sized array immediately following it. Such structures are often used for network messages with a header and some variable number of bytes following it. Before the C99 standard, gcc had a custom extension to the C language to support this feature:

```
struct msg {
  int len;
  char buf[0];
};
void func(void)
{
  struct msg *m;
  m = kmalloc(sizeof(*msg) + 10);
}
```

The C99 standard now includes this extension with a slightly different syntax. Despite the relative maturity of this C extension, several kernel programmers have created their own open structures as follows:

```
struct msg {
  int len;
  char *data;
};
void func(void)
{
  struct msg *m;
  m = kmalloc(sizeof(*msg) + 10);
  m->data = (char*)(m+1);
}
```

Since this method for creating an open structure doesn't provide a separate name for the buffer following the header, a type inference engine must assign the same type to the structure head as to the data that follows. By giving it a separate name, this problem can be avoided. Declaring open structures properly also has the advantage of being simpler and easier to understand.

Temporary Variables. Programmers can fix false positives caused by reuse of temporary variables by using two temporary variables instead.

User-kernel Assignment. Several kernel drivers used the following idiom:

```
copy_from_user(kp, up, ...);
up = kp;
```

Sometimes, `up` is later used as a temporary variable, but most of the time the assignment is just a safety net to make future accidental references to `up` safe. In either case, it's easy to eliminate the assignment, or change it to `up = NULL`, to eliminate the false positive.

Device Buffer Access. A few device drivers read and write volatile device buffers. These buffers may have a high level structure, but the drivers treat them as flat buffers, reading and writing to device specific offsets. Thus the problem is similar to the C type misuse example above, where drivers construct control messages in unstructured buffers. Here, we have the added complexity of device-specific semantics for these buffers. Since these drivers depend on the behaviour of the device in question, it is impossible for any program analysis tool to verify that these are correct without knowledge of the devices being controlled.

```
unsigned long copy_from_user(void $user * $kernel to, const void   * $user from,
                             unsigned long n);
unsigned long __copy_from_user(void $user * $kernel to, const void   * $user from,
                               unsigned long n);
unsigned long __copy_from_user_ll(void $user * $kernel to, const void   * $user from,
                                  unsigned long n);

long __copy_to_user_ll(void   * $user to, const void * $kernel from,
                       unsigned long n);
unsigned long __copy_to_user(void * $user to, const void * $kernel from,
                             unsigned long n);
unsigned long copy_to_user(void  * $user to, const void * $kernel from,
                           unsigned long n);

unsigned long __generic_copy_from_user_nocheck(void $user * $kernel to,
                                               const void *$user from,
                                               unsigned long n);
unsigned long __generic_copy_to_user_nocheck(void *$user to,
                                             const void *$kernel from,
                                             unsigned long n);
unsigned long  __generic_copy_to_user(void *$user to,
                                      const void *$kernel from,
                                      unsigned long);
unsigned long  __generic_copy_from_user(void $user * $kernel to,
                                        const void *$user from,
                                        unsigned long);

unsigned long __constant_copy_to_user(void *$user to, const void *$kernel from,
                                      unsigned long n);
unsigned long __constant_copy_from_user(void $user * $kernel to,
                                        const void *$user from,
                                        unsigned long n);
unsigned long __constant_copy_to_user_nocheck(void *$user to,
                                              const void *$kernel from,
                                              unsigned long n);
unsigned long __constant_copy_from_user_nocheck(void $user * $kernel to,
                                                const void *$user from,
                                                unsigned long n);

long strncpy_from_user(char $user * $kernel dst, const char *$user src,
                       long count);
long __strncpy_from_user(char $user * $kernel dst, const char *$user src,
                         long count);
long strnlen_user(const char   * $user str, long n);

unsigned long clear_user(void *$user mem, unsigned long len);
unsigned long __clear_user(void *$user mem, unsigned long len);

$$a _op_deref ($$a * $_1 $kernel x) $_1_2;
```

Figure 8: Annotations for the user space access functions and `__op_deref`.