Verifying Security Properties using Type-Qualifier Inference

by

Robert Timothy Johnson

B.S. (University of North Carolina at Greensboro) 1998

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

Doctor of Philosophy
in

Engineering - Electrical Engineering and Computer Sciences

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:

Professor David Wagner, Chair
Professor Doug Tygar
Professor Lior Pachter

Fall 2006

The dissertation of Robert Timothy Johnson is approved.

Chair Date

Date

Date

University of California, Berkeley

Fall 2006

Verifying Security Properties using Type-Qualifier Inference

Copyright © 2006

by

Robert Timothy Johnson

Abstract

Verifying Security Properties using Type-Qualifier Inference

by

Robert Timothy Johnson

Doctor of Philosophy in Engineering - Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor David Wagner, Chair

Almost all software must accept input from untrustworthy sources: web servers, file servers, databases, and web browsers all read data from a network that is usually shared with untrusted, and often malicious, third parties. Even in the absence of network threats, untrusted inputs are still a problem. Operating system kernels receive untrusted data as arguments to system calls and windowing systems handle requests from clients they may not trust and that may not trust eachother. Code that handles these untrusted inputs is a natural avenue for attack because any bugs in that code may lead to a critical vulnerability.

In this dissertation, we describe new static analysis techniques for verifying that software safely handles its untrusted inputs. Static analysis is performed at compile time, so bugs are caught during the development cycle, before the software is deployed. Our approach is based on type-qualifier inference, a lightweight software verification technique that has many practical benefits. Type-qualifier inference is sound, which means that, up to certain limitations of the C programming language, if our verification tool reports that a program is vulnerability-free, then it really is immune to certain classes of attack.

We have implemented and tested our algorithms in CQual, a type-qualifier inference tool originally developed by Jeff Foster, et al. We then used our analysis tool to find dozens of bugs in the Linux kernel and to build the best format-string bug detector to

date. Analysis of our experimental results revealed two simple rules for writing easy-to-analyze code and several problems that future research in static security analysis tools should address.

Professor David Wagner
Dissertation Committee Chair

Contents

Contents i

List of Figures iii

List of Tables v

Acknowledgements vi

1 Introduction 1

2 Input-Validation Bugs 7

2.1 User/kernel Pointer Bugs . . . 8

2.2 Format-String Bugs . . . 13

3 Type-Qualifier Inference 19

3.1 Assertions, Annotations, and Type Checking . . . 21

3.2 Inference . . . 25

3.3 Semantics and Soundness . . . 29

4 Refinements 33

4.1 Parametric Polymorphism . . . 33

4.2 Structures . . . 47

4.3 Type Casts . . . 52

4.4 Multiple Files . . . 54

4.5 Presenting Qualifier Inference Results . . . 59

5 Evaluation 62

5.1 Linux kernel experiments . . . 62

5.1.1 Bug-finding . . . 64
5.1.2 Scalability of Type-Qualifier Inference. . . . 68
5.2 Format-String Experiments . . . 70
5.3 Linux Kernel False Positives . . . 72
5.4 False Positive Details . . . 74

6 Related Work **82**
6.1 Flow-Insensitive Type Qualifiers . . . 82
6.2 Error Detection and Prevention Systems . . . 84
6.3 Static Analysis and Security . . . 86
6.4 Algorithmics . . . 87

7 Conclusion **89**

Bibliography **90**

List of Figures

2.1 The Linux virtual memory layout on 32-bit architectures. 9
2.2 Stack layout of a call to `printf` 15
2.3 A function vulnerable to format-string attacks 16
2.4 Type-qualifier annotations on `sayhello` and `snprintf` 17

3.1 Source Language 20
3.2 Standard Type Checking System 20
3.3 Example Qualifier Partial Order 21
3.4 Qualified-Type Checking System 22
3.5 Qualified-Type Inference System 26
3.6 Constraint Resolution 28
3.7 Values for Big-Step Semantics 30
3.8 Big-Step Operational Semantics with Qualifiers 32

4.1 Polymorphic Constraint Graph Construction 35
4.2 Context-free Grammars for MP, CP, OP, and NMP. 39
4.3 CP-reachability closure rules 41
4.4 CP reachability algorithm 42
4.5 Qualified-Type Checking System Extended for Structures 49
4.6 Efficient representation for structure types 50
4.7 A C program demonstrating the imprecision of field unification. 51
4.8 Qualified-Type Checking System Extended for Void Pointers 53
4.9 Qualifier Constraint Graph Compaction Algorithm 56
4.10 Local Compaction Rules 57
4.11 Tainting Analysis Error Filtering Results 61

5.1 Annotations for the basic user space access functions in Linux 63
5.2 A user/kernel pointer bug in Linux 2.4.20 66
5.3 A user/kernel pointer bug in Linux 2.4.23 67

List of Tables

4.1 Compaction performance results 58

5.1 Linux kernel bug-finding results 65
5.2 Linux kernel scalability results 68
5.3 Format-string bug experimental results 71
5.4 False positive causes 72

Acknowledgements

This research would not have been possible without the brilliant and patient guidance of my advisor, David Wagner. He has worked tirelessly to support and aid this research and to give me every possible resource necessary to complete it. I am very lucky to have had him for my advisor.

Jeff Foster wrote the original version of CQual upon which all this work is based, and has helped me tremendously in developing new ideas and features for this project. John Kodumal implemented an early version of the polymorphic analysis, and has also provided many useful suggestions. Ben Liblit patiently answered all my questions on program analysis. Alex Aiken's suggestions during my qualifying exam were extremely helpful. Umesh Shankar helped me get started in this research by sharing his experiences with his project, Percent-S. Hao Chen was also happy to share his experiences writing another program analysis tool, MOPS. Naveen Sastry has been an enthusiastic supporter and has helped me keep the "big picture" in mind. David Gay wrote the original frontend used in CQual, and I've enjoyed our conversations about our respective research projects. Lior Pachter has been a great committee-member and was a pleasure to teach with.

Much of this dissertation has been published in conferences and journals, and I thank the anonymous reviewers for their comments. Part of this research was supported by a grant from the National Science Foundation, for which I am very grateful.

Monica Chew, my wife while working on this research and still a dear friend, has been extremely supportive and always insightful.

Finally, Tong has shown incredible patience and given valuable encouragement while I finished this dissertation, for which I will always be in her debt.

Chapter 1

Introduction

Software bugs are cause a tremendous number of security vulnerabilities and breakins today[1]. Hackers exploit bugs to break into remote machines, vandalize websites, steal secret information, such as customer credit cards, write worms and viruses, and even to launch floods of spam. The United States Treasury Department estimated that, in 2004, criminals netted over $100 billion from cybercrime[28].

Programmers are not good at writing software without such vulnerabilities, nor are they good at auditing code for security bugs. Many programmers simply do not know all the rules of writing secure programs, which is understandable when guides to secure programming exceed 150 pages[89]. The rules for writing secure programs are often obscure and baroque, having developed over many years as imperfectly designed operating systems were expanded and patched to support new features and address old security flaws[10, 14]. Even when programmers understand the rules and attempt to apply them correctly, mistakes still creep in. A single bug can span dozens of source files and hundreds of lines of code, making manual detection exceedingly difficult.

As computers become more network-centric the trusted computing base has expanded dramatically, forcing security-naive applications programmers to deal with new issues with which they have little familiarity and which do not directly advance their goal of developing new features. Just over a decade ago, computer security was primarily focused on protecting

the operating system from untrusted users, so only the core components of the OS were part of the trusted computing base. With the rise of network-centric personal computers, security goals have grown to encompass everything from preventing spam and phishing to stopping worms and viruses. As a consequence, applications such as word processors, spreadsheets and graphics programs are enforcing their own application-specific security policies, and bugs, such as buffer overflows or format-string errors, in these programs can lead to serious system compromise. As a result, much more code needs to be audited for security and the programmers who write that code are often less security-adept.

Automated software-verification tools can improve the state of the art. Since the code-auditing task has outgrown programmers' resources and abilities, we need new tools to help them find bugs quickly and with less effort. Automated audit tools can offer several advantages:

- Tools can include a built-in suite of checks, so programmers don't need to read and remember all the rules of secure coding. Instead, the tool can check those rules automatically.
- A code-analysis tool can perform audits automatically during the development process, catching bugs early, when they are less expensive to fix.
- By eliminating bugs during product development, software makers can avoid issuing patches later on. This also saves end-users, such as system administrators, money, as they often perform extensive regression tests before installing security patches.

This dissertation describes a code-auditing algorithm and tool focused primarily on tracking the flow of data inside programs. In large programs, data flows from inputs, through internal data structures, and to outputs via numerous function call interfaces and across source module boundaries, making manual tracking of data-flow exceedingly difficult, yet the origins and destinations of data can have a large impact on the security of a program.

- Buffer overflows occur when data from an untrusted source flows into a buffer that is not large enough to hold it.

- Format-string bugs occur when a string from an untrusted source flows to the format argument of a printf-style function.

- SQL-injection bugs allow data from untrusted sources to flow into the SQL commands the program issues to its database engine.

- User/kernel pointer bugs occur when a pointer argument passed into a system call flows to the site of a memory reference.

- Cross-site scripting bugs occur when client input flows through a web server to its outputs without passing through an appropriate sanitization function.

Data-flow is so fundamental because, to exploit most software bugs, an attacker must provide some sort of malicious input to a vulnerable program, and this input must flow to a location in the program that is not prepared to deal with it.

The algorithms presented here build on type-qualifier inference, a program-analysis technique developed by the programming-languages community[35, 37, 34]. Type qualifiers are a good foundation for a security-auditing tool for several reasons:

- Type qualifiers are easy for programmers to use and understand. The C programming language already contains qualifiers, such as *const* and *restrict*, so programmers have an intuitive understanding of how qualifiers work. This will ease adoption and reduce the errors introduced by incorrect annotations.

- Type qualifiers have a sound theoretical foundation. When applied correctly to a C program that type-checks (in the basic C type system), we can prove that the analysis will not miss any inconsistent uses of qualifiers. We can thus guarantee that, if the qualifiers are set up such that any bug will result in a qualifier inconsistency, then the analysis will catch all the bugs. This enables type-qualifier inference to act as a form of lightweight software verification, yielding software with provable security properties.

- Type-qualifier inference is efficient. In our experiments, we perform type-qualifier inference on very large pieces of software, and the computation required is comparable

to the time required to compile the same software. Asymptotically, type-qualifier inference is nearly linear in the size of the program being analyzed.

- Type qualifiers can be usefully retrofit into many existing languages. This dissertation describes CQual, a tool for performing type-qualifier inference on C programs, but the same algorithms could be applied to C++, Java, C#, or any other language with a static type system.

- Type-qualifier inference requires few annotations from the programmer, enabling him to get useful results with little effort. In our experiments, we found dozens of format-string and user/kernel pointer bugs in millions of lines of code, but only had to write a few hundred annotations.

- Type-qualifier inference produces error paths that illustrate the potential bug to the programmer, helping him understand and fix the error.

- Type-qualifier inference is precise. Many static analysis tools, including CQual, model the analyzed program conservatively, which can lead to false positives, i.e. warnings that do not correspond to any real bug. Such warnings waste developer's time and reduce the usefulness of the tool. Our experiments show that, with the enhancements described in this dissertation, CQual can find format-string bugs in C programs with a false-positive rate of about 40%. More extensive experiments by Chen, et al[15], have found the false-positive rate to be much lower – under 15%.

Type-qualifier inference provides a solid foundation for program analysis, but a naive implementation has many shortcomings. We had to overcome numerous challenges to get good results:

- Earlier versions of CQual used a monomorphic analysis, which models data-flow through functions too conservatively. We implemented a state-of-the-art polymorphic analysis, greatly improving the precision of the type-qualifier inference engine. In some experiments, this gave an order-of-magnitude reduction in false positives.

- To support efficient polymorphic analysis of C programs, we extend the linear-time matched-parenthesis graph reachability algorithm of Horwitz, Reps, and Sagiv[46] to handle graphs derived from programs with global variables in linear time.

- The C programming language supports complex data types, such as `struct`s and `union`s, and precisely modeling these features can be crucial to getting good results, as our experiments on finding user/kernel pointer bugs shows. To address this problem, we developed a new approach to modeling structures that is both efficient and precise.

- Although type-qualifier inference is sound when applied to a program that contains no casts and type-checks in the C type system, many C programs bend the C type system through casts. In this dissertation, we develop new ways to handle C idioms, such as casts between pointers and `int`s, that simultaneously improve the precision of the analysis and make it sound for many C programs that use casts in idiomatic ways.

- Type-qualifier inference is memory intensive when applied to large programs. We modified the inference algorithm to work incrementally and produce compact summaries of intermediate results, enabling it to scale to large programs efficiently.

- A single programming error can lead to thousands of type errors, so presenting useful feedback to the programmer is a challenge. We describe new heuristics for clustering type errors together based on their common causes and selecting a representative error trace for each cluster. This usability enhancement is crucial to making type-qualifier inference accessible to programmers.

In addition to these contributions, we demonstrate a new application of type-qualifier inference to security: checking operating-system kernels for user/kernel pointer bugs. We tested these new algorithms and heuristics in two sets of experiments. First, we analyzed several versions of the Linux kernel to find user/kernel pointer bugs. We found over two dozen such bugs, even in code that had been audited manually and by other code-auditing tools. We also tested the scalability of our algorithms by performing a whole-kernel analysis on the Linux 2.4 and 2.6 kernels. Second, we revisited experiments, by Shankar, et al[78],

that used an earlier implementation of type-qualifier inference to find format-string bugs. They found several bugs but had hundreds of false positives and had to write application-specific annotations to get good results. The new techniques mentioned above eliminate all but two of the false positives without requiring any application-specific annotations. The research presented here thus can serve as the foundation for more advanced analyses, but it also has direct practical application on its own.

Chapter 2 describes several security vulnerabilities that can be statically detected using type-qualifier inference. Chapter 3 presents a simplified type-qualifier inference system for a simple language that lacks many of the nettlesome features of C, such as structures and casts. In Chapter 4, we detail the refinements to type-qualifier inference developed as part of this research. Chapter 5 describes the experiments we performed to validate these refinements and gives the results of those experiments. Chapter 6 surveys related work, and Chapter 7 summarizes the results of this work.

Chapter 2

Input-Validation Bugs

Programs receive input from many different sources: command-line arguments, environment variables, configuration files, input files, local IPC, RPC and other network communication. If the program is not written correctly, then malformed or unexpected inputs from any of these sources can cause it to crash or behave incorrectly. Attackers can specifically construct inputs that trigger bugs in a target program, causing it to perform actions advantageous to the attacker. In extreme cases, incorrect input can allow a hacker to completely take over the victim program. We call these bugs Input-Validation Bugs.

Many programmers are aware of these dangers and structure their programs to minimize the possibility of error. A common paradigm is to write a validation or sanitization function that checks inputs that come from untrusted sources. The difference between validation and sanitization functions is as follows. Upon encountering an invalid input, a validation function either discards that input or terminates the program, depending on the specific application scenario. A sanitization function cleans the input, e.g. by changing any dangerous characters to safe ones, and passes it on for regular processing.

Some programmers program defensively to further reduce the likelihood of input-validation vulnerabilities by limiting the number of vulnerable points in their programs. In all the examples below, there are only a few functions or operations that cannot handle

arbitrary user input. These are the only targets that must be protected by the sanitation or validation functions.

As a consequence, there are typically three aspects of a program that are relevant to its handling of untrusted inputs:

- The sources of untrusted inputs. For example, on UNIX systems, the `read` system call may return untrusted input.
- The sanitization and validation functions for untrusted inputs. These are usually application dependent.
- The operations that are vulnerable to attack. If data from an untrusted source reaches one of these operations without passing through a sanitization or validation function, then the program may be vulnerable to attack.

Typically, programmers can easily identify the functions that gather user input (e.g. the `read` system call), the functions or language constructs that are vulnerable to exploit (e.g. stack-allocated buffers), and the sanitization checks that must be performed. The prime difficulty that remains is: does every data-path from inputs to vulnerable targets pass through the appropriate sanitization or validation function?

In this section, we describe two specific instances of input-validation vulnerabilities that we will use later to evaluate our program analysis algorithms. SQL injection bugs, cross-site scripting bugs, and even integer overflow bugs all fit this model.

2.1 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 2.1. 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

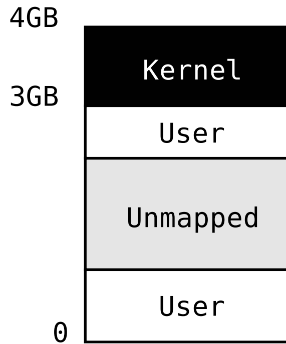


Figure 2.1. The Linux virtual memory layout on 32-bit architectures.

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

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

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 address, `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 an address 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 can gain 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 private key of the local web server.

User/kernel pointer bugs are hard to detect during testing because, in most cases, they do not cause failures during normal operation of the system. 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 in our experiments 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, a similar 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: Pointers whose value is under user control and hence untrustworthy.

Kernel pointers: Pointers whose values are under kernel control and guaranteed by the kernel to always point into the kernel's memory space, and hence are 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.

2.2 Format-String Bugs

Although they are not the primary focus of this research, format-string bugs are an important class of input-validation vulnerabilities. Previous research has shown that type-qualifier inference can find format-string bugs[78], and we extend those results (see Chapter 5).

The standard C library contains the `printf` family of functions for formatting and displaying text. For the purposes of discussing format-string bugs, all the functions in the

`printf` family are equivalent, so we will only consider `printf` for this exposition. The `printf` function has signature

```
int printf(const char *format, ...);
```

indicating that it takes a string argument, `format`, and a variable number of subsequent arguments. The format argument can contain literal characters, which will be written to the program's output, and special format characters that control the interpretation and display of subsequent arguments to `printf`. For example, the call

```
printf("Today is %s %d", "March", 23);
```

would write `Today is March 23` to the program's output. As the example shows, format commands begin with "%" and have a character indicating whether their corresponding argument is a string, integer, or other type of data.

The `printf` interface is easy to understand, but its implementation is trickier. Figure 2.2 shows the stack layout during the execution of the above call to `printf`. The caller pushes the arguments to `printf` onto the stack in reverse order, so it pushes 23 onto the stack first and the format string last. Then it pushes the return address onto the stack and transfers control to `printf`. When it begins execution, `printf` initializes its local variable, `nextarg` to point to the first of its optional arguments. It then processes the format argument one character at a time. Whenever it encounters a % format directive, it uses the argument to which `nextarg` is currently pointing, and then advances `nextarg`.

Notice that there is no signalling mechanism to indicate the number of optional arguments passed to `printf`, or their types. If the caller accidentally forgot to include one of the optional arguments, as in the call

```
printf("Today is %s %d, %d", "March", 23);
```

then printf will look for a third argument that isn't there. At that time `nextarg` will point to one of the caller's local variables. That local variable may be an integer, string, or some other data type, but `printf` will interpret the bytes at that memory location as an integer and print them out.

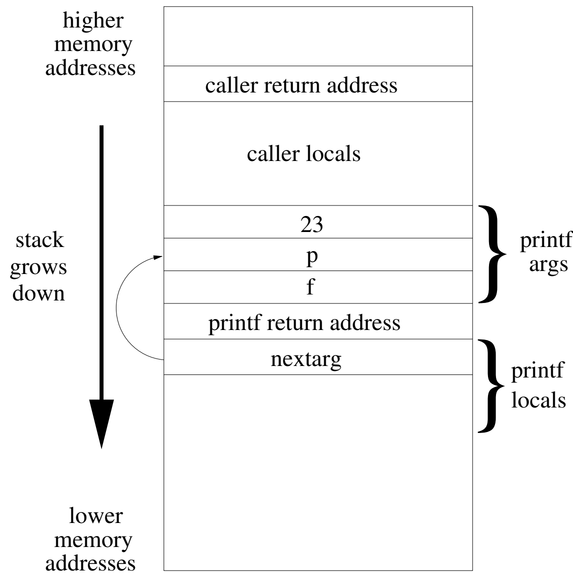


Figure 2.2. The stack layout during the call `printf("Today is %s %d", "March", 23)`. In this figure, `p` is a pointer to the string `"March"` and `f` is a pointer to the string `"Today is %s %d"`.

To see how an attacker can exploit this confusion to his advantage, consider the function shown in Figure 2.3. The `sayhello` function, which might be found in a network daemon, takes the user's login name and prints out a simple hello message. Many network protocols, such as SMTP and FTP, contain similar messages. This function uses `snprintf`, which behaves like `printf` except that it places its output in the buffer passed as its first argument. The format string is its second argument, and here the programmer has taken the shortcut of passing the username directly as the format argument. As long as the username doesn't contain any %s, this will have the desired effect of copying the contents of `username` into `buf`.

If username contains a format directive, though, then `snprintf` will look for the corresponding argument at `nextarg`, which points into `buf`. An attacker could therefore enter a username of the form "$b_1b_2b_3b_4$%s", where $b_1b_2b_3b_4$ is a byte-sequence representing an address in memory. The `snprintf` function will copy these bytes from `username` into `buf`. Note that `nextarg` points to the beginning of `buf`, so the next format directive will cause `snprintf` to treat these bytes as the associated optional argument. When `snprintf` encounters the `%s`, it interprets the bytes $b_1b_2b_3b_4$ as a pointer to a string and prints out

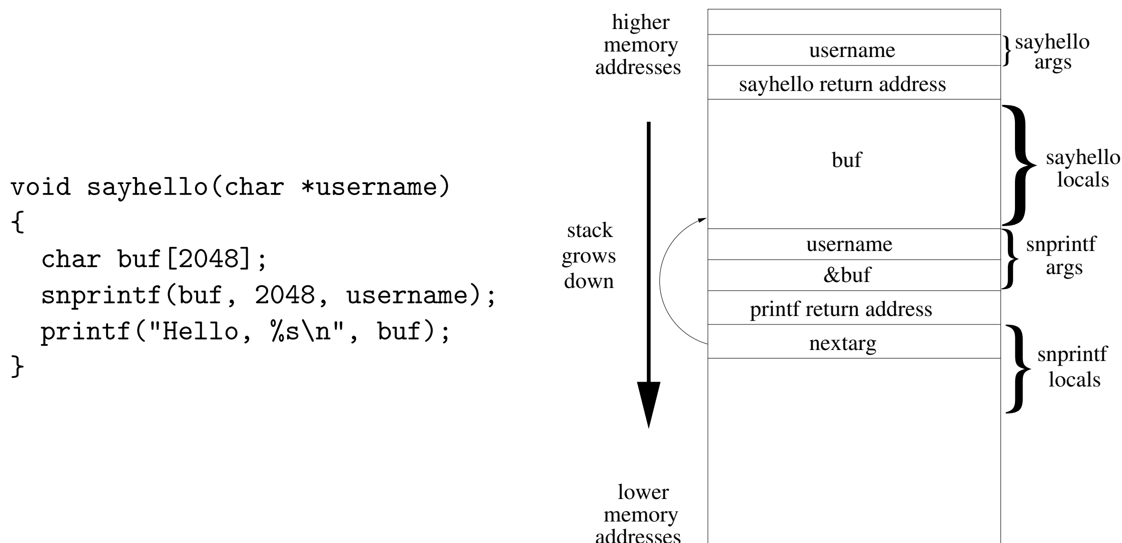


Figure 2.3. A function vulnerable to format-string attacks and the stack layout during the execution of this function.

that string. This is remarkable: the attacker has tricked `sayhello` into printing out the contents of memory at an address of his choosing. If the program executing `sayhello` has any secrets in memory, such as a password database or private encryption key, then the attacker can use this trick to extract that secret.

The `printf` function also supports a `%n` directive that causes `printf` to store the number of bytes output so far into the corresponding integer argument. For example, the call

```
int count;
printf("Hello%nGoodbye", &count);
```

would cause `printf` to store 5 into count.

An attacker can use this feature to corrupt the memory of a vulnerable program, ultimately taking complete control of its execution. Referring again to Figure 2.3, if the attacker sends a username of the form "$b_1b_2b_3b_4$%n", where $b_1b_2b_3b_4$ is a memory address, then, as in the attack above, the `snprintf` function will copy that address into the first four bytes of `buf`. When it encounters the `%n`, it will interpret those bytes as a pointer to an integer and write the value 4 into the attacker's chosen memory location. To write other values, the attacker can insert padding bytes between b_4 and `%n`. Although this basic technique will only let the attacker write small values to memory (they can't send billions

```
int snprintf(char * dest, char untainted * fmt, ...);

void sayhello(char tainted *username)
{
  char buf[2048];
  snprintf(buf, username);
  printf("Hello, %s\n", buf);
}
```

Figure 2.4. Type-qualifier annotations on the `sayhello` and `snprintf` functions make the format-string bug in this code an obvious typing error.

of bytes of padding), by combining several writes of small values, it is possible to write any value into the target memory location[61].

As with user/kernel pointer bugs, the ability to write any chosen value to any memory location is sufficient to give an attacker complete control over the victim program. An attacker could, for example, include arbitrary machine code in the given username and use `%n` directives to overwrite `sayhello`'s return address so that it jumps to the attacker's code.

We can catch format-string bugs using type-qualifier inference by introducing two new qualifiers, *tainted* and *untainted* [78]. The programmer can then annotate all sources of untrusted data as returning *tainted* values and annotate the format argument to each `printf`-like function as only accepting *untainted* data. If any data ever flows into a format-string, then it will cause a type-qualifier mismatch, and hence an analysis warning. Figure 2.4 shows the `sayhello` function with these annotations. In this example, `snprintf` is clearly called with a format argument of the wrong type.

Strings support other operations, such as concatenation and filtering, and type-qualifiers can model these operations, too. Observe that a function that is sufficiently careful to accept *tainted* string arguments should operate correctly even if we happen to pass it an *untainted* string. In general, it is safe to use an *untainted* string anywhere a *tainted* string is expected. The simple typing rule `char` *untainted* `*` $<$ `char` *tainted* `*`, where $\tau < \tau'$ means that type τ is a subtype of type τ', captures this notion exactly.

For example, a function that takes a string as input and changes all occuramces of `%` to

`#` is sufficient to sanitize a string for use as a `printf` format argument. The straightforward implementation below captures this fact:

```
char untainted * pcnts2octlthrp(char tainted * s)
{
  char * p = s;
  while (*p)
    if (*p == '\%')
      *p = '\#';
  return (char untainted *)s;
}
```

Notice that the function returns an *untainted* string by simply casting its *tainted* argument to *untainted*.

Chapter 3

Type-Qualifier Inference

This chapter provides the background on type-qualifiers needed to understand the contributions presented in Chapter 4.

We will present the basic theory of type-qualifiers using lambda calculus extended with updatable references, as shown in Figure 3.1. In general, type qualifiers can be added to any language with a standard type system, but showing the system on the language in Figure 3.1 will illustrate most of the important points. In this Chapter we present a basic type qualifier system that will need some extensions, described in Chapter 4, to be more useful in practice and to handle certain classes of qualifiers. The next chapter contains a discussion of how to address some of the issues that come up in applying type qualifiers to the C programming language.

We will assume for the remainder of this section that our input programs are type correct with respect to the standard type system, shown in Figure 3.2 for completeness, and that function definitions have been annotated with standard types s. If that is not the case, we can always perform a preliminary standard type inference pass.

In our framework, the user specifies a set of qualifiers Q and a partial order $\leq$ among the qualifiers. In practice, the user may wish to specify several sets $(Q_i, \leq_i)$ of qualifiers that do not interact, each with their own partial order. But then we can form $(Q, \leq) = (Q_1, \leq_1) \times \cdots \times (Q_n, \leq_n)$, so without loss of generality we can assume a single partial order of

e	$::=$	v	values
	$\mid$	$e_1\ e_2$	application
	$\mid$	$\texttt{let}\, x = e_1\, \texttt{in}\ e_2$	name binding
	$\mid$	$\texttt{ref}\, e$	allocation
	$\mid$	$\texttt{*}e$	dereference
	$\mid$	$e_1 \texttt{:=} e_2$	assignment
v	$::=$	x	variable
	$\mid$	n	integer
	$\mid$	$\lambda x\!:\!s.e$	function
s	$::=$	int	integer type
	$\mid$	$\mathit{ref}\ (s)$	pointer to type s
	$\mid$	$s \longrightarrow s'$	function from type s to type s'

Figure 3.1. Source Language

$$\frac{x \in \mathit{dom}(\Gamma)}{\Gamma \vdash_s x : \Gamma(x)}\ (\mathrm{Var}_s) \qquad \frac{}{\Gamma \vdash_s n : \mathit{int}}\ (\mathrm{Int}_s)$$

$$\frac{\Gamma[x \mapsto s] \vdash_s e : s'}{\Gamma \vdash_s \lambda x\!:\!s.e : s \longrightarrow s'}\ (\mathrm{Lam}_s) \qquad \frac{\Gamma \vdash_s e : s}{\Gamma \vdash_s \texttt{ref}\, e : \mathit{ref}\ (s)}\ (\mathrm{Ref}_s)$$

$$\frac{\Gamma \vdash_s e_1 : s \longrightarrow s' \quad \Gamma \vdash_s e_2 : s}{\Gamma \vdash_s e_1\ e_2 : s'}\ (\mathrm{App}_s) \qquad \frac{\Gamma \vdash_s e_1 : s_1 \quad \Gamma[x \mapsto s_1] \vdash_s e_2 : s_2}{\Gamma \vdash_s \texttt{let}\, x = e_1\, \texttt{in}\ e_2 : s_2}\ (\mathrm{Let}_s)$$

$$\frac{\Gamma \vdash_s e : \mathit{ref}\ (s)}{\Gamma \vdash_s \texttt{*}e : s}\ (\mathrm{Deref}_s) \qquad \frac{\Gamma \vdash_s e_1 : \mathit{ref}\ (s) \quad \Gamma \vdash_s e_2 : s}{\Gamma \vdash_s e_1 \texttt{:=} e_2 : s}\ (\mathrm{Assign}_s)$$

Figure 3.2. Standard Type Checking System

qualifiers. For example, Figure 3.3 gives two independent partial orders and their equivalent combined, single partial order (in this case the partial orders are lattices). In Figure 3.3, as in the rest of this paper, we write elements of Q using *slanted text*. We sometimes refer to elements of Q as *type-qualifier constants* to distinguish them from type-qualifier variables introduced in Section 3.2.

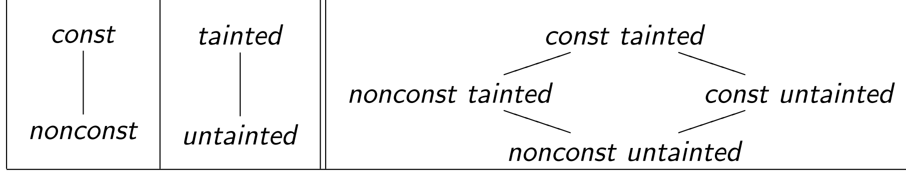


Figure 3.3. Example Qualifier Partial Order

For our purposes, *types* *Typ* are terms over a set Σ of n-ary type constructors. Grammatically, types are defined by the language

$$\mathit{Typ} ::= c(\mathit{Typ}_1, \ldots, \mathit{Typ}_{\mathit{arity}(c)}) \qquad c \in \Sigma$$

In our source language, the type constructors are $\{\mathit{int}, \mathit{ref}, \longrightarrow\}$ with arities 0, 1, and 2, respectively. We construct the *qualified types* *QTyp* by pairing each standard type constructor in Σ with a type qualifier (recall that a single type qualifier in our partial order may represent a set of qualifiers in the programmer's view). We allow type qualifiers to appear on every level of a type. Grammatically, our new types are

$$\mathit{QTyp} ::= Q\ c(\mathit{QTyp}_1, \ldots, \mathit{QTyp}_{\mathit{arity}(c)}) \qquad c \in \Sigma$$

For our source language, the qualified types are

$$\begin{array}{rcl} \tau & ::= & Q\ \nu \\ \nu & ::= & \mathit{int} \mid \mathit{ref}\ (\tau) \mid \tau \longrightarrow \tau \end{array}$$

To avoid ambiguity, when writing down qualified function types we parenthesize them as $Q\ (\tau \longrightarrow \tau)$. Some example qualified types in our language are *tainted* int and *const* $\mathit{ref}\ ($*untainted* $\mathit{int})$. We define the *top-level qualifier* of type $Q\ \nu$ as its outermost qualifier Q.

3.1 Assertions, Annotations, and Type Checking

So far we have types with attached qualifiers and a partial order among the qualifiers. A key idea behind our framework is that the partial order on type qualifiers induces a *subtyping* relation among qualified types. In a subtyping system, if type B is a subtype of

$$\frac{Q \leq Q'}{Q\ \ int \leq Q'\ \ int}\ (\text{Int}_{\leq}) \qquad \frac{Q \leq Q' \quad \tau \leq \tau' \quad \tau' \leq \tau}{Q\ \ ref\ (\tau) \leq Q'\ \ ref\ (\tau')}\ (\text{Ref}_{\leq})$$

$$\frac{Q \leq Q' \quad \tau_1' \leq \tau_1 \quad \tau_2 \leq \tau_2'}{Q\ (\tau_1 \longrightarrow \tau_2) \leq Q'\ (\tau_1' \longrightarrow \tau_2')}\ (\text{Fun}_{\leq})$$

(a) Subtyping Qualified Types

$$\frac{}{\Gamma \vdash n : int}\ (\text{Int}) \qquad \frac{\Gamma[x \mapsto \tau] \vdash e : \tau' \quad strip(\tau) = s}{\Gamma \vdash \lambda x{:}s.e : \tau \longrightarrow \tau'}\ (\text{Lam})$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{ref}\, e : ref\ (\tau)}\ (\text{Ref}) \qquad \frac{\Gamma \vdash e : \nu}{\Gamma \vdash \texttt{annot}(e, Q) : Q\ \nu}\ (\text{Annot})$$

(b) Rules for Unqualified Types ν

$$\frac{x \in dom(\Gamma)}{\Gamma \vdash x : \Gamma(x)}\ (\text{Var}) \qquad \frac{\Gamma \vdash e_1 : Q\ (\tau \longrightarrow \tau') \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_2 \leq \tau}{\Gamma \vdash e_1\ e_2 : \tau'}\ (\text{App})$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma[x \mapsto \tau_1] \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}\, x = e_1\, \texttt{in}\ e_2 : \tau_2}\ (\text{Let}) \qquad \frac{\Gamma \vdash e : Q\ \ ref\ (\tau)}{\Gamma \vdash \texttt{*}e : \tau}\ (\text{Deref})$$

$$\frac{\Gamma \vdash e_1 : Q\ \ ref\ (\tau) \quad \Gamma \vdash e_2 : \tau' \quad \tau' \leq \tau}{\Gamma \vdash e_1 \texttt{:=} e_2 : \tau}\ (\text{Assign})$$

$$\frac{\Gamma \vdash e : Q'\ \nu \quad Q' \leq Q}{\Gamma \vdash \texttt{check}(e, Q) : Q'\ \nu}\ (\text{Check})$$

(c) Rules for Qualified Types τ

Figure 3.4. Qualified-Type Checking System

type A, which we write $B \leq A$ (note the overloading on $\leq$), then wherever an object of type A is allowed an object of type B may also be used. (Subclassing in object-oriented programming languages such as Java and C++ is closely related to subtyping.)

Figure 3.4(a) shows how a given qualifier partial order is extended to a subtyping relation for our source language. These rules are standard, and a discussion of them can be found elsewhere [56]. In general, for any $c \in \Sigma$ the rule

$$\frac{Q \leq Q' \qquad \tau_i \leq \tau_i' \qquad \tau_i' \leq \tau_i \qquad i \in [1..n]}{Q\ c(\tau_1, \ldots, \tau_n) \leq Q'\ c(\tau_1', \ldots, \tau_n')}$$

should be sound. Whether the equality (here expressed by two $\leq$ constraints) can be relaxed for any particular position depends on the meaning of the type constructor c.

Next we wish to extend our standard type system to work with qualified types. Thus far, however, we have supplied no mechanism that allows programmers to talk about the qualifiers used in their programs. One place where this issue comes up is when constructing a qualified type during type checking. For example, if we see an occurrence of the integer `0` in the program, how do we decide which qualifier Q to pick for its type $Q\ \ int$? We wish to have a generic solution for this problem, so in our system, we extend the syntax with two new forms:

$$\begin{array}{lcll} e & ::= & \cdots & \\ & | & \texttt{annot}(e, Q) & \text{qualifier annotation} \\ & | & \texttt{check}(e, Q) & \text{qualifier check} \end{array}$$

A *qualifier annotation* $\texttt{annot}(e, Q)$ specifies the outermost qualifier Q to add to e's type. Annotations may only be added to expressions that construct a term, and whenever the user constructs a term our type system requires that they add an annotation. Clearly this last requirement is not always desirable, and in Section 3.2 we describe an inference algorithm that allows programmers to omit these annotations if they like. Dually, a *qualifier check* $\texttt{check}(e, Q)$ tests whether the outermost qualifier of e's type is compatible with Q. Notice that if we want to check a qualifier deeper in a non-function type, we can do so by first applying the language's deconstructors. (For example, we can check the qualifier on the contents of a reference x using $\texttt{check}(*x, Q)$).

Finally, we wish to extend the original type checking system to a *qualified-type system*

that checks programs with qualified types, including our new syntactic forms $\texttt{annot}(\cdot,\cdot)$ and $\texttt{check}(\cdot,\cdot)$. Intuitively this extension should be natural, in the sense that adding type qualifiers should not modify the underlying type structure (we make this precise below). We also need to incorporate subsumption [56] into our qualified-type system to allow subtyping.

We define a pair of translation functions between standard and qualified types and expressions. For a qualified type $\tau \in \mathit{QTyp}$, we define $\mathit{strip}(\tau) \in \mathit{Typ}$ to be τ with all qualifiers removed. Analogously, $\mathit{strip}(e)$ is e with any qualifier annotations or checks removed. In the other direction, for a standard type $s \in \mathit{Typ}$ we define $\mathit{embed}(s, q)$ to be the qualified type with the same shape as s and all qualifiers set to q. Analogously, $\mathit{embed}(e, q)$ is e with $\texttt{annot}(e', q)$ wrapped around every subexpression e' of e that constructs a term.

Figures 3.4(b) and c show the qualified-type system for our source language. Judgments are either of form $\Gamma \vdash e : \nu$ (three rules in Figure 3.4(b)) or $\Gamma \vdash e : \tau$ (the remaining rules in Figure 3.4(c)), meaning that in type environment Γ, expression e has unqualified type ν or qualified type τ. Here Γ is a mapping from variables to qualified types.

The rules (Int) and (Ref) are identical to standard type checking rules, and (Lam) simply adds a check that the parameter's qualified type τ has the same shape as the specified standard type. (This check is not strictly necessary—see Lemma 1 below.) Notice that these three rules produce types that are missing a top-level qualifier. The rule (Annot) adds a top-level qualifier to such a type, which is produced in our qualified-type grammar by non-terminal ν. Inspection of the type rules shows that judgments of the form $\Gamma \vdash e : \nu$ can only be used in the hypothesis of (Annot). Thus the net effect of the four rules in Figure 3.4(b) is that all constructed terms must be assigned a top-level qualifier with an explicit annotation.

The rules (Var) and (Let) are identical to the standard type checking rules. The rules (App), (Deref), and (Assign) are similar to the standard type checking rules, except that they match the types of their subexpressions against qualified types, and (App) and (Assign) allow subsumption. Notice that these three rules allow arbitrary qualifiers (denoted by Q) when matching a type. Only the rule (Check) actually tests a qualifier on a type.

Lemma 1 *Let e be a closed term, and let $\vdash_s$ be the standard type checking judgment.*

- *If $\emptyset \vdash_s e : s$, then for any qualifier q we have $\emptyset \vdash \mathit{embed}(e, q) : \mathit{embed}(s, q)$.*
- *If $\emptyset \vdash e : \tau$, then $\emptyset \vdash_s \mathit{strip}(e) : \mathit{strip}(\tau)$.*

Proof: (Sketch) Both properties can be proven by structural induction on e. The proofs are straightforward, since each rule in the standard type system corresponds exactly to one rule in Figure 3.4 with the qualifiers removed, and vice-versa. For the first claim, we also must observe that consistently adding the same qualifier q to all types maintains the same equalities between types in the program, which is sufficient for maintaining typability because $\mathit{embed}(\cdot, \cdot)$ does not add any type qualifier checks. For the second statement, we must also observe that the subsumption rules in Figure 3.4(a) require type structures to be equal, and so if $\tau \leq \tau'$, it is always the case that $\mathit{strip}(\tau) = \mathit{strip}(\tau')$. $\square$

This lemma formalizes our intuitive requirement that type qualifiers do not affect the underlying type structure.

3.2 Inference

As described so far, type qualifiers place a rather large burden on programmers wishing to use them: programmers must add explicit qualifier annotations to all constructed terms in their programs. We would like to reduce this burden by performing *type-qualifier inference*, which is analogous to standard type inference. As with standard type inference, we introduce *type-qualifier variables* QVar to stand for unknown qualifiers for which we need to solve. We write qualifier variables with the Greek letter κ. In the remainder of this paper we use *type qualifier constants* to refer to elements of the given qualifier partial order, and we use *type qualifiers* to refer to either a qualifier constant or a qualifier variable. We define a function $\mathit{embed}'(s)$ that maps standard types to qualified types by inserting fresh type

$$\frac{}{\Gamma \vdash' n : \mathit{int}}\ (\mathrm{Int}') \qquad \frac{\Gamma[x \mapsto \tau] \vdash' e : \tau' \qquad \tau = \mathit{embed}'(s)}{\Gamma \vdash' \lambda x{:}s.e : \tau \longrightarrow \tau'}\ (\mathrm{Lam}')$$

$$\frac{\Gamma \vdash' e : \tau}{\Gamma \vdash' \texttt{ref}\, e : \mathit{ref}\ (\tau)}\ (\mathrm{Ref}') \qquad \frac{\Gamma \vdash' e : \nu}{\Gamma \vdash' \texttt{annot}(e, Q) : Q\ \nu}\ (\mathrm{Annot}')$$

$$\frac{\Gamma \vdash' e : \nu \qquad \kappa \text{ fresh}}{\Gamma \vdash' e : \kappa\ \nu}\ (\mathrm{Fresh}')$$

(a) Rules for Unqualified Types ν

$$\frac{x \in \mathit{dom}(\Gamma)}{\Gamma \vdash' x : \Gamma(x)}\ (\mathrm{Var}') \qquad \frac{\Gamma \vdash' e_1 : Q\ (\tau \longrightarrow \tau') \qquad \Gamma \vdash' e_2 : \tau_2 \qquad \tau_2 \leq \tau}{\Gamma \vdash' e_1\ e_2 : \tau'}\ (\mathrm{App}')$$

$$\frac{\Gamma \vdash' e_1 : \tau_1 \qquad \Gamma[x \mapsto \tau_1] \vdash' e_2 : \tau_2}{\Gamma \vdash' \texttt{let}\, x = e_1\, \texttt{in}\, e_2 : \tau_2}\ (\mathrm{Let}') \qquad \frac{\Gamma \vdash' e : Q\ \ \mathit{ref}\ (\tau)}{\Gamma \vdash' \texttt{*}e : \tau}\ (\mathrm{Deref}')$$

$$\frac{\Gamma \vdash' e_1 : Q\ \ \mathit{ref}\ (\tau) \qquad \Gamma \vdash' e_2 : \tau' \qquad \tau' \leq \tau}{\Gamma \vdash' e_1 \texttt{:=} e_2 : \tau'}\ (\mathrm{Assign}')$$

$$\frac{\Gamma \vdash' e : Q'\ \nu \qquad Q' \leq Q}{\Gamma \vdash' \texttt{check}(e, Q) : Q'\ \nu}\ (\mathrm{Check}')$$

(b) Rules for Qualified Types τ

Figure 3.5. Qualified-Type Inference System

qualifier variables at every level:

$$\begin{array}{rcll} embed'(int) & = & \kappa\ int & \kappa \text{ fresh} \\ embed'(ref\ (s)) & = & \kappa\ ref\ (embed'(s)) & \kappa \text{ fresh} \\ embed'(s \longrightarrow s') & = & \kappa\ (embed'(s) \longrightarrow\ embed'(s')) & \kappa \text{ fresh} \end{array}$$

The type-qualifier inference rules for our source language are shown in Figure 3.5. In this system, Q stands for either a qualifier constant or a qualifier variable κ. These rules are essentially the same as the rules in Figure 3.4, with two differences. First, we allow qualifier annotations to be omitted in the source program. If they are, rule (Fresh′) is used to introduce a fresh type qualifier variable to stand for the unknown qualifier on the term. Although this rule is not syntax-driven, it could be made so, since either rule (Annot′) or rule (Fresh′) must be used before applying any of the rules in part (b). Second, we use *embed′* in (Lam′) to map the given standard type to a type with fresh qualifier variables. To simplify the rules slightly we use our assumption that the program is correct with respect to the standard types to avoid some shape matching constraints. For example, in (App′) we know that e_1 has a function type, but we do not know its qualifier.

Given a solution to the constraints generated by inference (see below), we believe it is straightforward to show that the inference system in Figure 3.5 is sound and complete with respect to the checking system in Figure 3.4, although we have not proven it formally.

After we have applied the type rules in Figure 3.5, we have a typing derivation that contains qualifier variables to be solved for. The qualifier variables must satisfy two kinds of constraints that form the side conditions of the inference rules: *subtyping constraints* of the form $\tau_1 \leq \tau_2$ and *qualifier constraints* (from the rule (Check′)) of the form $Q_1 \leq Q_2$. We say that these constraints are *generated* by the typing rules. In order to find a valid typing (if one exists), we must solve the generated constraints for the qualifier variables.

The first step is to apply the rules of Figure 3.6(a) to reduce the subtyping constraints to qualifier constraints. Notice that because we assume that the program we are analyzing type checks with respect to the standard types, we know that none of the structural matching cases in Figure 3.4(a) can fail.

$$C \cup \{Q\ \mathit{int} \leq Q'\ \mathit{int}\} \quad \Rightarrow \quad C \cup \{Q \leq Q'\}$$
$$C \cup \{Q\ \mathit{ref}\ (\tau) \leq Q'\ \mathit{ref}\ (\tau')\} \quad \Rightarrow \quad C \cup \{Q \leq Q'\} \cup \{\tau \leq \tau'\} \cup \{\tau' \leq \tau\}$$
$$C \cup \{Q\ (\tau_1 \longrightarrow \tau_2) \leq Q'\ (\tau_1' \longrightarrow \tau_2')\} \quad \Rightarrow \quad C \cup \{Q \leq Q'\} \cup \{\tau_1' \leq \tau_1\} \cup \{\tau_2 \leq \tau_2'\}$$

(a) Resolution Rules for Subtyping Constraints

$$C \cup \{\mathsf{q} \leq Q\} \cup \{Q \leq Q'\} \quad \cup\!\Rightarrow \quad \{\mathsf{q} \leq Q'\}$$
$$C \cup \{Q \leq Q'\} \cup \{Q' \leq \mathsf{q}\} \quad \cup\!\Rightarrow \quad \{Q \leq \mathsf{q}\}$$

(b) Resolution Rules for Qualifier Constraints

Figure 3.6. Constraint Resolution

After exhaustively applying the rules in Figures 3.5 and 3.6(a), we are left with qualifier constraints of the form $Q_1 \leq Q_2$, where the Q_i are type-qualifier constants from Q or type-qualifier variables κ. We need to solve these qualifier constraints to complete type-qualifier inference.

Definition 2 *A* solution *S to a system of qualifier constraints C is a mapping from type-qualifier variables to type-qualifier constants such that for each constraint $Q_1 \leq Q_2$, we have $S(Q_1) \leq S(Q_2)$.*

We write $S \models C$ if S is a solution to C. Note that there may be many possible solutions to C. We say that C is *satisfiable* if there exists an S such that $S \models C$. For many uses of type qualifiers, we are interested in satisfiability rather than the actual solution. If we are looking for a solution there are two in particular that we may be interested in.

Definition 3 *If $S \models C$, then S is a* least *(respectively* greatest*)* solution *if, for any other S' such that $S' \models C$, and for all $\kappa \in dom(S)$, we have $S(\kappa) \leq S'(\kappa)$ (respectively $S(\kappa) \geq S'(\kappa)$).*

For example, when inferring *const* annotations to enable more compiler optimizations[34], finding a greatest solution is more useful since it infers as many *const*s as possible. For untrusted-input vulnerabilities, though, we are mostly interested in satisfiability. However, if we wanted to compute a full solution, we would find a least solution so as to label as much data as possible as trusted.

A system of qualifier constraints is also known as an *atomic subtyping constraint system*. In general, checking satisfiability and/or solving atomic subtyping constraints over an arbitrary partial order is NP-hard, even with fixed Q [72]. However, there are well-known linear-time algorithms for solving such constraints efficiently if Q is a semilattice [74]. In this case, given a system of constraints C of size n and a fixed set of qualifiers, we can check satisfiability of the constraints and find a solution in $O(n)$ time [74, 37].

For example, if Q is a lattice, we can repeatedly apply the transitive closure rules in Figure 3.6(b) to the qualifier constraints until we reach a fixpoint. These rules effectively propagate qualifier constants q through the constraints. Here $\cup\!\Rightarrow$ means the constraint on the right-hand side is added given the constraints on the left-hand side. After reaching a fixpoint, the constraint system C is satisfiable if and only if there is no constraint $q \leq q' \in C$ where $q \not\leq q'$ in the lattice. For a qualifier variable κ, its least solution is $\bigsqcup_{\{q \leq \kappa\} \in C} q$ and its greatest solution is $\bigsqcap_{\{\kappa \leq q\} \in C} q$.

3.3 Semantics and Soundness

We can prove that our qualified-type system (applied to our extended lambda calculus), including parametric polymorphic type qualifiers (Section 4.1), is sound under a natural semantics for type qualifiers. Figure 3.8 gives the semantic reduction rules, taken from prior work on type-qualifier inference[34]. In these semantics a store S is a mapping from locations l to values v. In order to type check programs with locations in them, we treat locations l as free variables and type them using rule (Var). Thus in the proof of soundness, as reduction proceeds and new locations are allocated we keep track of their types in the type environment. We use $\emptyset$ for the empty store. Values are standard values (integers,

$$
\begin{array}{rcll}
v & ::= & n^Q & \text{integers} \\
 & | & l^Q & \text{locations} \\
 & | & (\lambda x.e)^Q & \text{functions} \\
e & ::= & t & \text{terms} \\
 & | & e_1\ e_2 & \text{application} \\
 & | & \texttt{let}\ x = e_1\ \texttt{in}\ e_2 & \text{name binding} \\
 & | & \texttt{ref}\ e & \text{allocation} \\
 & | & \texttt{*}e & \text{dereference} \\
 & | & e_1 \texttt{:=} e_2 & \text{assignment} \\
 & | & \texttt{annot}(e, Q) & \text{qualifier annotation} \\
 & | & \texttt{check}(e, Q) & \text{qualifier check} \\
t & ::= & x & \text{variable} \\
 & | & n & \text{integer} \\
 & | & \lambda x.e & \text{function}
\end{array}
$$

Figure 3.7. Values for Big-Step Semantics

locations, or functions) paired with uninterpreted qualifiers, written as a superscript. The language of values is shown in Figure 3.7. Notice that in these semantics the rules that deconstruct values ([App], [Deref], and [Assign]) throw away the outermost qualifier. Only rule [Check] tests the top-level qualifier of a value.

As mentioned before, these semantics require `annot` to wrap every expression that constructs a term in the language. When qualifier inference is used, the programmer may have omitted many of these annotations. Given a solution S to the type qualifier inference problem, we can insert all the required annotations. To do so, all that is required is to replace every expression e to which the (Fresh′) rule was applied by $\texttt{annot}(e, S(\kappa))$, where κ is the fresh qualifier created for e using (Fresh′).

Any program to which none of the rules in Figure 3.8 apply is *stuck*, which we express by reducing the program to a special symbol **err**. The symbol **err** is not a value and

has no valid type. We can prove that our system is sound by showing that no well-typed program reduces to **err** . The proof is omitted, since it uses standard techniques [91, 26, 62]; Foster included a proof for the monomorphic type-qualifier system in his thesis [37]. In the theorem below, we use r to stand for a reduction result, either a value v^Q or **err**.

Theorem 4 *If* $\emptyset \vdash e : \tau$ *and* $\langle \emptyset, e \rangle \rightarrow \langle S', r \rangle$*, then* r *is not* ***err****.*

$$\frac{l \in dom(S)}{\langle S, l^Q \rangle \to \langle S, l^Q \rangle} \text{[Var]}$$

$$\frac{}{\langle S, \texttt{annot}(n, Q) \rangle \to \langle S, n^Q \rangle} \text{[Int]}$$

$$\frac{}{\langle S, \texttt{annot}(\lambda x\!:\!s.e, Q) \rangle \to \langle S, (\lambda x.e)^Q \rangle} \text{[Lam]}$$

$$\frac{\begin{array}{c}\langle S, e_1 \rangle \to \langle S_1, (\lambda x.e)^{Q_1} \rangle \qquad \langle S_1, e_2 \rangle \to \langle S_2, {v_2}^{Q_2} \rangle \\ \langle S_2, e[{v_2}^{Q_2}/x] \rangle \to \langle S_3, {v_3}^{Q_3} \rangle\end{array}}{\langle S, e_1\ e_2 \rangle \to \langle S_3, {v_3}^{Q_3} \rangle} \text{[App]}$$

$$\frac{\langle S, e_1 \rangle \to \langle S_1, {v_1}^{Q_1} \rangle \qquad \langle S_1, e_2[{v_1}^{Q_1}/x] \rangle \to \langle S_2, {v_2}^{Q_2} \rangle}{\langle S, \texttt{let}\, x = e_1 \,\texttt{in}\, e_2 \rangle \to \langle S_2, {v_2}^{Q_2} \rangle} \text{[Let]}$$

$$\frac{\langle S, e \rangle \to \langle S_1, v^Q \rangle \qquad l \notin dom(S_1)}{\langle S, \texttt{annot}(\texttt{ref}\, e, Q') \rangle \to \langle S_1[l \mapsto v^Q], l^{Q'} \rangle} \text{[Ref]}$$

$$\frac{\langle S, e \rangle \to \langle S_1, l^Q \rangle \qquad l \in dom(S_1)}{\langle S, \texttt{*}e \rangle \to \langle S_1, S_1(l) \rangle} \text{[Deref]}$$

$$\frac{\langle S, e_1 \rangle \to \langle S_1, l^Q \rangle \qquad \langle S_1, e_2 \rangle \to \langle S_2, v^{Q'} \rangle \qquad l \in dom(S_2)}{\langle S, e_1 \texttt{:=} e_2 \rangle \to \langle S_2[l \mapsto v^{Q'}], v^{Q'} \rangle} \text{[Assign]}$$

$$\frac{\langle S, e \rangle \to \langle S_1, v^{Q'} \rangle \qquad Q' \leq Q}{\langle S, \texttt{check}(e, Q) \rangle \to \langle S_1, v^{Q'} \rangle} \text{[Check]}$$

Figure 3.8. Big-Step Operational Semantics with Qualifiers

Chapter 4

Refinements

Type-qualifier inference provides the foundation for the algorithms developed in this thesis, but early experiments uncovered numerous deficiencies in the stock type-qualifier inference algorithms. In this chapter, we describe several algorithmic refinements that were necessary to reduce false-positives, correctly analyze C code, and report the inference results usefully to the developer.

4.1 Parametric Polymorphism

There is a well-known problem with standard monomorphic type systems, like the one we have presented so far: multiple calls to the same function are conflated, leading to a loss of precision. For example, suppose we have qualifier constants a and b with partial order $\mathsf{b} < \mathsf{a}$, and consider the following two function definitions of the identity function, where we have annotated the argument with a qualified type:

$\texttt{let } id_1 = \lambda x{:}(\mathsf{a}\ \ int).x$

$\texttt{let } id_2 = \lambda x{:}(\mathsf{b}\ \ int).x$

We would like to have only a single copy of this function, since both versions behave the same and in fact compile to the same code. Unfortunately, without polymorphism we need both. The return type of id_1 must be $\mathsf{a}\ \ int$, and thus an object of type $\mathsf{b}\ \ int$ can be passed

to *id1*, but the return value has qualifier a. The argument type of id_2 must be $\mathsf{b}\ int$, and thus an object of type $\mathsf{a}\ int$ cannot be passed to `id2`. The problem here is that the type of the identity function on integers is $Q\ int \longrightarrow Q\ int$ with Q appearing both covariantly (to the right of the arrow) and contravariantly (to the left of the arrow).

Notice, however, that the identity function behaves the same for any qualifier Q. We specify this in type notation with the *parametric polymorphic* type signature [55] $\forall\kappa.\kappa\ int \longrightarrow \kappa\ int$. When we apply a function of this type to an argument, we first *instantiate* its type at a particular qualifier, in our case either as $\mathsf{a}\ int \longrightarrow \mathsf{a}\ int$ or $\mathsf{b}\ int \longrightarrow \mathsf{b}\ int$.

The traditional way to add polymorphism to a constraint-based type inference system is to use *polymorphically constrained types* [26, 62]. In this approach, we modify our type grammar as follows:

$$
\begin{array}{rcl}
\sigma & ::= & \forall\vec{\kappa}[C].\tau \\
\tau & ::= & Q\ \nu \\
\nu & ::= & int \mid ref\ (\tau) \mid \tau \longrightarrow \tau \\
C & ::= & \emptyset \mid \{Q \leq Q\} \mid C \cup C
\end{array}
$$

The type $\forall\vec{\kappa}[C].\tau$ represents all types of the form $\tau[\vec{\kappa} \mapsto \vec{Q}]$ for any $\vec{Q}$ that satisfies the constraints $C[\vec{\kappa} \mapsto \vec{Q}]$. Note that polymorphism only applies to the qualifiers and not to the underlying types. Adding polymorphism only serves to make type qualifier inference more precise, and it does not affect the operational semantics. We can prove soundness for such a system using standard techniques [90, 26, 62, 58].

In practice, polymorphically constrained type inference systems are tricky to implement directly. Instead, we use an equivalent formulation based on instantiation constraints, due to Rehof et al. [73]. In this approach, inferring polymorphic qualifiers is reduced to a context-free language (CFL) reachability problem [75] on the qualifier constraints viewed as a graph. The CFL reachability problem can be solved in cubic time. This formulation has the added advantage of naturally supporting polymorphic recursion, which our implementation does as well.

$$\frac{\Gamma \vdash' v : \tau_1 \qquad \Gamma[x \mapsto (\tau_1, fv(\Gamma))] \vdash' e : \tau_2}{\Gamma \vdash' \texttt{let}\, x = v \,\texttt{in}\, e : \tau_2} \text{ (Let}_{\text{CFL}}\text{)}$$

$$\frac{\Gamma(x) = (\tau, \vec{\kappa}) \qquad \tau' = \mathit{fresh}(\tau) \qquad \tau \xrightarrow{)_i} \tau' \qquad \kappa \xrightarrow{(_i} \kappa, \kappa \xrightarrow{)_i} \kappa \quad (\forall \kappa \in \vec{\kappa})}{\Gamma \vdash' x^i : \tau'} \text{ (Var}_{\text{CFL}}\text{)}$$

(a) Type Rules

$$\begin{array}{rcl}
C \cup \{Q \;\; int \xrightarrow{(_i} Q' \;\; int\} & \Rightarrow & C \cup \{Q \xrightarrow{(_i} Q'\} \\
C \cup \{Q \;\; int \xrightarrow{)_i} Q' \;\; int\} & \Rightarrow & C \cup \{Q' \xrightarrow{)_i} Q\} \\
C \cup \{Q \;\; \mathit{ref}\,(\tau) \xrightarrow{(_i} Q' \;\; \mathit{ref}\,(\tau')\} & \Rightarrow & C \cup \{Q \xrightarrow{(_i} Q'\} \cup \{\tau \xrightarrow{(_i} \tau'\} \cup \{\tau' \xrightarrow{)_i} \tau\} \\
C \cup \{Q \;\; \mathit{ref}\,(\tau) \xrightarrow{)_i} Q' \;\; \mathit{ref}\,(\tau')\} & \Rightarrow & C \cup \{Q \xrightarrow{)_i} Q'\} \cup \{\tau \xrightarrow{)_i} \tau'\} \cup \{\tau' \xrightarrow{(_i} \tau\} \\
C \cup \{Q\; (\tau_1 \longrightarrow \tau_2) \xrightarrow{(_i} Q'\; (\tau_1' \longrightarrow \tau_2')\} & \Rightarrow & C \cup \{Q \xrightarrow{(_i} Q'\} \cup \{\tau_1' \xrightarrow{)_i} \tau_1\} \cup \{\tau_2 \xrightarrow{(_i} \tau_2'\} \\
C \cup \{Q\; (\tau_1 \longrightarrow \tau_2) \xrightarrow{)_i} Q'\; (\tau_1' \longrightarrow \tau_2')\} & \Rightarrow & C \cup \{Q \xrightarrow{)_i} Q'\} \cup \{\tau_1' \xrightarrow{(_i} \tau_1\} \cup \{\tau_2 \xrightarrow{)_i} \tau_2'\}
\end{array}$$

(b) Structural Edge Generation Rules

Figure 4.1. Polymorphic Constraint Graph Construction

In this formulation, the nodes in the qualifier constraint graph are qualifier constants and variables. A qualifier constraint $Q \leq Q'$ generated by the rules in Figures 3.5 and 3.6a is represented by an unlabeled directed edge $Q \longrightarrow Q'$ from the node for Q to the node for Q'. Labeled edges will be used to represent qualifier instantiation, as discussed next.

Figure 4.1a extends the rules in Figure 3.5 to add polymorphism. As is standard, the rule (Let_{CFL}) introduces polymorphism, which is restricted to syntactic values (variables, integers, or functions in our language) [90] in order to be sound in the presence of updatable references. For a type τ, let the free variables of τ, $\mathit{fv}(\tau)$, be the set of all the qualifier variables that occur in τ. In the polymorphic qualifier system, the typing environment can map a variable to a type-scheme, represented as $\Gamma(x) = (\tau, S)$, where S is contains all the qualifier variables not quantified in τ. In particular, the type scheme (τ_1, S) corresponds to the polymorphically constrained type $\forall \vec{\kappa}[C].\tau_1$ where $\vec{\kappa} = (\mathit{fv}(\tau_1) \cup \mathit{fv}(C)) - S$. Let

$$\mathit{fv}(\Gamma) = \bigcup_{\tau=\Gamma(x)} \mathit{fv}(\tau) \quad \bigcup \bigcup_{(\tau,S)=\Gamma(x)} S$$

In rule (Let_{CFL}), we bind variable x to a pair containing its type and $\mathit{fv}(\Gamma)$, the set of qualifier variables that *cannot* be quantified in the current typing environment[44].

Each instantiation occurs at a particular syntactic location in the program, which we associate with an index i. Rule (Var_{CFL}) instantiates the type of variable x, which is labeled with index i. We create a type $\tau' = \mathit{fresh}(\tau)$, where $\mathit{fresh}(\tau)$ is a type with the same shape as τ but fresh qualifier variables. Then we make an instantiation constraint $\tau \xrightarrow{)_i} \tau'$, represented as a labeled directed edge (we write the edge as a hypothesis to the rule). Intuitively this corresponds to a substitution S_i, where $S_i\tau = \tau'$. (See [73, 44] for details.) We also add self loops labeled with both $(_i$ and $)_i$ for each $\kappa \in \vec{\kappa}$, i.e., for each qualifier variable that (Let_{CFL}) determined was not generalizable. Intuitively this corresponds to requiring $S_i\kappa = \kappa$, meaning κ was instantiated to itself, i.e., κ was not actually instantiated.

After we have generated qualifier constraints and instantiation constraints, we apply the rules in Figure 4.1b exhaustively to propagate instantiation constraints from types to qualifiers, taking contravariance into account by flipping the kind of parenthesis and the direction of the edge. Given this graph, the CFL reachability problem is to find all paths

in the graph made up of edges whose labels do not contain any mismatched parentheses (ignoring the unlabeled edges), where only open and closed parentheses with the same index match [73]. We call such a path a realizable.

As an example, consider again the identity function, with two uses:

let $id = \lambda x\colon int\,.x$ in

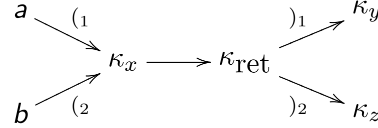
let $a =$ annot$(0, \mathsf{a})$ in

let $b =$ annot$(1, \mathsf{b})$ in

let $y = id^1\ a$ in

let $z = id^2\ b$ in 42

In this program, a has qualifier a and b has qualifier b, and we have labeled the two uses of id with location 1 and location 2. Then the CFL reachability graph generated by this example looks like the following:



Here κ_{ret} is the qualifier variable on the return type of id, and κ_x, κ_y, and κ_z are the qualifier variables for x, y, and z, respectively. Since we forbid paths with mismatched parentheses, there are only two paths through this graph: from a to κ_y, indicating that y should be qualified with a, and from b to κ_z. The other paths, from a to κ_z and from b to κ_y, are *unrealizable* [75], since they correspond to a call at position 1 followed by a return at position 2 and vice-versa. The CFL reachability technique eliminates these unrealizable paths from inference, giving us polymorphism.

We do not require all the parenthesis on a path to be matched because, for example, a function could return an $a\ \ int$ to a loca variable t, which is then passed into a function expecting a $b\ \ int$. This would induce the qualifier constraints $\mathsf{a} \xrightarrow{)_1} \mathsf{t} \xrightarrow{(_2} b$, which is an error if the qualifier lattice declares that $\mathsf{a} \not\leq \mathsf{b}$.

We model global variables by effectively treating them as pass-by-reference parameters to every function. To achieve this, we mark qualifier variables decorating global program

variables as “global” and treat them as if they have self-edges with every parenthesis in the language.

Efficient CFL Reachability With Globals Previous results on CFL reachability do not yield an efficient algorithm for the reachability problem that arises from type-qualifier inference of C programs. The standard CFL reachability algorithm computes an all-pairs reachability relation and runs in $O(n^3)$ time. In the specific case of type-qualifier inference, though, we can do much better. First, we don't need to compute all-pairs reachability, just the reachability relationships of the different qualifier constants in the system. Second, the CFL graphs generated by CQual have special structure because they are derived from procedural programs written by humans. Horwitz, Reps, and Sagiv presented a matched-parenthesis reachability algorithm that runs in linear time under a few basic assumptions about the input CFL graph[46]. Their algorithm is not directly applicable here, though, because they did not have global nodes in their graphs. Subsequently, Das, et al. sketched a “globals optimization” to the HRS algorithm, where they use the same representation for globals given above, but did not describe it in detail or claim that it beat the generic $O(n^3)$ running time[20].

Before describing our CFL-reachability algorithm, we should introduce several related languages.

Definition 5 *Let* MP *be the language of matched parenthesis. Let* NMP *be the language of strings with no mismatched parenthesis. Let* CP *be the language of strings with no mismatched parenthesis and no unmatched open parenthesis. Let* OP *be the language of strings with no mismatched parenthesis and no unmatched close parenthesis. Note that* $\mathrm{MP} \subset \mathrm{CP} \subset \mathrm{NMP}$ *and* $\mathrm{MP} \subset \mathrm{OP} \subset \mathrm{NMP}$*. If there exists a path from x to y whose edge labels spell the word $w \in L$, then we write $x \overset{w}{\leadsto} y$ and $x \overset{L}{\leadsto} y$.*

The grammars for these languages are given in Figure 4.2.

Recall that a global node is a node, g, that has self loops $g \xrightarrow{(_i} g$ and $g \xrightarrow{)_i} g$ for every

$$
\begin{array}{rcl}
S_{\text{MP}} & ::= & S_{\text{MP}} S_{\text{MP}} \\
& | & \epsilon \\
& | & (_i S_{\text{MP}})_i \\
\\
S_{\text{NMP}} & ::= & S_{\text{CP}} S_{\text{OP}}
\end{array}
\qquad
\begin{array}{rcl}
S_{\text{CP}} & ::= & S_{\text{CP}} S_{\text{CP}} \\
& | & S_{\text{MP}} \\
& | &)_i \\
S_{\text{OP}} & ::= & S_{\text{OP}} S_{\text{OP}} \\
& | & S_{\text{MP}} \\
& | & (_i
\end{array}
$$

Figure 4.2. Context-free Grammars for MP, CP, OP, and NMP.

$(_i$ and $)_i$ that occurs in the CFL grammar. In our graphs, these edges are represented implicitly by storing a bit with each node indicating whether or not it is global.

Definition 6 *A g-graph is a graph, $G = (V, E)$ with edge labels $(_i,)_i$, and ϵ, together with a function* global(v) *indicating whether each vertex in V is global.*

We can convert a g-graph, G, into a regular labeled graph via $f(G) = G' = (V, E')$, where

$$E' = E \bigcup \{ v \xrightarrow{(_i} v, v \xrightarrow{)_i} v | v \in V, \text{global}(v) = 1, (_i,)_i \text{ in grammar} \}$$

For a g-graph, we write $x \overset{w}{\rightsquigarrow} y$ and $x \overset{L}{\rightsquigarrow} y$ if $f(G)$ contains a path $x \rightsquigarrow y$ whose labels spell the word $w \in L$.

We wish to develop an efficient algorithm to determine, given a g-graph G and nodes $s_0, t_0 \in G$, whether $s_0 \overset{\text{NMP}}{\rightsquigarrow} t_0$.

We reduce the problem of finding an NMP path to finding CP paths as follows. As can be seen from the grammar in Figure 4.2, every NMP path can be decomposed into a CP path and an OP path. Instead of searching for an NMP path $s_0 \rightsquigarrow t_0$ directly, we instead compute the set, S, of vertices that are CP-reachable from s_0 and the set, T, of nodes that can reach t_0 via an OP path. We then check that $S \cap T = \emptyset$. Furthermore, we can compute T by computing the set of nodes that are CP-reachable from t_0 in the reverse graph of G, i.e. the graph obtained by reversing all the edges in G and changing all open-parenthesis labels to close-parentheses and all close-parenthesis labels to open-parentheses. As a result, we only need to develop a CP-reachability algorithm to solve this problem.

We now present a CP-reachability algorithm for g-graphs that efficiently handles global nodes and runs in linear-time when the input graph satisfies a few conditions given below. The algorithm is best expressed as a set of closure rules, shown in Figure 4.3, for computing a relation, $\twoheadrightarrow$, on V, a function, $R : V \to \{0, 1\}$, and a new graph, G'. The $\twoheadrightarrow$ relation represents MP suffixes of NMP paths in G, and we will refer to entries in this relation simply as "paths". The function R indicates variables that are CP-reachable from s_0. The graph G' contains some additional edges to summarize MP paths in G.

The closure rules have several subtleties. There is no transitive closure rule $a \twoheadrightarrow b \twoheadrightarrow c \Rightarrow a \twoheadrightarrow c$. This rule is not needed because the algorithm is essentially a graph exploration (e.g. a breadth-first or depth-first search, depending on the order in which the closure rules are applied). However, the algorithm must summarize MP paths for future searches (Rule 7) and restart old searches when a summary of an MP path is created.

Note that Rule 7 must create a summary edge $r \to c$ instead of the path $r \twoheadrightarrow c$ for two reasons. First, since there is no transitive closure rule, the path $r \twoheadrightarrow c$ would not be detected by future searches. Second, the $\twoheadrightarrow$ relation describes MP paths that are also suffixes of NMP paths from s_0. In the case of Rule 7, there may not be any such path to c. If the closure rules arrived at b via some path that does not cross the edge $r \stackrel{(}{\to} b$, then concluding that $r \twoheadrightarrow c$ would be unsound.

The rules presented here also contain several small optimizations. For example, if the $\times$ mark on c in Rule 5 were erased, the algorithm would still be correct and would run in linear time under the same assumptions. We choose to present the algorithm with all optimizations in place, though, because these are the closure rules actually implemented in CQUAL. By working with these rules instead of cleaned-up, simplified rules, we gain a higher degree of confidence that the implementation is correct.

Figure 4.4 shows an algorithm for implementing these closure rules.

Theorem 7 ***(Correctness)*** *Let G be a g-graph and let $s_0 \in G$. Let $(G', R) =$ cp-reachable(G, s_0). Then*

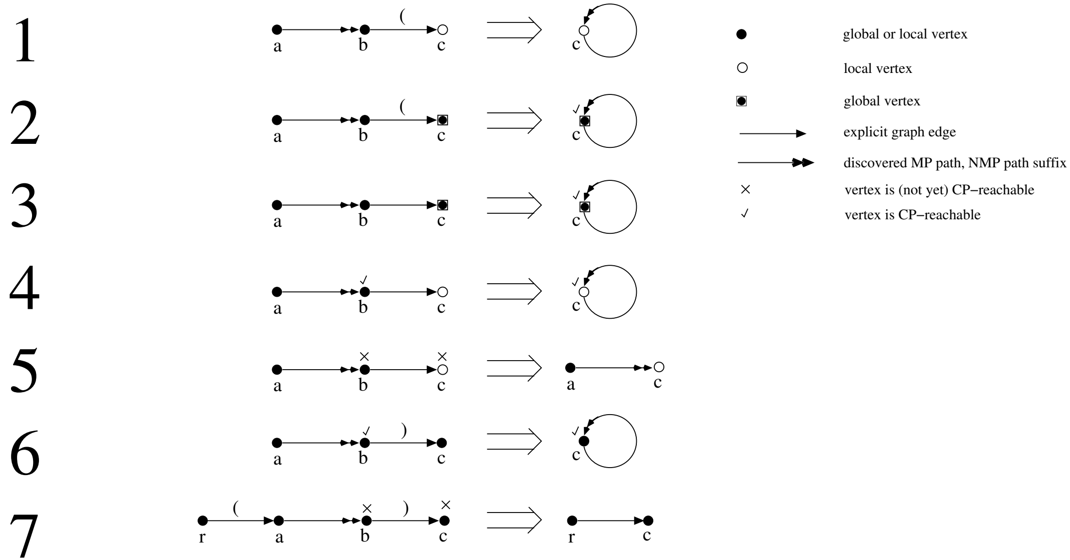


Figure 4.3. Closure rules for efficiently computing the the set of vertices reachable from a vertex, s_0, via a CP path. In these rules, $x \twoheadrightarrow y$ stands for an MP-path from x to y that is also a suffix of an NMP-path $s_0 \rightsquigarrow y$ that has been discovered by the closure algorithm. The algorithm maintains a bit, $R(v)$, for each vertex v, indicating whether it has discovered a CP-path to v. The templates on the LHS of each rule only apply when R has the values indicated by the $\times$ and $\checkmark$ symbols. The generated paths on the RHS may modify R as indicated. The rules must be primed with an initial MP path $s_0 \twoheadrightarrow s_0$, and s_0 should be marked as reachable, i.e. $R(s_0) = 1$. The rules should not match implicit or explicit self-loops on globals, e.g. Rule 1 should only match an explicitly represented edge $b \stackrel{(}{\rightarrow} c$ where $b \neq c$ or b is not global. Rule 7 requires the open and close parentheses to match. Note that, if a rule changes $R(v)$ to 1, then the resulting path should be considered new and subject to further closure rules.

Worklist = $\emptyset$
Workedlist=$\emptyset$
procedure newpath(a, b, r) *[Discovered path $a \twoheadrightarrow b$ and $R(b) = r$]*
  if $a \twoheadrightarrow b \notin$ Worklist $\cup$ Workedlist or $R(b) < r$
    insert $a \twoheadrightarrow b$ into Worklist
    $R(b) := \max(R(b), r)$

procedure cp-reachable(G, s_0)
  $(V, E) = G$
  for each $v \in V$
    $R(v) := 0$
  $R(s_0) := 1$
  Worklist = $\{s_0 \twoheadrightarrow s_0\}$
  Workedlist = $\emptyset$
  while Worklist is not empty
    remove $a \twoheadrightarrow b$ from Worklist
    add $a \twoheadrightarrow b$ to Workedlist
    for each edge $b \xrightarrow{(_i} c$
      newpath(c, c, global(c)) *[Rules 1 and 2]*
    for each edge $b \rightarrow c$
      if global(c) or $R(b) = 1$
        newpath(c, c, 1) *[Rules 3 and 4]*
      else
        newpath(a, c, 0) *[Rule 5]*
    for each edge $b \xrightarrow{)_i} c$
      if $R(b) = 1$
        newpath(c, c, 1) *[Rule 6]*
      else
        for each edge $r \xrightarrow{(_i} a$
          if $R(c) = 0$
            add edge $r \rightarrow c$ to G *[Rule 7]*
            [Apply Rules 3-5 to old paths and the new edge]
            if $R(r) = 1$
              newpath(c, c, 1) *[Rule 4]*
            else
              if global(c)
                if there exists a path $s \twoheadrightarrow r \in$ Workedlist
                  newpath(c, c, 1) *[Rule 3]*
              else if $R(c) = 0$
                for each path $s \twoheadrightarrow r \in$ Workedlist
                  newpath(s, c, 0) *[Rule 5]*
  return (G, R)

Figure 4.4. CP reachability algorithm

1. $x \twoheadrightarrow y$ *only if* $x \overset{\text{MP}}{\leadsto} y$.

2. $x \rightarrow y \in G'$ *only if* $x \overset{\text{MP}}{\leadsto} y \in G$.

3. $x \twoheadrightarrow y$ *if and only if* $s_0 \overset{\text{NMP}}{\leadsto} y$.

4. $R(y) = 1$ *if and only if* $s_0 \overset{\text{CP}}{\leadsto} y$.

5. $x \twoheadrightarrow y$ *or* $R(y) = 1$ *if* $s_0 \overset{\text{NMP}}{\leadsto} x' \overset{(}{\rightarrow} x \overset{\text{MP}}{\leadsto} y$.

Proof: The "only if" direction of the proof is a straightforward verification that each of the closure rules preserves the invariants listed above.

Before embarking on the "if" direction, observe that the rules guarantee that, if $R(y) = 1$, then the algorithm must have already discovered the path $y \twoheadrightarrow y$.

We prove the "if" direction by strong induction on the length of the path required in each invariant. Recall that, for g-graphs, $x \overset{L}{\leadsto} y$ refers to paths in $f(G)$. Thus the induction is on the length of the path in $f(G)$. For $n = 0$, the only path of length 0 is s_0, and by the initialization of the algorithm, $s_0 \twoheadrightarrow s_0$ and $R(s_0) = 1$, establishing the base case for inductive hypotheses (IH) 3 and 4. There is no length-0 path of the form required by IH 5, so it is trivially satisfied. Now suppose that all the IHs hold for all paths of length less than n. We need to prove that each hypothesis holds for paths of length n.

IH 3. Let $s_0 \overset{c_1}{\rightarrow} y_1 \overset{c_2}{\rightarrow} \cdots \overset{c_n}{\rightarrow} y_n$ be an NMP path of length n. The prefix of this path $s_0 \leadsto y_{n-1}$ is also an NMP path, so, by IH 3, the algorithm will at some point discover a path $x \twoheadrightarrow y_{n-1}$. We may assume that $R(y_n) = 0$, since otherwise the algorithm must have discovered some path $y_n \twoheadrightarrow y_n$. If $c_n = ($, then Closure Rule 1 or 2 will apply, yielding a path $y_n \twoheadrightarrow y_n$. If $c_n = \epsilon$, then one of Closure Rules 3-5 will apply.

Now suppose $c_n =)$. If c_n is not matched in $c_1 \cdots c_n$, then $c_1 \cdots c_{n-1}$ must not contain any unmatched open parenthesis, i.e. $c_1 \cdots c_{n-1} \in \text{CP}$. By IH 4, $R(y_{n-1}) = 1$. Furthermore, this can only occur if the algorithm has already discovered the path $y_{n-1} \twoheadrightarrow y_{n-1}$. Closure Rule 6 will then apply. So suppose c_n is matched in $c_1 \cdots c_n$ by some $c_j = ($. Note that, in this case, we must have $c_{j+1} \cdots c_{n-1} \in \text{MP}$. Thus we have a path of length n of the form

$s_0 \overset{\text{NMP}}{\rightsquigarrow} y_{j-1} \overset{(}{\rightarrow} y_j \overset{\text{MP}}{\rightsquigarrow} y_{n-1} \overset{)}{\rightarrow} y_n$. By IH 3, the rules will discover some path $x \twoheadrightarrow y_{j-1}$. By IH 5, $y_j \twoheadrightarrow y_{n-1}$ or $R(y_{n-1}) = 1$. If $R(y_{n-1}) = 1$ then, as above, we will discover the path $y_n \twoheadrightarrow y_n$. So suppose $R(y_{n-1}) = 0$. In this case, Rule 7 will create edge $y_{j-1} \rightarrow y_n$, and then one of Rules 3-5 will apply to the already discovered path $x \twoheadrightarrow y_{j-1}$ and the new edge.

IH4. Let $s_0 \overset{c_1}{\rightarrow} y_1 \overset{c_2}{\rightarrow} \cdots \overset{c_n}{\rightarrow} y_n$ be a CP path of length n. In this case, $c_n \neq ($. If the prefix of this path to y_{n-1} is also a CP path, then, by IH 4, $R(y_{n-1}) = 1$, so one of Rules 4 or 6 will set $R(y_n) = 1$. If the prefix of this path is not CP, then c_n must be matched by some c_j in $c_1 \cdots c_n$. As before, we must have $c_{j+1} \cdots c_{n-1} \in \text{MP}$, and in fact our path is of the form $s_0 \overset{\text{CP}}{\rightsquigarrow} y_{j-1} \overset{(}{\rightarrow} y_j \overset{\text{MP}}{\rightsquigarrow} y_{n-1} \overset{)}{\rightarrow} y_n$. If the algorithm ever sets $R(y_{n-1}) = 1$, then Rule 6 will apply, so suppose this never happens. By IH 4, $R(y_{j-1}) = 1$ and, by IH 5, $y_j \twoheadrightarrow y_{n-1}$. Because of the path $y_j \twoheadrightarrow y_{n-1}$, Rule 7 will create the edge $y_{j-1} \rightarrow y_n$. Rule 3 or 4 applied to the path $y_{j-1} \twoheadrightarrow y_{j-1}$ and the new edge will set $R(y_n) = 1$.

IH5. Let $s_0 \overset{c_1}{\rightarrow} y_1 \overset{c_2}{\rightarrow} \cdots \overset{c_n}{\rightarrow} y_n$ be a path of the form $s_0 \overset{\text{NMP}}{\rightsquigarrow} y_j \overset{(}{\rightarrow} y_{j+1} \overset{\text{MP}}{\rightsquigarrow} y_n$. If $c_n = ($, then $n = j + 1$, and by IH 3, $x \twoheadrightarrow y_j$ for some x. In this case, Rule 1 or 2 will construct the required path $y_n \twoheadrightarrow y_n$. If $c_n = \epsilon$, then the prefix of this path is of the form $s_0 \overset{\text{NMP}}{\rightsquigarrow} y_j \overset{(}{\rightarrow} y_{j+1} \overset{\text{MP}}{\rightsquigarrow} y_{n-1}$. By IH 5, $y_{j+1} \twoheadrightarrow y_{n-1}$ or $R(y_{n-1}) = 1$. If $R(y_{n-1}) = 1$, then Rule 3 or 4 will set $R(y_n) = 1$. If $R(y_{n-1}) = 0$, then Rule 3 will set $R(y_n) = 1$ if y_n is global and Rule 5 will create path $y_{j+1} \twoheadrightarrow y_n$ othwerwise.

So suppose $c_n =)$ and consequently is matched by some c_k. Observe that $k > j + 1$ and $c_{k+1} \cdots c_{n-1} \in \text{MP}$. As a consequence, the path is of the form

$$s_0 \overset{\text{NMP}}{\rightsquigarrow} y_j \overset{(}{\rightarrow} y_{j+1} \overset{\text{MP}}{\rightsquigarrow} y_{k-1} \overset{(}{\rightarrow} y_k \overset{MP}{\rightsquigarrow} y_{n-1} \overset{)}{\rightarrow} y_n$$

By two applications of IH 5, $y_{j+1} \twoheadrightarrow y_{k-1}$ or $R(y_{k-1}) = 1$ and $y_k \twoheadrightarrow y_{n-1}$ or $R(y_{n-1}) = 1$. If $R(y_{n-1}) = 1$, then Rule 6 will set $R(y_n) = 1$, so suppose $R(y_{n-1}) = 0$. In this case, Rule 7 will add the edge $y_{k-1} \rightarrow y_n$. If $R(y_{k-1}) = 1$, then Rule 3 or 4 will set $R(y_n) = 1$, so suppose $R(y_{k-1}) = 0$. By IH 3, there exists an x such that $x \twoheadrightarrow y_{k-1}$, and Rule 5 will apply to this path and the edge $y_{k-1} \rightarrow y_n$, creating path $x \twoheadrightarrow y_n$. □

The efficiency argument for this algorithm assumes that the graph is generated from a

program written in a procedural language and that the size of each procedure is bounded by a constant, M. We first prove that the algorithm is efficient whenever the graph G has a certain structure, and then argue informally that programs with bounded-sized functions will generate graphs with this structure.

Definition 8 *A path from a to b is global-free (*GF*) if it doesn't cross any global nodes, except possibly a and b. The path is totally global-free (*TGF*) if it is global-free and a and b are not global. Let*

$$S_x = \begin{cases} \{x\} & \text{if } x \text{ is global} \\ \{y \mid x \overset{\text{MP,TGF}}{\rightsquigarrow} y \in G\} & \text{otherwise} \end{cases}$$

Also, set $P_x = \{y \mid x \in S_y\}$ and $T_x = \{y \mid x \overset{\text{MP,GF}}{\rightsquigarrow} y\} \in G$.

Lemma 9 *The closure rules maintain the following invariants:*

- $a \to b \in G'$ *only if* $a \overset{\text{MP,GF}}{\rightsquigarrow} b \in G$.
- $a \twoheadrightarrow b$ *for* $a \neq b$ *only if* $a \overset{\text{MP,TGF}}{\rightsquigarrow} b \in G$.

Proof: It is straightforward to prove that all the closure rules maintain these invariants. □

Theorem 10 ***(Efficiency)*** *Suppose that for all $x \in G$, $|P_x| \leq P_{\max}, |S_x| \leq S_{\max}$, and $\sum_x |T_x| = O(m+n)$. If G contains n nodes and m edges, then the running time of the algorithm is $O(m+n)$.*

Proof: Let m' be the number of edges in G'. Closure rule 7 will never add more than $|T_x|$ outbound edges to any node x, so

$$m' = \sum_x \text{outdegree}_{G'}(x) \leq \sum_x (\text{outdegree}_G(x) + |T_x|) = m + O(m+n)$$

Hence $O(m') = O(n+m)$, so we only need to show the running time of the algorithm is $O(m')$. We analyze each closure rule separately. We charge the running time of rules 1-6

to the node at the end of the causal path, b. These rules are invoked for every discovered path ending at b, and take $\text{outdegree}_{G'}(b)$ steps to execute. The CFL algorithm discovers $O(|P_b|)$ paths ending at b, so the total charge to any node b is $O(|P_b|\text{outdegree}_{G'}(b))$. By using the bound $|P_b| \leq P_{\max}$, the total cost of rules 1-6 is $O(m')$.

In the analysis of rules 7, we assume that, given an inbound edge $r \xrightarrow{(} a$, we can iterate over the set of all matching-parenthesis outbound edges $b \xrightarrow{)} c$ in time linear in the size of the matching set.

For Rule 7, we charge the cost to the pair (a, b). We must iterate over all the inbound (edges of a and the matching outbound edges of b. For each edge $r \xrightarrow{(} a$, though, there can be at most $T_r < T_{\max}$ matching outbound edges of b. Thus the total cost to the pair (a, b) is bounded by $\text{indegree}_{G'}(a)T_{\max}$. For a given node a, there are at most S_a pairs of the form (a, b) with nonzero cost, so the total cost from pairs beginning at a is at most $S_a\text{indegree}_{G'}(a)T_{\max}$. Summing over all a and using the bound $S_a \leq S_{\max}$ gives an overall bound of $O(m')$. □

Type-qualifier inference generates a qualifier variable for each position in the type of each variable (parameters, locals, and globals) and subexpression in the program. Unlabeled edges in the graph correspond to statements in the program that are not function calls, and the labeled edges connect actual to formal parameters and return values. For a procedure p, let V_p be the set of qualifier variables that decorate some program variable or expression that is directly referenced by p. In other words, V_p contains the qualifiers for the subexpressions and local variables of p and the global variables that p references directly, i.e. if a pointer to a global variable is passed into p, then V_p should contain the qualifiers on the local pointer's type, but not necessarily on the global variable itself.

Let P be the set of procedures in the input program, and suppose that $|V_p| \leq V_{\max}$ for all $p \in P$. Suppose also that $|P| = O(n)$, e.g. most procedures have at least one local variable or expression. For any qualifier variables x and y, $x \overset{MP,GF}{\rightsquigarrow} y$ implies that x and y are directly referenced by some common function, p, i.e. $x \in V_p$ and $y \in V_p$. There are

at most $|V_p|^2 \leq V_{\max}^2$ such pairs of variables, so $\sum_x |T_x| \leq \sum_p |V_{\max}|^2 = O(n+m)$. Similar arguments show that $|S_x| \leq V_{\max}$ and $|P_x| \leq V_{\max}$ for all x. Thus the CFL reachability algorithm above will run in $O(n+m)$ time. Since the size of the graph constructed during type-qualifier inference is linear in the size of the input program, the whole process runs in time linear in the size of the input program.

The CQual implementation of CFL reachability deviates from the bounds achieved here in one way: It does not support efficiently looking up the matching parenthesis edges needed for achieving the time-bound for closure Rule 7. It would be possible to add support for this operation to CQual, but so far we have found it unnecessary.

4.2 Structures

One of the key considerations in any whole-program analysis of C code is how structures (record types) are modeled [12, 43, 94]. In Chapter 3, we used the *embed′* operation to give every expression in the program its own type annotated with fresh qualifier variables. Recall the definition of *embed′*:

$$\begin{array}{rcll} \mathit{embed}'(\mathit{int}) & = & \kappa\ \mathit{int} & \kappa \text{ fresh} \\ \mathit{embed}'(\mathit{ref}\ (s)) & = & \kappa\ \mathit{ref}\ (\mathit{embed}'(s)) & \kappa \text{ fresh} \\ \mathit{embed}'(s \longrightarrow s') & = & \kappa\ (\mathit{embed}'(s) \longrightarrow \mathit{embed}'(s')) & \kappa \text{ fresh} \end{array}$$

The most natural extension of the basic qualifier system to structures is to treat `struct` as an n-ary type constructor: [1]

$$\mathit{embed}'(\mathit{struct}\ (s_1, \ldots, s_n)) \quad = \quad \kappa\ \mathit{struct}\ (\mathit{embed}'(s_1), \ldots, \mathit{embed}'(s_n)) \quad \kappa \text{ fresh}$$

Programmers can declare recursive types in C, and the standard way of dealing with recursive types is to turn the type-tree into a type-graph, i.e. every recursive instance of a type is given the same type as its root. We can adapt *embed′* to handle recursive types in the same way.

[1] Technically, `struct` is not an n-ary type constructor because two structure definitions with identical field layouts but with different names for the `struct`s are considered different types in C. However, for the purposes of type-qualifier inference, it is safe to model `struct`s as simple record types.

Suppose that the programmer declares a structure for `s100`:

```
struct s1 { int a; int b; };
struct s2 { struct s1 *a; struct s1 *b; };
struct s3 { struct s2 *a; struct s2 *b; };
struct s4 { struct s3 *a; struct s3 *b; };
...
struct s100 { struct s99 *a; struct s99 *b; };
```

If we see two variable declarations `struct s100 a` and `struct s100 b` this definition of *embed′* will assign `a` and `b` two distinct copies of the type `struct s100`. Unfortunately, the full type of `struct s100` contains $2^{101} - 1$ different positions (one for every field reachable through pointers contained in a `struct s100`), so we would need to create $2^{101} - 1$ fresh qualifier variables for every instance, which is clearly impractical. This example shows that the naive definition of *embed′* has worst case exponential complexity. Real C programs don't declare `struct`s like this, but they use structures that are sufficiently complex to make this approach infeasible. We implemented this method and were unable to analyze programs as short as 5000 lines without running out of memory.

An alternative approach to modeling `struct`s is to completely dissociate fields from their containing structure. For example, the declaration of `struct s1` would be treated as a declaration of two global variables, `a` and `b`. Any reference to field `a` of an instance of `struct s1` would be treated as a reference to the global variable `a`. In this model, `struct` is a 0-ary type constructor, so the definition of *embed′* is much simpler:

$$embed'(struct\,()) \quad = \quad \kappa \;\; struct\,() \quad \kappa \text{ fresh}$$

This model for structures only requires a linear number of qualifiers but can generate numerous false positives from the false sharing of structure fields.

To solve this problem, we use a hybrid approach, as layed out in Figure 4.5. In our system, `struct` is a type constructor that takes as its argument a mapping, F, from field names to their types. The *embed′* operation gives each `struct` its own mapping that is

$$embed'(struct\,()) \;=\; \kappa\;\; struct\,(F) \quad F,\kappa \text{ fresh}$$

(a) Definition of *embed′* for `struct` types

$$\frac{\Gamma \vdash e : Q\;\; struct\,(F) \qquad F(f) = \tau}{\Gamma \vdash e\;.\;f : \tau}\;(\text{Field}')$$

(b) Rule for Field References in Qualified Structure Types

$$\begin{aligned}
C \cup \{Q\;\; struct\,(F) \leq Q'\;\; struct\,(F')\} &\;\Rightarrow\; C \cup \{Q \leq Q'\} \cup \{F = F'\} \\
C \cup \{Q\;\; struct\,(F) \xrightarrow{(_i} Q'\;\; struct\,(F')\} &\;\Rightarrow\; C \cup \{Q \xrightarrow{(_i} Q'\} \cup \{F = F'\} \\
C \cup \{Q\;\; struct\,(F) \xrightarrow{)_i} Q'\;\; struct\,(F')\} &\;\Rightarrow\; C \cup \{Q \xrightarrow{)_i} Q'\} \cup \{F = F'\}
\end{aligned}$$

(c) Resolution Rules for Subtyping Constraints of Structures

Figure 4.5. Qualified-Type Checking System Extended for Structures

initially undefined on all inputs. Whenever the programmer references a field `f` in expression e of type $Q\;\; struct\,(F)$, we constrain F's value at `f`. In other words, we define F lazily as the programmer references different fields of e. If the programmer ever makes a `struct` assignment or passes a structure as the argument to a function, this will generate constraints between their types, which are resolved as shown in Figure 4.5(c). Note that we conservatively equate the two types' field mappings, which has the effect of converting subtyping and polymorphic constraints into monomorphic equality constraints that can be resolved using a unification-based algorithm.

Figure 4.6 illustrates an efficient implementation of these typing rules. Each instance of a `struct` is given its own type, but we do not create fresh types for its fields unless we see them explicitly referenced in the source program. For example, type σ_1 in Figure 4.6(a) is for a `struct` variable from which the programmer has referenced fields 1 and 2, which were given fresh copies, τ_1 and τ_2, of their types. The type σ_2 is for a `struct` from which the programmer explicitly referenced the first and third fields. This guarantees that the number of fields created in this analysis is linear in the size of the input program.

When the type-qualifier inference analysis generates a constraint $\sigma_1 \leq \sigma_2$ between two

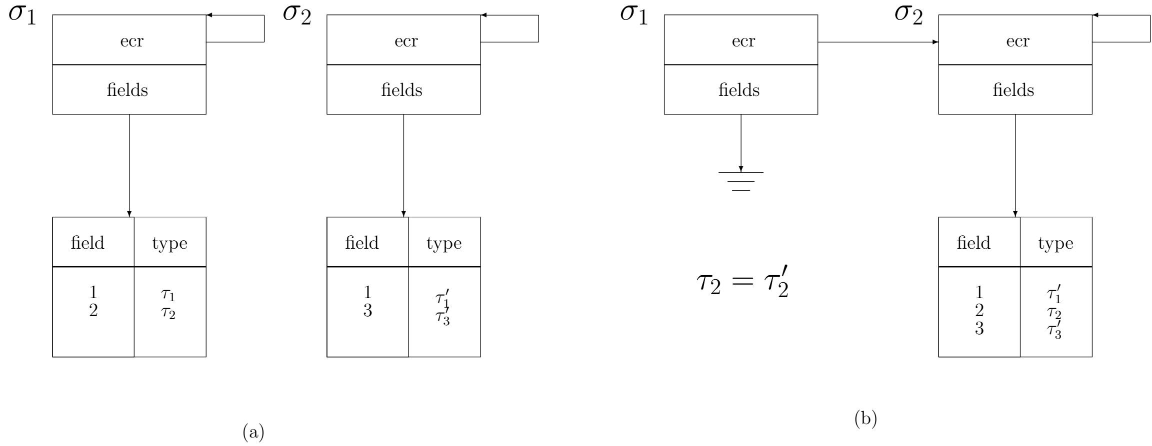


Figure 4.6. (a) Representation for `struct` types σ_1 and σ_2. The `ecr` field of a struct's type points to the equivalence class representative for that type. (b) σ_1 and σ_2 after processing the constraint $\sigma_1 = \sigma_2$. Processing this constraint generates the sub-constraint $\tau_1 = \tau'_1$.

`struct` types, we take each pair of matching fields, τ_i and τ'_i, and generate the constraint $\tau_i = \tau'_i$. We then take any other fields in σ_1 and add them to σ_2. Finally, we unify σ_1 with σ_2 so that all future references to σ_1 will resolve to σ_2. So, for example, in Figure 4.6, the only matching field in σ_1 and σ_2 is field 1, so we generate the constraint $\tau_1 = \tau'_1$. We then add τ_2 to σ_2's set of fields and point σ_1 to σ_2.

This typing scheme is efficient because it can be implemented using unification, but this precludes supporting subtyping and polymorphism for fields of structures, which can lead to imprecision. Figure 4.7 shows a simple C program that illustrates the imprecision of a unification-based analysis of fields. Type-qualifier inference will generate a polymorphic constraint between the qualified types of `x` and `*s`, which will be resolved by unifying the types of `x.a` and `(*s).a`. Similarly, the types of `y.a` and `(*s).a` will be unfied. Since `x.a` is tainted, `y.a` will also be tainted, causing an error when `y.a` is assigned to `u`.

C permits assignments between different `struct` types via casts. One structure may have more fields than another, or corresponding fields may differ in their type and, importantly, their size in memory. For these kinds of casts, CQual matches up as many fields as possible and ignores any extra fields that only appear in one of the structs. Also, the fields are matched by their index in the structure, i.e., the first fields are paired, then the second fields, etc., instead of matching by byte-offset.

```
struct S { int a; };
int get_a(struct S *s)
{
  return s->a;
}
void main(void)
{
  struct S x, y;
  int t;
  untainted int u; x.a = (tainted int)0;
  t = get_a(&x);
  t = get_a(&y);
  u = y.a;
}
```

Figure 4.7. A C program demonstrating the imprecision of field unification.

This method of modeling structures initially generates $O(n)$ fields. Each constraint $\sigma_1 \leq \sigma_2$ may generate subconstraints $\tau_i \leq \tau_i'$, which may in turn generate more subconstraints. However, every time a subconstraint is generated, one of the fields involved in that constraint is permanently eliminated by the system. The number of subconstraints generated during the analysis is therefore $O(n)$. We can charge the time required to process each subconstraint (excluding the time required to process its subconstraints) to the field that is eliminated by that constraint. Since each field can only be charged once before being eliminated, the total work required to dispatch all constraints is $O(n)$. We must also maintain the links between unified `struct` types. A union-find data structure can accomplish this in $O(n\alpha(n))$ time, where α is the inverse Ackerman function.

4.3 Type Casts

C programs cast pointers to and from integral types often enough that it is worth having special support for this programming idiom. We handle this case by treating every integral program variable as if it were a void pointer. For example, `int a` is given type $a\ \mathit{ref}\,(a'\ \mathit{void})$. All the standard type inference rules for void pointers are applied to `a`. This technique captures most of the casts between pointers and integral types, especially when combined with the special handling of casts between void pointers and other types described below.

This approach to modeling integers differs from prior approaches by being more precise and sound. Older implementations of type-qualifier inference in CQUAL handled these casts by collapsing the type of the pointer to match that of the integer. For example, the cast

```
char a * a' a;
int b b;
b = (int) a;
```

would generate constraints $a \leq b$ and $a' = b$. The second constraint can cause numerous false positives since it moves a qualifier up from the referent to the level of the pointer. Even worse, it's not sound:

```
char a * a' a;
int b b;
int c c;
char d * d' d;
b = (int) a;
c = b;
d = (char *) c;
```

Using the above hack, the statements in this code fragment entail qualifier constraints $a' = b \leq c = d'$, but since `a` and `d` point to the same location, we should have $a' = d'$. The new technique generates the correct constraints: $a' = b' = c' = d'$.

$$embed'(void) \quad = \quad \kappa \ void(F) \quad F, \kappa \text{ fresh}$$

(a) Definition of *embed′* for *void* types

$$\begin{aligned}
C \cup \{Q \ \ void(F) \leq Q' \ \ void(F')\} &\Rightarrow C \cup \{Q \leq Q'\} \cup \{F = F'\} \\
C \cup \{Q \ \ void(F) \xrightarrow{)_i} Q' \ \ void(F')\} &\Rightarrow C \cup \{Q \xrightarrow{)_i} Q'\} \cup \{F = F'\} \\
C \cup \{Q \ \ void(F) \xrightarrow{(_i} Q' \ \ void(F')\} &\Rightarrow C \cup \{Q \xrightarrow{(_i} Q'\} \cup \{F = F'\}
\end{aligned}$$

$$\begin{aligned}
C \cup \{Q \ \ void(F) \leq Q' \ \tau\} &\Rightarrow C \cup \{Q \leq Q'\} \cup \{F(strip(\tau) = \tau)\} \\
C \cup \{Q \ \ void(F) \xrightarrow{)_i} Q' \ \tau\} &\Rightarrow C \cup \{Q \xrightarrow{)_i} Q'\} \cup \{F(strip(\tau) = \tau)\} \\
C \cup \{Q \ \ void(F) \xrightarrow{(_i} Q' \ \tau\} &\Rightarrow C \cup \{Q \xrightarrow{(_i} Q'\} \cup \{F(strip(\tau) = \tau)\}
\end{aligned}$$

(b) Resolution Rules for Subtyping Constraints of Voids

Figure 4.8. Qualified-Type Checking System Extended for Void Pointers

Another common C idiom is to cast structure pointers to void pointers. To improve the precision of our analysis, we let void pointers "masquerade" as pointers to any type, and have the void pointer don the appropriate mask when interacting with a different type. More concretely, we treat *void* as a unary type constructor that takes a mapping $F : \mathit{Typ} \rightarrow \mathit{QTyp}$. The *embed′* function creates a fresh mapping for each *void* type. We resolve subtyping constraints between *void*s analogously to the resolution procedure for `struct`s, as shown in Figure 4.8. For a typing constraint between a *void* and another type, τ, we set $F(strip(\tau)) = \tau$. This is the same as treating the *void* as a structure that has a field of every possible type and automatically chooses the field to match the context in which it is used. This approach is not always safe, since programmers can convert from one type to another via a `void` pointer and no constraints between the types will be generated. This is a relatively rare operation in many coding styles, though, and so we believe that this choice balances safety and precision.

4.4 Multiple Files

To scale to large programs consisting of hundreds of source files, CQual implements a modular analysis: it analyzes each source file independently, saves the intermediate results in output files, and then combines those results to achieve a whole-program analysis. To realize any scalability gains, though, the intermediate results must be a small summary of that portion of the overall computation. In CQual, the results of analyzing a source file are

- The qualified types of all the expressions in the program.
- The constraints between the qualifiers.

CQual aggressively converts type constraints into qualifier constraints, so there is no need to store the constraints at the type level.

To compress this information, we first discard the qualified types (but not the qualifiers) on any expression or variable that is not visible outside of the current module. For example, we discard the type information for static functions and globals, local variables (except parameters of externally visible functions), and static local variables. This is safe because the analysis of other files cannot ever produce a type constraint involving a type visible only in the current file. We are now left with the list of externally visible symbols in the current file, the qualified types of those symbols, and a system of constraints, represented as a graph, on a set of qualifiers. After discarding the non-externally visible qualified types, many of the qualifiers in the graph will no longer correspond to any qualified type. We cannot simply discard them, though, because they may still be involved in important constraints on the externally visible qualifiers.

CQual performs a graph compression pass to eliminate as many of these orphaned qualifiers as possible. It then saves the results — the table of externally visible symbols, their qualified types, and the compressed qualifier graph — to an output file. To link two of these files together, it simply loads them into memory and equates the qualified types on all matching symbols in their symbol tables.

To describe the graph compression algorithm, let $G = (V, E)$ be a qualifier constraint graph and $X \subseteq V$ the externally visible qualifiers in G. An X-extension of G is a graph $G' = (V', E')$ such that $V' \cap V \subseteq X$. We can link G and G' into one unified graph, $G \cup G' = (V \cup V', E \cup E')$. We wish to produce a new graph $G_C = (V_C, E_C)$ such that $X \subseteq V_C$, $|V_C| + |E_C|$ is minimized, and such that for every X-extension $G' = (V', E')$ of G and G_C and every pair of vertices $x, y \in V'$, there exists a realizable path $x \rightsquigarrow y \in G \cup G'$ if and only if there exists a realizable path $x \rightsquigarrow y \in G_C \cup G'$.

The notion of "realizable path" in the above description is deliberately undefined, giving rise to a class of problems, depending on the definition of realizable. For our purposes, a path is realizable if and only if it is an NMP-path. Note that the definition does not require that the path $x \rightsquigarrow y \in G \cup G'$ have the same sequence of edge-labels as the path $x \rightsquigarrow y \in G_C \cup G'$, i.e. there is no requirement that there be some correspondence between paths. We are only concerned with preserving the reachability relation.

A naive solution to this problem is to compute and store, for each $x, y \in X$, whether there exists a path $x \rightsquigarrow y \in G$. This has two problems. First, it's not efficient. To see why, suppose $X = \{x_1, \ldots, x_x\}$ and G contains edges $x_i \rightarrow x_1$ and $x_1 \rightarrow x_i$ for all i. This graph contains n nodes and $2n$ edges, but computing its closure would produce $O(n^2)$ edges. Second, it's wrong. The qualifier-constraint graphs produced by CQual have labeled edges, so the reachability relation between nodes is more complex than a simple yes/no relation. For example, the edge $x \stackrel{(}{\rightarrow} y$ can participate in a different set of matched-parenthesis paths than the edge $x \rightarrow y$.

Figure 4.9 presents the compression algorithm currently implemented in CQual. The algorithm considers each node in G and deletes nodes when it can replace all the two-edge paths through that node with a single equivalent edge. The algorithm is greedy in that it will not delete a node if it would have to add more edges than it would delete. The algorithm also computes the strongly connected components of the graph and replaces them with single nodes. For this, we could use an online algorithm for maintaining the strongly-connected components efficiently[64] but, as we shall see, this won't affect the asymptotic running time of the algorithm.

procedure newEdges(E, b)
  $S := \emptyset$
  for each pair of edges $(e_1, e_2) = a \stackrel{\alpha_i}{\rightarrow} b \stackrel{\alpha_j}{\rightarrow} c$
    if (e_1, e_2) does not match the LHS of any rule in Figure 4.10
      return $(0, \emptyset)$
    else
      insert e_3 in to S, where $(e_1, e_2) \Rightarrow e_3$ in Figure 4.10
  return $(1, S \setminus E)$

procedure compact(G, X)
  $(V, E) = G$
  Worklist := $V \setminus X$
  while Worklist is not empty
    pick and remove a vertex b from Worklist
    (d, S) = newEdges(E, b)
    if $d = 1$ and $|S| <$ indegree(b) + outdegree(b) + 1
      add b's neighbors not in X to Worklist
      delete b from G
      $E := E \cup S$
      merge each strongly-connected component of G into a single vertex
      for all merged nodes and neighbors of merged nodes, x, such that $x \notin X$
        add x to Worklist

Figure 4.9. Qualifier Constraint Graph Compaction Algorithm. The qualifier constraint graph is G, and X is the set of externally-visible qualifiers in G. The algorithm deletes vertices from G until it can't find anything else to delete. The strongly-connected component computation ignores labeled edges in G and vertices in X, i.e. x and y are strongly connected if there exists a path of unlabeled edges $x \rightsquigarrow y \rightsquigarrow x$ that does not contain any vertices in X.

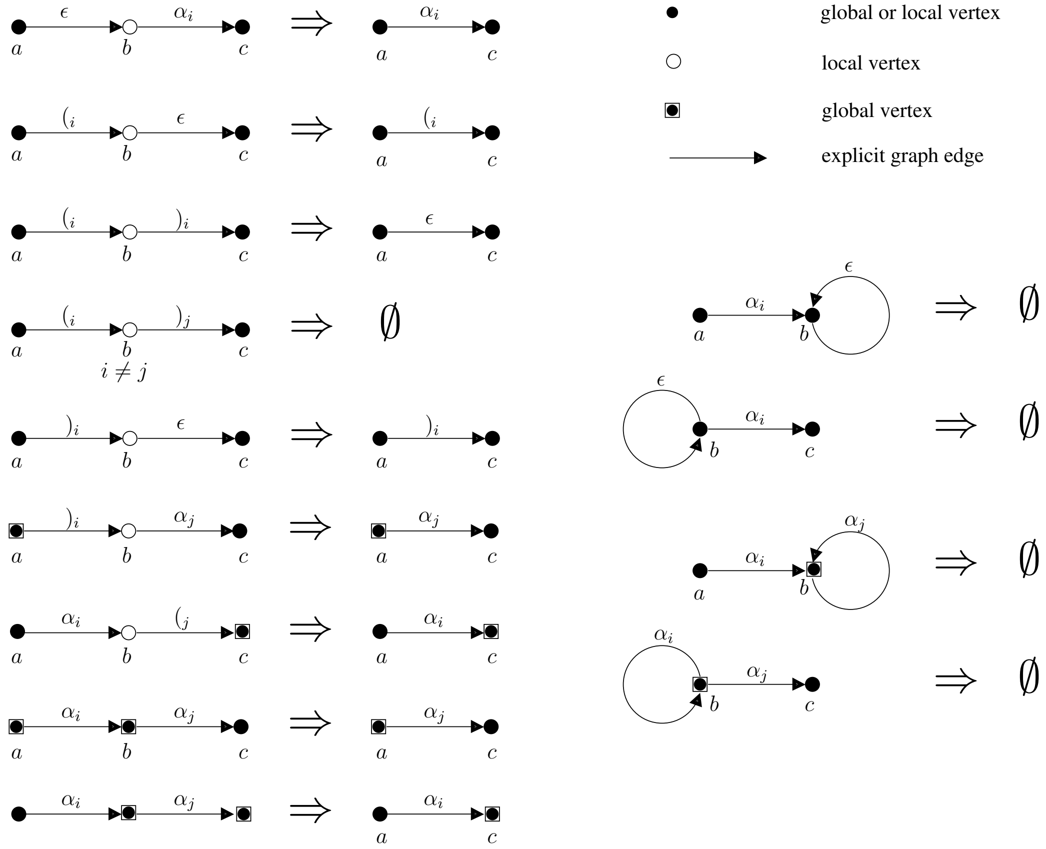


Figure 4.10. Local rules for deleting node b. If every pair of edges incident on b matches one of the templates above, then b may be deleted from the graph and each pair of edges replaced with the indicated edge. The rules in the left column only apply when $a \neq b$ and $b \neq c$. For the rules in the right column, we allow $b = c$ and $a = b$. Note that α_i can match $(_i$, $)_i$ or ϵ.

File	Externs	Original		Compressed		Ratio	Time(s)
		Nodes	Edges	Nodes	Edges		
fs/nfsd/nfscache.c	9592	54664	69450	10402	5738	13%	4.4
mm/mmap.c	10419	70011	93785	11921	11814	14.5%	7.7
net/ipv4/tcp_output.c	14182	112419	154886	16679	19783	13.7%	13.5
arch/i386/kernel/signal.c	8918	53132	68327	9961	6544	13.6%	4.5

Table 4.1. Compaction performance on source files from Linux kernel 2.6.10-rc2.

This algorithm is not guaranteed to find the smallest compressed graph, G_C, for a given G. The algorithm is sound, i.e. preserves the reachability relation of the elements of X, because, for each pair of edges in Figure 4.10, a path can cross that pair of edges if and only if it can cross the edge on the RHS of the rule. In practice, the algorithm does quite well, as shown in Table 4.1. Note that the algorithm eliminates almost all of the non-externally visible nodes, so it cannot do substantially better at eliminating nodes from the graph.

As for running time, the body of the **if** in compact can only execute n times because it deletes a node of G every time it executes. Since the only time that nodes can be inserted into the Worklist is inside this **if**, and since each execution of the body of the **if** may insert up to n nodes into the Worklist, no more than $O(n^2)$ insertions can ever be performed on the Worlist during the entire execution of the algorithm. Thus we may execute the body of the **while** loop $O(n^2)$ times. The running time of newEdges is dominated by the **for** loop, which takes $O(\text{indegree}(b)\text{outdegree}(b))$ time. Since a node's degree changes during the computation, it is difficult to reason about indegree(b)outdegree(b). However, the graph never has more than m edges, so we can use the trivial bound $O(\text{indegree}(b)\text{outdegree}(b)) = O(m^2)$. Thus this part of the while loop takes a total of $O(n^2m^2)$ time to execute.

As for the running time of the body of the **if**, if the algorithm decides to delete b, then b's neighbors can be added to the worklist and b can be deleted from G in $O(\text{indegree}(b) + \text{outdegree}(b)) = O(m)$ time. Since we must have $|S| \leq \text{indegree}(b) + \text{outdegree}(b)$, we can add the edges of S to E in $O(m)$ time, as well. The connected components computation takes $O(n + m)$ time. Thus each execution of the **if** takes $O(n + m)$ time, and so the algorithm spends a total of $O(n(n + m))$ time executing the **if**.

The total running time is thus $O(n^2m^2 + n(n+m)) = O(n^2m^2)$. If $m = O(n)$, as is the case in graphs generated for type-qualifier inference, then the running time is simply $O(n^4)$. This analysis is very coarse, and a more careful treatment may find that the running time is much better. The performance results in Table 4.1 suggest that, when analyzing graphs generated from C programs, the algorithm's time complexity is $O((n+m)^\alpha)$ for $1 \leq \alpha \leq 2$.

The execution of newEdges is a major factor in the running time computed above and many times all that's really needed is $|S|$, not S itself. If we could compute $|S|$ more efficiently, then we could obtain a faster algorithm. This does not look easy. For example, we have to check each edge to determine whether it's already in E. We know of no way to do this other than to compute each edge and perform the check. Alternatively, we can relax the algorithm to merely approximate $|S|$. For example, we could simply upper-bound $|S|$ by indegree(b)outdegree(b). The newEdges function would then take constant time, but we would still have to compute S in the **if**. The **if** gets executed much less often, though, so the new running time would be $O(n^2 + nm^2)$. This approximation is very imprecise, but more-precise efficient approximations exist. For example, we could count the different types of edges incident on a node separately to obtain a better estimate.

In our implementation, the condition $|S| < \text{indegree}(b) + \text{outdegree}(b) + 1$ is relaxed to $|S| < \text{indegree}(b) + \text{outdegree}(b) + 10$ because in practice this flexibility enables the algorithm to find more opportunities for compaction. We derived this bound by trying several values and choosing the bound that worked best in practice. Also, the worklist in the CQual implementation uses FIFO semantics. We have not experimented to determine how this choice affects the performance of the compaction.

4.5 Presenting Qualifier Inference Results

Unlike traditional optimizing compiler technology, in order to be useful the results of the analysis performed by CQual must be presented to the user. We have found that in practice this often-overlooked aspect of program analysis is critically important—a user of

CQual needs to know not only what was inferred but why it was inferred, especially when the analysis detects an error.

After performing type-qualifier inference, CQual presents a list of type-qualifier warnings to the user for evaluation. A naive algorithm for generating warnings would be to solve the generated constraints on-line, emitting a warning whenever newly generated constraints are unsatisfiable. Using this approach, a single program error can result in thousands of warnings. The problem is that the original error can "leak out" to rest of the program and pollute the inferred types because, once a portion of the constraint graph is unsatisfiable, any constraints that interact with that portion of the graph typically also become unsatisfiable. Displaying these extraneous warnings would not only be overwhelming, but would also make it difficult for the programmer to find the root cause of the error since most of the warnings would be only remotely related to the original programming mistake.

CQual reduces the number of error messages by solving the constraints off-line, after all constraints have been generated, and then using heuristics to decide where to generate warning messages. Suppose qualifier variable, v has inconsistent bounds $c_1 \leq v \leq c'_1, c_2 \leq v \leq c'_2, \ldots, c_n \leq v \leq c'_n$. For each inconsistent bound, CQual computes the length of the path needed to explain this bound, i.e. the length, $L(v, i)$, of a realizable path $c_i \overset{\text{CP}}{\leadsto} v \overset{\text{OP}}{\leadsto} c'_i$. Let $R(v) = \min_i L(v, i)$. It then sorts the qualifier variables by R. This tends to put qualifier variables with short, easy-to-explain errors up front. CQual then scans over the variables in sorted order and, for each inconsistent bound $c_i \leq v \leq c'_i$ on a variable v, it checks whether the path $c_i \leadsto v \leadsto c'_i$ is *derivative* or *redundant*, which are described below. If not, then CQual outputs a warning explaining this qualifier error.

For *derivative* errors, which result when one error spreads through a large portion of the constraint graph, CQual uses the following heuristic. Let v be a qualifier variable with error path $c_i \to l_1 \to \cdots \to l_n \to v \to u_1 \to \cdots \to u_m \to c'_i$. Then CQual considers the type error on v to be *derivative* if there exists some $v' = l_j = u_k$. In this case, the error on v is probably just a side-effect of the error on v', and so CQual only reports an error involving v' and not one involving v. Note that the path $c_i \to l_1 \to \cdots \to l_j = v' = u_k \to$

Name	Warnings	
	Unfiltered	Filtered
bftpd-1.0.11	4	1
cfengine-1.5.4	5261	3
muh-2.05d	20	1

Figure 4.11. Tainting Analysis Error Filtering Results

$u_{k+1} \cdots \to u_m \to c_i'$ is not necessarily an NMP-path. The heuristic implemented in CQual only considers v' for which this path is NMP.

CQual also contains a heuristic for eliminating *redundant* error messages. Even if the path $c_i \to l_1 \to \cdots \to l_n \to v \to u_1 \to \cdots \to u_m \to c_i'$ does not cross any node twice, and hence does not satisfy the derivative condition, printing out a warning message for each l_i, u_j, and for v would give the programmer little new information. Thus, after printing a warning for v, CQual flags the other nodes in this path so that it will not print any other error path that crosses one of those nodes. This may suppress other, unrelated warnings, but our experience has found this to be a rare occurance.

Lastly, CQual flags type variables corresponding to intermediate values in the program as *anonymous*, and CQual will initially only report warnings for positions involving non-anonymous variables. In the unlikely event that all type errors involve only anonymous variables, CQual disables this heuristic and reports warnings involving anonymous variables.

Despite their ad-hoc nature, these heuristics have proven extremely effective. Figure 4.11 shows that these error filtering techniques can reduce the number of warnings by over three orders of magnitude. (See Chapter 5 for more information on the tainting analysis used in this benchmark.) Furthermore, in our experience the warnings that are produced have pointed us directly to the source of the error. In most cases, we have found that fixing the errors that CQual displays has resulted in a program that type checks the next time we analyze it with CQual. Thus these heuristics seem to do a very good job of making CQual display exactly one useful error message for each programming error. Overall, we have found these heuristics to be indispensable to making CQual much more usable.

Chapter 5

Evaluation

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 show that the refinements developed to improve the analysis of the Linux kernel were general and would improve results in other applications of type-qualifier inference.

5.1 Linux kernel experiments

To begin, we annotated all the user pointer accessor functions and the dereference operator, as shown in Figure 5.1. 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

```
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 5.1. 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.

took about half a day. Later, we ported these annotations to kernel 2.6.10-rc2. Despite the substantial changes between the 2.4 and 2.6 kernel series, this also took less than half a day.

CQual can be used to perform three types of analyses: file-by-file, whole-program, and modular. A file-by-file analysis looks at each source file in isolation. This type of analysis may miss bugs because it cannot see the entire program, and hence cannot see all the typing constraints implied by the program text. It is very convenient, though, because it closely matches how programmers develop software: by editing, compiling, and debugging one module at a time. A whole-program analysis is sound, but takes more time and memory. Modular analysis, one of the refinements described in Chapter 4, performs a whole-program analysis by analyzing each source file independently and then combining the intermediate results to compute the final answer. The modular analysis should produce the same results as the whole-program analysis but, since it doesn't have to analyze the entire program at once, it can be more efficient. We conducted different experiments using each mode of analysis to understand CQual's bug-finding and scaling properties.

To partially compensate for the unsoundness of a file-by-file analysis, we disabled the subtyping relation *kernel* $<$ *user* in these experiments. Disabling subtyping enables CQual to detect inconsistent use of pointers, which is likely to represent a programming error. For example, 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 therefore fail to typecheck.

The Linux kernel can be configured with a variety of features and drivers. We used two different configurations in our experiments. Since the file-by-file mode has no scaling problems, we configured the kernel to enable as many drivers and features as possible for our file-by-file experiments. We call this the "full" configuration. For the whole-kernel analyses, we used the default configuration as shipped with kernels on kernel.org. We also used the default configuration for the experiments using CQual's modular analysis mode.

5.1.1 Bug-finding

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. For completeness, we also report the number of warnings generated in our whole-kernel analysis of Linux 2.4.23. Since that experiment was primarily to test scalability, though, we did not perform a detailed analysis of the warnings.

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 real bugs. Figure 5.2 shows

Version	Configuration	Mode	Warnings		Exploitable Bugs
			Raw	Unique	
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 5.1. Bug-finding 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.

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 5.3 shows a simple user/kernel bug that an adversary could easily exploit to gain root privileges or crash the system.

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.

Bugs and warnings are not distributed evenly throughout the kernel. Of the eleven bugs

```
 1: int i2cdev_ioctl (struct inode *inode, struct file *file,
 2:                   unsigned int cmd, 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 5.2. 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.

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.

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

```
1: static int
2: w9968cf_do_ioctl(struct w9968cf_device* cam, unsigned cmd, void* arg)
3: {
4: ...
5:      case VIDIOCGFBUF:
6:      {
7:              struct video_buffer* buffer = (struct video_buffer*)arg;
8:
9:              memset(buffer, 0, sizeof(struct video_buffer));
```

Figure 5.3. 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.

stable, mature, slowly changing software systems may have large numbers of undiscovered security holes waiting to be exploited.

These false-positive rates are quite high, which leads to several observations. First, as a user/kernel bug-finding tool, CQual is better suited for code-auditors than code-developers. Even with the large number of false-positives, CQual can reduce the workload of a code-auditor by statically proving that 90% of the source code does not need to be audited for user/kernel pointer bugs. Second, these false-positive rates represent a tremendous improvement over the results obtained using a monomorphic, field-insenstive analysis with no error-filtering heuristics. Early experiments with the Linux kernel produced thousands of warnings in almost every file. Furthermore, CQual is a general tool and is extremely successful in some other application domains. For example, the format-string experiments described later had a 40% false positive rate, and subsequent experiments by Chen has revealed that the false-positive rate can be as low as 13%[15].

The unique warning counts reveal the limitations of our error-clustering heuristics. The raw warning counts in Table 5.1 are after clustering. We performed additional manual clustering to obtain the unique warning counts, demonstrating that there's room for improvement to the clustering algorithms. Also, the drop in unique warning counts in the last experiment is surprising, but not impossible. For example, a whole-program analysis may enable CQual to find a commonly-used function that acts as a source of user pointers and is frequently misused. The function itself may not contain any type-qualifier errors

Version	Configuration	Mode	KLOCs	Time(min)	Max. Memory(GB)
2.4.23	Default	Whole	386	90	10
2.6.10-rc2	Default	Whole	763	failed	failed
2.6.10-rc2	Default	Modular	763	398	4.6

Table 5.2. Scalability results. The experiment with Linux 2.4.23 was conducted on a different machine and with a different methodology than the other experiments, so it is not directly comparable. Line counts are for the original C source files.

so that, when performing a polymorphic analysis, CQual's error-clustering heuristics will ignore the parts of error-paths that lie inside this function. As a result, CQual will report every erroneous call-site to this function as a separate error, and the raw warning count will go up. Furthermore, some errors that CQual found via different paths in the file-by-file analysis may now all be found via this function, causing previously distinct errors to be lumped together in our manual clustering.

We also did a detailed analysis of the false positives generated in this experiment; see Section 5.3.

5.1.2 Scalability of Type-Qualifier Inference.

We performed three experiments to investigate the scalability of the whole-program and modular program analysis modes. During our file-by-file experiments, CQual was almost always able to analyze a single file in under 5 seconds and using less than 100MB of RAM, so we did not bother measuring the scalability of the file-by-file analysis. Our results are presented in Table 5.2.

In our earliest scalability experiment, we performed a whole-program analysis of the default configuration of Linux kernel 2.4.23 on an 800MHz Itanium with 13GB of RAM. We used the CIL program analysis tool[39] to merge all the source files in the default configuration of Linux 2.4.23 into one monolithic source file. This preprocessing step merged duplicate definitions in header files included in many source files into one definition, saving memory during the CQual phase of the analysis. With early versions of CQual, we could not get the experiment to run at all without this step. The merged file contained 466K

non-blank, non-comment lines of code. The time and memory requirements reported in Table 5.2 do not include the CIL portion of the analysis. As the table demonstrates, the analysis requires too much memory to run on standard 32-bit workstations.

In response, we developed the modular analysis algorithms described in Chapter 4. We compared the modular analysis to a whole-program analysis using Linux 2.6.10-rc3 as the test input. We performed the analysis on a 24-way 750MHz UltraSPARC 3 with 72GB of RAM, although CQual is single-threaded, limiting it to one CPU.

The whole-program analysis was performed by modifying the kernel build process to generate preprocessed ".i" files for each source file in the kernel. We then ran CQual on all the ".i" files at once. This analysis failed to complete because the Linux kernel contains fatal type errors where a global symbol is declared as an integer in one source file and as a structure in another file and, in whole-program mode, CQual forces all type declarations to be consistent across all source files. CQual did manage to parse 189 of the 901 source files in the default configuration of the Linux kernel before encountering this error, though, and consumed 13GB of RAM. We project that, if the kernel declarations were fixed to enable the analysis to complete, it would require over 62GB of RAM.

The modular analysis stores the intermediate results of the analysis in ".q" files. We performed the modular analysis on the kernel by creating a .q file for each .i file. The .q file contained the compacted qualifier graph, as described in Chapter 4. We then walked up the kernel source directory tree, creating per-directory .q files by linking together all the .q files in each directory. Our experiments have found that performing graph compaction after linking together already-compacted graphs provides little compression, so CQual performs no further compaction when linking .q files. Finally, we performed type-qualifier inference on the top-level .q file containing the compacted qualifier graph for the entire kernel. In this mode, each file uses its own type-declarations, more closely implementing C semantics and enabling the analysis to process the kernel without encountering the error mentioned above. The analysis required 398 minutes and 4.6GB of RAM to complete, a substantial savings over the estimated 62GB required for the whole-program analysis. The time given for the

modular analysis is the wall-clock time for all these steps, and hence includes a substantial amount of I/O time for reading and writing all the .q files.

It's tempting to compare these results with the numbers from the scalability experiments with the 2.4.23 kernel, but a precise comparison must be made with care. There are three significant differences:

- Linux 2.6.10-rc2 is almost exactly twice as large as Linux 2.4.23,
- We used CIL to merge Linux 2.4.23 into a single source file, eliminating many redundant declarations.
- We tuned CQual's memory usage in many other ways between the two sets of experiments.

If we normalize for the different sizes of the kernels involved, then the modular analysis appears to use about 7.5% as much memory as the whole-program analysis and roughly 25% as much memory as the whole-program analysis using CIL. As for comparing the time required for these analyses, the differing processors make comparison difficult. Also, the 2.4.23 experimental results do not include the time spent running CIL, and the 2.6.10-rc2 times include a lot of I/O. Despite this uncertainty, its safe to say that the modular analysis is slower than the whole-kernel analysis, which makes sense because of the extra compaction and I/O operations.

5.2 Format-String Experiments

The refined type-qualifier analysis yielded dramatic improvements when analyzing the Linux kernel, but we wanted to verify that the refinements were not specific to one application. Earlier work by Shankar, et al[78] had shown that type-qualifier inference can be used to find format-string bugs using a tool call Percent-S. The earlier research had some limitations, though. Many of the programs in their benchmarks generated hundreds or even thousands of warnings – the exact numbers were not reported in their paper. Additional, program-specific annotations reduced the false-positives to the values shown in Table 5.3.

Program	Warnings		Security Holes
	Percent-S	CQual	
muh	12	1	1
cfengine	5	3	1
bftpd	2	1	1

Table 5.3. Format-string bug experimental results. The Percent-S warning counts are as reported by Shankar, et al[78], and are after additional manual annotations have been added to reduce the false-positives. No such annotations were needed with CQual.

We re-ran their experiments, obtaining the results in Table 5.3. As the warning and bug counts show, CQual often had no false-positives and had an overall false-positive rate of 40%. This experiment is too small to support strong conclusions about the false-positive rate, but subsequent work by Chen found a false-positive rate of about 13% in 400 different programs[15]. This makes CQual the best static format-string bug finding tool of which we are aware.

Why are the false-positive rates for format-string bugs and user-kernel bugs so different? This question is difficult to answer because, not only are the problems different, but the coding style of the kernel differs from userspace programs. Modeling user and kernel pointers is more difficult for several reasons. First, a user pointer must point to user data, and our analysis enforces this rule. No similar constraint exists for format strings, so qualifiers can flow around to more places in the user/kernel analysis. This effect is compunded by the fact that the kernel frequently transfers structured data back and forth to user-space, causing all the fields of those structures to be marked as user. Similarly, the kernel stores user pointers in structures, but most user programs do not store format-strings in structures. Thus the user/kernel pointer problem is much more sensitive to the analysis' handling of structures. Finally, almost all calls to `printf`-like functions are trivially safe, since they have a constant format-argument. In contrast, almost every pointer dereference requires a non-trivial analysis to prove safe. This gives many more opportunities for error.

Source	Frequency	Useful	Fix
User flag	50	Maybe	Pass two pointers instead of `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 5.4. 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.

5.3 Linux Kernel 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,[1] 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 preserving the functionality of the code. We kept careful notes on the nature of our changes, and their effect on CQual's output. Table 5.4 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

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

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. Details on each class of false positive is provided in Section 5.4.

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. [2] 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 5.4, 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.

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

5.4 False Positive Details

This section provides detailed descriptions of each class of false-positive.

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 $Q_1^* \;\; \mathit{ref}\,(Q_1 \;\; \texttt{array}\,(Q_1' \;\; \mathit{char}))$. When this gets coerced to $Q_2 \;\; \mathit{ref}\,(Q_2' \;\; \mathit{char})$ 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. Recent versions of CQUAL recognize this C-ism and analyze it correctly.

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: [3]

```
void makemsg(char *buf)
{
  struct msg m;
```

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

```
    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, ...);
}
```

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->body, &m->body, ...);
  km.body = kmalloc(100);
  copy_from_user(km.body, t.body, ...);
}
```

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 4.2, CQual uses unification for fields of structures in order to ensure that memory usage is linear. The downside of 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.

FS Tricks. In a few special circumstances, the kernel can manipulate the memory management hardware to change the semantics of user and kernel pointers. For historical reasons, this is performed with functions `get_fs` and `set_fs`. These functions are used extremely rarely, so we believe their use can simply be verified by hand.

Chapter 6

Related Work

There are three main threads of related work: prior systems that use type qualifiers, other systems similar to CQual that find and prevent general errors in programs, and particular systems for finding security bugs. We also touch on some related algorithmic work.

6.1 Flow-Insensitive Type Qualifiers

Specific examples of flow-insensitive type qualifiers have been proposed to solve a number of problems. For example, ANSI C contains the type qualifier *const* [5]. The *const* qualifier was added to the standard in 1989 [4], inspired by C++'s *const*, which Stroustrup "invented" [82]. Binding-time analysis [25] can be viewed as associating one of two qualifiers with expressions, either *static* for expressions that may be computed at compile time or *dynamic* for expressions not computed until run-time. The Titanium programming language [93] uses qualifiers *local* and *global* to distinguish data located on the current processor from data that may be located at a remote node [52]. Solberg [81] gives a framework for understanding a particular family of related analyses as type annotation (qualifier) systems. Pratikakis et al. [71] give a system for inferring qualifiers for transparent Java futures. In contrast to these

systems, our approach is an extensible, general framework for adding new, user-specified qualifiers that are expressible in our system.

Chin et al. [16] develop a semantic type qualifier system that allows qualifiers to be associated with language operations. Their system uses theorem proving to check that qualifier specifications are correct, and they can verify richer properties than subtyping constraints allow (for example, *pos* and *neg* qualifiers for integers). Their system does not currently include inference, unlike CQual.

Several related techniques have been proposed for using qualifier-like annotations to address security issues. A major topic of recent interest is *secure information flow* [2, 23, 80, 85], which associates *high* and *low* security levels with expressions and tries to prevent high-security data from "leaking" to low-security outputs. Other examples of security-related annotation systems are lambda calculus with trust annotations [63] and Java security checking [79]. These systems include checks for implicit flows from conditional guards to the body of the conditional. Section 6.3 below contains a discussion of why CQual does not include this feature.

Type qualifiers, like any type system, can be seen as a form of abstract interpretation [17]. Flow-insensitive type qualifiers can be viewed as a label flow system [58] in which we place constraints on where labels may flow. Type qualifiers can also be viewed as refinement types [38], which have the same basic property: refinement types do not change the underlying type structure. The key difference between qualifiers and Freeman and Pfenning's refinement types is that the latter is based on the theory of intersection types, which is significantly more complex than atomic subtyping. Mandelbaum et al. [53] have developed a type system that incorporates a logic of type refinements to allow reasoning about state, corresponding to flow-sensitivity. There is currently a limited implementation of their type system, but as of yet its scalability and effectiveness in practice are unknown.

6.2 Error Detection and Prevention Systems

Many systems have recently been proposed that allow programmers to check more properties of their programs. Vault [22, 30] and Cyclone [41, 40] are two safe variants of C that allow a programmer to enforce conditions on how resources are used in programs. These systems allow flow-sensitive tracking of resources, which is not modeled in our flow-insensitive framework (but see Foster et al. [36]). However, Vault and Cyclone also require per-function annotations, whereas CQual only requires a few annotations on the entire program and performs whole-program inference. Additionally, to use Vault and Cyclone the programmer must rewrite their program into the new language (which may vary from trivial to difficult), whereas CQual is designed to work with a legacy language, namely C. Because CQual operates on C, it cannot be fully sound, unlike these new languages.

Several systems based on dataflow analysis have been proposed to statically check properties of source code. These systems are flow-sensitive, in contrast to the flow-insensitive qualifiers described in this paper. One such system is Evans's Splint [29], which introduces a number of additional qualifier-like annotations to C as an aid to debugging memory usage errors. Evans found Splint to be valuable in practice [29], and Splint has also been used to check for buffer overruns [50]. The main different between Splint and CQual is annotations. Splint's analysis is intraprocedural, relying on programmers-supplied annotations at function calls, whereas CQual can perform whole-program inference.

Another such system is meta-level compilation [27, 42], in which the programmer specifies a flow-sensitive property as a finite state automaton. Meta-level compilation includes an interprocedural dataflow component [42] but does not model general aliasing, unlike CQual. Meta-level compilation has been used to find many different kinds of bugs in programs, including tainting bugs [92]. The key difference between CQual's approach and meta-level compilation is soundness. While CQual is not fully sound (e.g., due to arbitrary pointer arithmetic in C), CQual strives for soundness up to the limitations of C. In contrast, the goal of meta-level compilation is bug finding, and features like aliasing are ignored in order to limit false positives (which there are more of in CQual).

A third dataflow-based system is ESP [19], an error detection tool based on sound dataflow analysis. ESP incorporates a conservative alias analysis to model pointers, and uses path-sensitive symbolic execution to model predicates. ESP has been used to check the correctness of C stream library usage in `gcc` [19]. ESP is designed to soundly detect all errors with a minimum of false positives; as such, the algorithms it uses for tracking state are quite sophisticated, and it is not easy for a programmer to predict in advance whether their program will check successfully. In contrast, CQual produces more warnings, but gives the programmer a relatively simple, predictable, type-based discipline to avoid errors.

The Extended Static Checking (ESC) system [24, 51, 33] is a theorem-proving based tool for finding errors in programs. Programmers add extensive annotations, including preconditions, postconditions, and loop invariants to their program, and ESC uses sophisticated theorem proving technology to verify the annotations. ESC includes a rich annotation language; the Houdini assistant [32] can be used to reduce the burden of adding annotations. ESC provides significantly more sophisticated checking than CQual, but at the cost of scalability, both in terms of annotations and efficiency.

SLAM [7, 8] and BLAST [45] verify software using model checking techniques. Both tools can track program state very precisely and are by their nature flow- and path-sensitive. They use predicate abstraction followed by successive refinement to make analysis more tractable, and they have been used to check properties of device drivers. SLAM includes techniques for producing small counterexamples to explain error messages [6]. While the scalability of these tools is promising, the systems' worst-case complexity is much higher than CQual.

A number of techniques that are less easy to categorize have also been proposed. The AST toolkit provides a framework for posing user-specified queries on abstract syntax trees annotated with type information. The AST toolkit has been successfully used to uncover many bugs [88]. The PREfix tool [11], based on symbolic execution, is also highly effective at finding bugs in practice [65]. Both of these tools are unsound, and are designed to catch bugs rather than show the absence of errors.

A number of systems have been proposed to check that implementations of data structures are correct. Graph types [48, 57] allow a programmer to specify the shape of a data structure and then check, with the addition of pre- and postconditions and loop invariants, that the shape is preserved by data structure operations. Shape analysis with three-valued logic [77] can also model data structure operations very precisely. Both of these techniques are designed to run on small inputs, and neither in its current form scales to large programs.

6.3 Static Analysis and Security

The format-string taint analysis approach is conceptually similar to Perl's taint mode [86], but with a key difference: unlike Perl, which tracks tainting dynamically, CQUAL checks tainting statically without ever running the program. Moreover, CQUAL's results are conservative over all possible runs of the program. This gives us a major advantage over dynamic approaches for finding security flaws. Often security bugs are in the least-tested portions of the code, and a malicious adversary is actively looking for just such code to exploit. Using static analysis, we conceptually analyze all possible runs of the program, providing complete code coverage. Recent work by Sekar, et al, has added dynamic taint-tracking to C programs and, by extension, to interpreted languages with an interpreter written in C[87].

Several lexical techniques have been proposed for finding security vulnerabilities. Pscan [21] searches the source code for calls to `printf`-like functions with a non-constant format string. Thus pscan cannot distinguish between safe calls when the format string is variable and unsafe calls. Lexical techniques have also been proposed to find other security vulnerabilities [9, 84]. RATS[47] and ITS4[84] flag a large number of potentially dangerous operations using lexical techniques, although they suffer from high false-positive rates. The main advantage of lexical techniques is that they are extremely fast and can analyze non-preprocessed source files. However, because lexical tools have no knowledge of language semantics there are many errors they cannot find, such as those involving aliasing or function calls.

Another approach to eliminating security vulnerabilities is to add dynamic checks. The `libformat` library intercepts calls to `printf`-like functions and aborts when a format string contains `%n` and is in a writable address space [76]. A disadvantage to `libformat` is that, to be effective, it must be kept in synchronization with the C libraries. Another dynamic system is FormatGuard, which injects code to dynamically reject bad calls to `printf`-like functions [18]. The main disadvantage of FormatGuard is that programs must be recompiled with FormatGuard to benefit.

CCured uses static and dynamic analysis to prevent buffer-overflows[60]. CCured uses a type-qualifier-based analysis to statically prove most memory references in a program are safe and inserts run-time checks on all the other references. The CCured analysis is sound (or, at least, as sound as it can be on C programs), giving strong guarantees of security, but it has some costs. Some programs must be extensively modified before CCured can process them, linking CCured-compiled code with non-CCured code is difficult, and the run-time overhead is high — roughly 50%.

MOPS is a special-purpose model-checker designed for catching control-flow-based security bugs[13]. It differs substantially from CQual in that it is primarily concerned with control-flow, whereas type-qualifier inference considers the dataflow properties of a program. Interestingly, the model-checking problem solved by MOPS can be reduced to a CFL-reachability problem similar to that described in Chapter 4.

6.4 Algorithmics

Horwitz, Reps, and Sagiv first made the connection between data-flow problems and CFL reachability and presented linear-time algorithms for solving matched-parenthesis reachilibilty on graphs generated from procedural programs[75]. Subsequent research by Reps and Kodumal explored the connection between CFL-reachability and certain classes of set constraints[54, 49]. As mentioned before, Das, et al. described a globals optimization that improved the running time in practice but did not provide an asymptotic speed-up[20].

The graph compaction problem is new, but researchers have investigated several related

problems on unlabeled graphs, where reachability is just normal graph-reachability. The transitive reduction problem seeks to find the smallest graph with the same transitive closure as the original graph. Note that, in this variant, the vertex set is fixed. Transitive reduction has the same time complexity as transitive closure[3]. Feder and Motwani gave an algorithm, based on finding large bipartite cliques, for reachability-preserving graph compression[31]. Their compression algorithm may add nodes but, unlike our graph compaction problem, their compression is lossless, i.e. the original graph can be reconstructed from its compressed version. Naor, among others, has looked into the problem of compressing graphs in the traditional sense, i.e. finding a representation with the fewest number of bits[59]. We were more concerned with the in-memory size of the graph, so this notion of graph-compression is not directly applicable to our problem.

Chapter 7

Conclusion

This dissertation describes several refinements to type-qualifier inference and demonstrates that it can form the basis for effective code auditing tools for finding real security bugs.

First we described an algorithm for efficiently computing matched-parenthesis reachability queries in graphs containing global nodes. These graphs can arise from a variety of program-anlysis problems, including type-qualifier inference and alias analysis. Ours is the first algorithm for solving this problem in linear time. We also described several techniques for improving the analysis of C programs without sacrificing efficiency. We then posed the lossy graph compression problem and described a greedy algorithm for computing good solutions efficiently, enabling the modular analysis of large programs. Finally, we presented heuristics for selecting good warnings to display to the programmer to help him fix any type-qualifier errors discovered by our analysis.

We evaluated these refinements with several experiments on the Linux kernel and other UNIX programs. We found dozens of bugs in the kernel and demonstrated a very low false-positive rate for detecting format-string bugs, making CQual the best static format-string bug detector to date. We also investigated the scalability of our algorithms by performing a modular analysis of the Linux kernel, and found that reasonable developer workstations can perform this analysis.

Bibliography

[1] Common vulnerabilities and exposures list. `http://cve.mitre.org/`.

[2] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. A Core Calculus of Dependency. In POPL'99 [70], pages 147–160.

[3] M. R. Aho, A. Garey and J. D Ullman. The transitive reduction of a directed graph. *SIAM Journal of Computing*, 1:131–137, 1972.

[4] ANSI. *Rationale for American National Standard for Information Systems—Programming Language—C*, 1989. Associated with ANSI standard X3.159-1989.

[5] ANSI. *Programming languages – C*, 1999. ISO/IEC 9899:1999.

[6] Thomas Ball, Mayur Naik, and Sriram K. Rajamani. From Symptom to Cause: Localizing Errors in Counterexample Traces. In *Proceedings of the 30th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 97–105, New Orleans, Louisiana, USA, January 2003.

[7] Thomas Ball and Sriram K. Rajamani. Automatically Validating Temporal Safety Properties of Interfaces. In *The 8th International SPIN Workshop on Model Checking of Software*, number 2057 in Lecture Notes in Computer Science, pages 103–122, May 2001.

[8] Thomas Ball and Sriram K. Rajamani. The SLAM Project: Debugging System Software via Static Analysis. In POPL'02 [69], pages 1–3.

[9] Matt Bishop and Michael Dilger. Checking for Race Conditions in File Accesses. *Computing Systems*, 2(2):131–152, 1996.

[10] Nikita Borisov, Rob Johnson, Naveen Sastry, and David Wagner. Fixing races for fun and profit: How to abuse atime. In *Proceedings of the 14th USENIX Security Symposium*, pages 303–314, Baltimore, MD, USA, August 2005.

[11] William R. Bush, Jonathan D. Pincus, and David J. Sielaff. A static analyzer for finding dynamic programming errors. *Software—Practice and Experience*, 30(7):775–802, June 2000.

[12] Satish Chandra and Thomas W. Reps. Physical Type Checking for C. In *Proceedings of the ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 66–75, Toulouse, France, September 1999.

[13] 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.

[14] Hao Chen, David Wagner, and Drew Dean. Setuid demystified. In *Proceedings of the 11th USENIX Security Symposium*, pages 171–190, San Francisco, CA, USA, August 2002.

[15] Karl Chen. Personal communication. June 2006.

[16] Brian Chin, Shane Markstrum, and Todd Millstein. Semantic Type Qualifiers. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Chicago, Illinois, June 2005.

[17] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[18] Crispin Cowan, Matt Barringer, Steve Beattie, and Greg Kroah-Hartman. FormatGuard: Automatic Protection From printf Format String Vulnerabilities. In USENIXSEC'01 [83].

[19] Manuvir Das, Sorin Lerner, and Mark Seigle. ESP: Path-Sensitive Program Verification in Polynomial Time. In PLDI'02 [67], pages 57–68.

[20] Manuvir Das, Ben Liblit, Manuel Fähndrich, and Jakob Rehof. Estimating the impact of scalable pointer analysis on optimization. In Patrick Cousot, editor, *Static Analysis, Eighth International Symposium*, volume 2126 of *Lecture Notes in Computer Science*, Paris, France, July 2001. Springer-Verlag.

[21] Alan DeKok. PScan: A limited problem scanner for C source files. `http://www.striker.ottawa.on.ca/~aland/pscan`.

[22] Robert DeLine and Manuel Fähndrich. Enforcing High-Level Protocols in Low-Level Software. In PLDI'01 [66], pages 59–69.

[23] Dorothy E. Denning. A Lattice Model of Secure Information Flow. *Communications of the ACM*, 19(5):236–243, May 1976.

[24] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended Static Checking. Technical Report 159, Compaq Systems Research Center, December 1998.

[25] Dirk Dussart, Fritz Henglein, and Christian Mossin. Polymorphic Recursion and Subtype Qualifications: Polymorphic Binding-Time Analysis in Polynomial Time. In Alan Mycroft, editor, *Static Analysis, Second International Symposium*, number 983 in Lecture Notes in Computer Science, pages 118–135, Glasgow, Scotland, September 1995. Springer-Verlag.

[26] Jonathan Eifrig, Scott Smith, and Valery Trifonov. Type Inference for Recursively Constrained Types and its Application to OOP. In *Mathematical Foundations of Programming Semantics, Eleventh Annual Conference*, volume 1 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1995.

[27] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions. In *Fourth symposium on Operating System Design and Implementation*, San Diego, California, October 2000.

[28] Noam Eppel. Security absurdity: The complete, unquestionable, and total failure of information security. `http://www.securityabsurdity.com/failure.php`, 2006.

[29] David Evans. Static Detection of Dynamic Memory Errors. In *Proceedings of the 1996 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 44–53, Philadelphia, Pennsylvania, May 1996.

[30] Manuel Fähndrich and Robert DeLine. Adoption and Focus: Practical Linear Types for Imperative Programming. In PLDI'02 [67], pages 13–24.

[31] Tomas Feder and Rajeev Motwani. Clique partitions, graph compression and speeding-up algorithms. In *STOC '91: Proceedings of the twenty-third annual ACM symposium on Theory of computing*, pages 123–133, New York, NY, USA, 1991. ACM Press.

[32] Cormac Flanagan and K. Rustan M. Leino. Houdini, an Annotation Assitant for ESC/Java. In J. N. Oliverira and Pamela Zave, editors, *FME 2001: Formal Methods for Increasing Software Productivity, International Symposium of Formal Methods*, number 2021 in Lecture Notes in Computer Science, pages 500–517, Berlin, Germany, March 2001. Springer-Verlag.

[33] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended Static Checking for Java. In PLDI'02 [67], pages 234–245.

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

[35] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. A Theory of Type Qualifiers. In PLDI'99 [68], pages 192–203.

[36] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-Sensitive Type Qualifiers. In PLDI'02 [67], pages 1–12.

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

[38] Tim Freeman and Frank Pfenning. Refinement Types for ML. In *Proceedings of the 1991 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 268–277, Toronto, Ontario, Canada, June 1991.

[39] Shree P. Rahul Westley Weimer George C. Necula, Scott McPeak. Cil: Intermediate language and tools for analysis and transformation of c programs. In *11th International Conference on Compiler Construction*. Springer Berlin / Heidelberg, April 2002.

[40] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-Based Memory Management in Cyclone. In PLDI'02 [67], pages 282–293.

[41] Dan Grossman, Greg Morrisett, Yanling Wang, Trevor Jim, Michael Hicks, and James Cheney. Cyclone User's Manual. Technical Report 2001-1855, Department of Computer Science, Cornell University, November 2001.

[42] Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson Engler. A System and Language for Building System-Specific, Static Analyses. In PLDI'02 [67], pages 69–82.

[43] Nevin Heintze and Olivier Tardieu. Ultra-fast Aliasing Analysis using CLA: A Million Lines of C Code in a Second. In PLDI'01 [66], pages 254–263.

[44] Fritz Henglein. Type Inference with Polymorphic Recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, April 1993.

[45] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy Abstraction. In POPL'02 [69], pages 58–70.

[46] Susan Horwitz, Thomas Reps, and Mooly Sagiv. Demand Interprocedural Dataflow Analysis. In *Third Symposium on the Foundations of Software Engineering*, pages 104–115, Wasington, DC, October 1995.

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

[48] Nils Klarlund and Michael I. Schwartzback. Graph Types. In *Proceedings of the 20th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 196–205, Charleston, South Carolina, January 1993.

[49] John Kodumal and Alex Aiken. The set constraint/CFL reachability connection in practice. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 207–218. ACM Press, 2004.

[50] David Larochelle and David Evans. Statically Detecting Likely Buffer Overflow Vulnerabilities. In USENIXSEC'01 [83].

[51] K. Rustan M. Leino and Greg Nelson. An Extended Static Checker for Modula-3. In Kai Koskimies, editor, *Compiler Construction, 7th International Conference*, volume 1383 of *Lecture Notes in Computer Science*, pages 302–305, Lisbon, Portugal, April 1998. Springer-Verlag.

[52] Ben Liblit and Alexander Aiken. Type Systems for Distributed Data Structures. In *Proceedings of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 199–213, Boston, Massachusetts, January 2000.

[53] Yitzhak Mandelbaum, David Walker, and Robert Harper. An Effective Theory of Type Refinements. In *Proceedings of the eighth ACM SIGPLAN International Conference on Functional Programming*, pages 213–225, Uppsala, Sweden, August 2003.

[54] D. Melski and T. Reps. Interconvertibility of set constraints and context-free language reachability. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 74–89, Amsterdam, The Netherlands, June 1997.

[55] Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[56] John C. Mitchell. Type inference with simple subtypes. *Journal of Functional Programming*, 1(3):245–285, July 1991.

[57] Anders Møller and Michael I. Schwartzbach. The Pointer Assertion Logic Engine. In PLDI'01 [66], pages 221–231.

[58] Christian Mossin. *Flow Analysis of Typed Higher-Order Programs.* PhD thesis, DIKU, Department of Computer Science, University of Copenhagen, 1996.

[59] Naor. Succinct representation of general unlabeled graphs. *DAMATH: Discrete Applied Mathematics and Combinatorial Operations Research and Computer Science*, 28, 1990.

[60] George Necula, Scott McPeak, and Westley Weimer. CCured: Type-Safe Retrofitting of Legacy Code. In POPL'02 [69], pages 128–139.

[61] Tim Newsham. Format string attacks. `http://community.corest.com/~juliano/tn-usfs.txt`, September 2000.

[62] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type Inference with Constrained Types. In Benjamin Pierce, editor, *Proceedings of the 4th International Workshop on Foundations of Object-Oriented Languages*, January 1997.

[63] Peter Ørbæk and Jens Palsberg. Trust in the λ-calculus. *Journal of Functional Programming*, 3(2):75–85, 1997.

[64] David J. Pearce and Paul H. J. Kelly. Online algorithms for topological order and strongly connected components. Technical report, Imperial College of Science, Technology, and Medicine, Department of Computing, 180 Queen's Gate, London SW7 2BZ, UK, Sep 2003.

[65] Jonathan D. Pincus. Personal communication, 2002.

[66] *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Snowbird, Utah, June 2001.

[67] *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Berlin, Germany, June 2002.

[68] *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Atlanta, Georgia, May 1999.

[69] *Proceedings of the 29th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Portland, Oregon, January 2002.

[70] *Proceedings of the 26th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Antonio, Texas, January 1999.

[71] Polyvios Pratikakis, Jaime Spacco, and Michael Hicks. Transparent Proxies for Java Futures. In *Proceedings of the nineteenth annual conference on Object-oriented programming systems, languages, and applications*, pages 206–223, October 2004.

[72] Vaughan Pratt and Jerzy Tiuryn. Satisfiability of Inequalities in a Poset. *Fundamenta Informaticae*, 28(1-2):165–182, 1996.

[73] Jakob Rehof and Manuel Fähndrich. Type-Based Flow Analysis: From Polymorphic Subtyping to CFL-Reachability. In *Proceedings of the 28th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 54–66, London, United Kingdom, January 2001.

[74] Jakob Rehof and Torben Æ. Mogensen. Tractable Constraints in Finite Semilattices. In Radhia Cousot and David A. Schmidt, editors, *Static Analysis, Third International Symposium*, volume 1145 of *Lecture Notes in Computer Science*, pages 285–300, Aachen, Germany, September 1996. Springer-Verlag.

[75] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Proceedings of the 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 49–61, San Francisco, California, January 1995.

[76] Tim J. Robbins. libformat–protection against format string attacks, January 2001. `http://www.wiretapped.net/~fyre/software/libformat.html`.

[77] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric Shape Analysis via 3-Valued Logic. In POPL'99 [70], pages 105–118.

[78] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting Format String Vulnerabilities with Type Qualifiers. In USENIXSEC'01 [83].

[79] Christian Skalka and Scott Smith. Static Enforcement of Security with Types. In *Proceedings of the fifth ACM SIGPLAN International Conference on Functional Programming*, pages 34–45, Montreal, Canada, September 2000.

[80] Geoffrey Smith and Dennis Volpano. Secure Information Flow in a Multi-Threaded Imperative Language. In *Proceedings of the 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 355–364, San Diego, California, January 1998.

[81] Kirsten Lackner Solberg. *Annotated Type Systems for Program Analysis*. PhD thesis, Aarhus University, Denmark, Computer Science Department, November 1995.

[82] Bjarne Stroustrup. C++ Style and Technique FAQ, January 2005. `http://www.research.att.com/~bs/bs_faq2.html#constplacement`.

[83] *Proceedings of the 10th Usenix Security Symposium*, Washington, D.C., August 2001.

[84] 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`.

[85] Dennis Volpano and Geoffrey Smith. A Type-Based Approach to Program Security. In Michel Bidoit and Max Dauchet, editors, *Theory and Practice of Software Development, 7th International Joint Conference*, volume 1214 of *Lecture Notes in Computer Science*, pages 607–621, Lille, France, April 1997. Springer-Verlag.

[86] Larry Wall, Tom Christiansen, and Jon Orwant. *Programming Perl*. O'Reilly & Associates, 3rd edition edition, July 2000.

[87] Sandeep Bhatkar Wei Xu and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *Proceedings of the 15th Usenix Security Symposium*, Vancouver, BC, August 2006.

[88] Daniel Weise, 2001. Personal communication.

[89] David A. Wheeler. Secure Programming for Linux and Unix HOWTO. March 2003.

[90] Andrew K. Wright. Simple Imperative Polymorphism. In *Lisp and Symbolic Computation 8*, volume 4, pages 343–356, 1995.

[91] Andrew K. Wright and Matthias Felleisen. A Syntactic Approach to Type Soundness. *Information and Computation*, 115(1):38–94, 1994.

[92] 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 Communications Security*, pages 321–334, Washington, D.C., USA, 2003.

[93] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A High-Performance Java Dialect. In *ACM 1998 Workshop on Java for High-Performance Network Computing*, February 1998.

[94] Suan Hsi Yong, Susan Horwitz, and Thomas Reps. Pointer Analysis for Programs with Structures and Casting. In PLDI'99 [68], pages 91–103.