Quantitative Information-Flow Tracking for Real Systems

by

Stephen Andrew McCamant

B.A., Computer Science
University of California, Berkeley, 2002
S.M., Electrical Engineering and Computer Science
Massachusetts Institute of Technology, 2004

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2008

Author . . . .
Department of Electrical Engineering and Computer Science
May, 2008

Certified by . . . .
Michael D. Ernst
Associate Professor
Thesis Supervisor

Accepted by . . . .
Terry P. Orlando
Chairman, Department Committee on Graduate Students

Quantitative Information-Flow Tracking for Real Systems

by

Stephen Andrew McCamant

Submitted to the Department of Electrical Engineering and Computer Science
on May 23, 2008, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science

Abstract

An information-flow security policy constrains a computer system's end-to-end use of information, even as it is transformed in computation. For instance, a policy would not just restrict what secret data could be revealed directly, but restrict any output that might allow inferences about the secret. Expressing such a policy quantitatively, in terms of a specific number of bits of information, is often an effective program-independent way of distinguishing what scenarios should be allowed and disallowed.

This thesis describes a family of new techniques for measuring how much information about a program's secret inputs is revealed by its public outputs on a particular execution, in order to check a quantitative policy on realistic systems. Our approach builds on dynamic tainting, tracking at runtime which bits might contain secret information, and also uses static control-flow regions to soundly account for implicit flows via branches and pointer operations. We introduce a new graph model that bounds information flow by the maximum flow between inputs and outputs in a flow network representation of an execution. The flow bounds obtained with maximum flow are much more precise than those based on tainting alone (which is equivalent to graph reachability). The bounds are a conservative estimate of channel capacity: the amount of information that could be transmitted by an adversary making an arbitrary choice of secret inputs.

We describe an implementation named Flowcheck, built using the Valgrind framework for x86/Linux binaries, and use it to perform case studies on six real C, C++, and Objective C programs, three of which have more than 250,000 lines of code. We used the tool to check the confidentiality of a different kind of information appropriate to each program. Its results either verified that the information was appropriately kept secret on the examined executions, or revealed unacceptable leaks, in one case due to a previously unknown bug.

Thesis Supervisor: Michael D. Ernst
Title: Associate Professor

Acknowledgments

Portions of this thesis draw on work previously published in the proceedings of ACM SIGPLAN 2007 Workshop on Programming Languages and Analysis for Security (PLAS) [ME07b], and the proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI) [ME08], and described in a series of MIT CSAIL technical reports [ME06, ME07a].

Many colleagues encouraged and supported the research described in this thesis. Foremost of course is Michael Ernst, my thesis supervisor and research advisor over all six years of graduate school. He contributed valuable insights throughout the research described here, and was a tireless source of suggestions on improving its presentation. I also received helpful suggestions from the other members of my thesis committee, Barbara Liskov and Robert Morris, as well as from Frans Kaashoek, who provided quite a bit of useful advice on early parts of this research, before stepping down from the committee for scheduling reasons. The Valgrind-based implementation described here drew inspiration from my collaboration with Philip Guo on Kvasir and DynComp, and I am grateful to the original Valgrind developers, especially Nicholas Nethercote and Julian Seward, for making their work available, providing documentation, and answering questions. This work also benefited from suggestions and insightful questions from the MIT Program Analysis and Parallel & Distributed Operating Systems research groups, talk audiences at Harvard, IBM, MIT Lincoln Labs, and PLAS 2007; and from more detailed comments from Andrew Myers, and anonymous reviewers for the IEEE Security and Privacy symposium, USENIX Security, ACM CCS, PLAS, and PLDI.

This research was supported in part by the Defense Advanced Research Projects Agency (under contracts FA8750-06-2-0189 and HR0011-06-1-0017), by the National Science Foundation (under grant CCR-0133580), and by a National Defense Science and Engineering Graduate Fellowship.

Biographical note

Stephen McCamant was born in Chicago, Illinois in 1980. He attended the University of California, Berkeley, from 1998 to 2002, earning a Bachelor of Arts in Computer Science. A student at MIT since 2002, he earned a Master of Science in Electrical Engineering and Computer Science in 2004, with thesis research on "Predicting Problems Caused by Component Upgrades."

Contents

List of Figures

Chapter 1

Introduction

Computers should be able to keep secrets. Much of the information that computers process is private, sensitive, or confidential, so users would like to know that it is not revealed to others who should not have it. However, it is difficult for computer systems to provide limits on where information propagates as it is processed, because digital information can be very easily copied and need bear no trace of its origin, and because it is difficult to know whether software obeys its (often missing) specification. *Information-flow security* refers to enforcing limits on the use of information that cover not just access to data in the first instance, but how the information contained in the data might be revealed in subsequent computations.

Our goal in this thesis to automatically check whether real software meets an information-flow policy, by measuring the amount of information disclosed on a particular execution. To that end, we present a family of new techniques, which for the first time make such measurements practical on large, preexisting software systems, without a need for excessive developer assistance. A developer can use a tool like ours to measure the amount of secret information that a program actually reveals. Then, the developer can decide whether that amount is acceptable. If the program is revealing too much information, the tool's results indicate the code location where the excessive flow occurs.

Because our tool's results apply only to the executions it observes, it is best used as part of the testing and auditing process; but users would like be sure that a security policy is obeyed on every production execution. The dynamic approach of our tool can be used to obtain this kind of runtime checking, preventing a program from revealing too much information, though at a performance cost (for instance, approximately a factor of two for a technique described in Section 4.2). Static checking, which would bound the amount of information a program might reveal on any possible execution, is not yet practical for full-scale applications, but our techniques could be used to help ease the developer burden of static checking, and they point to a future direction for static checking research (discussed in Section 8.2.2).

As one example of how a tool like ours could be used, suppose that you are the developer of a network-based strategy game, and you wish to evaluate how much the network protocol implementation reveals about strategically important game information. Your initial expectation is that the protocol should reveal 1 bit of secret information per round of the game. You annotate the program to mark the strategic information as secret on input, and run the program under our tool. The tool reports a code location that reveals 8 bits of information per round. On examining that code, you discover that the implementation has a bug that is revealing extra information to an opponent. After fixing the bug, you rerun our tool; it now measures the program as revealing 2 bits of information per round, consisting of the 1 bit you had originally expected, plus a second one that you also realize is correct. You then use a streamlined version of our technique to automatically check that future executions reveal no more than 2 bits per round. In fact, we examine just such a network game in the case study of Section 7.2, and discover a previously unknown bug.

In the rest of this introduction, we first describe our goals for such a tool in more detail (Section 1.1), then outline how our approach achieves them (Section 1.2).

1.1 Requirements for an information-flow tool

To be usable for assessing the information-flow security of a program, a tool must satisfy three general domain requirements. First, it must be able to enforce a policy that allows certain flows. Our system supports this by quantifying flows and allowing

	Partial flows	Soundness	Scalability
Domain goal	Allow some information to be revealed, but not too much	Must not miss flows in any program construct	Applicable to realistic programs
Technical requirement	Measure quantitative confidentiality policy	Measurement is an upper bound, accounting for implicit flows	Works on large pre-existing C and C++ programs

Figure 1-1: Summary of the requirements for an information-flow measurement tool, as detailed in Section 1.1.

small ones. Second, it must be sound in the sense of never missing a flow that actually occurs in a program, even if the information is transformed via computations. Third, it must be practical to apply to real software, working on large code bases in industrial languages without running for too long or requiring too much help from developers. These requirements are summarized in Figure 1-1. The rest of this section explains in more detail the kinds of information-flow policies (Section 1.1.1), what it means for a policy to be enforced soundly (Section 1.1.2), and the requirements for a tool to scale to practically interesting programs (Section 1.1.3).

1.1.1 Information flow policies

Information-flow policies are of two broad varieties. A *confidentiality* policy requires that a program that is entrusted with secrets should not "leak" those secrets into public outputs. An *integrity* policy requires that a program that operates on information that might be untrustworthy does not let that information affect results that must be correct. There is a duality between these policy varieties, and both can be enforced with many of the same techniques. However, we concentrate on confidentiality problems, because they are addressed less well by existing systems.

An ideal confidentiality property called *non-interference* is a guarantee that no information about secret inputs can be obtained by observing a program's public outputs, for any choice of its public inputs. Unfortunately, such absolute prohibitions on information flow are rarely satisfied by real programs. (If a program really does satisfy non-interference, so that there is no connection between its secret and public computations, it would be better to remove the secret computation entirely.)

Rather, the key challenge for information-flow security is to distinguish acceptable from unacceptable flows.

Systems often deal with private or sensitive information by revealing only a portion or summary of it. The summary contains fewer bits of secret information, providing a mathematical limit on the inferences an attacker could draw. For instance, an e-commerce web site prints only the last four digits of a credit card number, a photograph is released with a face obscured, an appointment scheduler shows what times I'm busy but not who is meeting me, a document is released with text replaced by black rectangles, or a strategy game reveals my moves but not the contents of my board. However, it is not easy to determine by inspection how much information a program's output contains. For instance, if a name is replaced by a black rectangle, it might appear to contain no information, but if the rectangle has the same width as the text it replaces, and different letters have different widths, the total width might determine which letters were replaced. Or a strategy game might reveal extra information in a network message that is not usually displayed.

The approach of *quantitative* information-flow security expresses a confidentiality property as a limit on the number of bits that may be revealed. Quantification is a general approach that can be used with any confidentiality policy, since any kind of information can be measured in bits. Of course, for quantification to be useful, the allowable flows must contain less information than the undesirable ones (perhaps after distinguishing different classes of secret information). This is not necessarily the case, but we have found quantification to be useful over a wide variety of examples (see Chapter 7).

Quantification could potentially apply to very small flows: for instance, an unsuccessful login attempt reveals only a small fraction of a bit, if the attacker had no previous knowledge of the password. But in the examples we consider, the acceptable flows may be any number of bits.

1.1.2 Soundness for information-flow measurement

A program analysis is *sound* if its results are always correct in their intended interpretation, though they may still not be the most informative results possible. A confidentiality tool is sound if its measurement is an upper bound: it can overestimate the amount of information revealed, but can never underestimate it. (Note that for our technique, this is still a property of the tool's measurement of a single execution: the results for one execution need not apply to other executions.)

In some violations of information-flow policies, confidential data is exposed directly, for instance if the memory containing a password is not cleared before being reused. However, in many other cases information is transformed among formats, and may eventually be revealed in a form very different from the original input. This is what makes soundness a challenge: a tool must account for all of the influence that the secret input has on the program's output, even when the influence is indirect. Specifically, this means a tool must account for *implicit flows* in which the value of a variable depends on a previous secret branch condition or pointer value.

1.1.3 Making information-flow measurement practical

A flow measurement technique can only find and prevent real information leakage bugs if it can be applied to the software systems that are really used with secret data. It must apply to real programming languages, preferably ones already used in practice. It must be able to operate correctly on large programs, and without using too much time or computing resources. And using the tool must not present too much of a burden to a developer, in terms of specifying a policy, describing how the program meets that policy, or interpreting a tool's results.

1.2 Our flow-measurement approach

This thesis presents a family of techniques for building information-flow measurement tools, and evaluates those techniques by using our Flowcheck tool to measure flows in six different real applications. We found that building a practical tool required a careful combination of well-known techniques, previous techniques applied in new ways, and new insights.

Our overall approach is dynamic analysis: our tool measures the execution of a program on one or more particular inputs. In addition to providing the program itself (which the technique considers public), a user must tell our tool which program inputs contain secret information. By default, the tool treats all outputs as potentially public. The tool only measures flows to the outputs of the program: information that might be revealed by other observable aspects of its behavior, such as its use of time or system resources, are outside the tool's scope. The tool's output is a flow bound that applies only to the examined execution: other executions might reveal either more information or less.

Our techniques build on the general information-flow approach of *tainting*: a variable or value in a program is tainted if it might contain secret data. The basic rule of tainting is that the result of an operation should be tainted if any of the operands is. However, the tainting analysis we use is unusual in two ways: it operates at the level of individual bits, and it soundly includes implicit flows (Chapter 3). Tainting is appropriate for determining whether an illegal flow is present or not, but it cannot give a precise measurement of secret information because of its conservative treatment of propagation. A single tainted input can cause many later values to be tainted, but making copies of secret data does not multiply the amount of secret information present.

A key new idea in this thesis is to measure information-flow not simply using tainting but as a kind of network flow capacity. One can model the possible information channels in an execution of a program as a network of limited-capacity pipes, and secret information as an incompressible fluid. The maximum rate at which fluid can flow through the network corresponds to the amount of secret information the execution can reveal. With appropriate algorithms, including a kind of graph compression, it is possible to build a flow graph as a program executes and compute the maximum flow through it on the fly (Chapter 5).

Alternatively, a graph's maximum flow capacity also corresponds to the minimum capacity of a cut. A cut is a set of edges whose removal disconnects the secret input from the public output, so the intermediate values that occurred on the cut edges are sufficient (along with public information) to determine the program's public output. A minimum cut (a cut with the smallest possible total capacity) is a set of locations in the program's execution at which the secret information being processed had the most compact representation. Any cut, whether provided by a developer or built automatically from a maximum flow (in which case it is guaranteed to be minimal), can be used by our tool as a means of checking whether an information-flow policy is satisfied on future executions. Cut-based checking is possible with much with less overhead than building a new graph (Chapter 4).

Though our tool's results describe only the particular executions it was used to measure, there are several ways it can be used to ensure that all program executions obey a confidentiality property. One choice is to use the tool to measure the information flow in every execution, and abort an execution if it is about to reveal too much information. This kind of runtime checking incurs a performance overhead, but the use of cut-based checking can make it small enough (a factor of 2) to be acceptable for some security-sensitive applications. Second, the tool's results highlight which parts of a program operate on secret data and where that data has a compact representation, which can direct developers to secrecy-relevant code that should be manually audited.

A static information-flow measurement tool, one which would produce a formula that describes the information a program reveals on any possible execution, is not yet practical for the realistic-scale programs we consider. However, our technique represents significant progress towards that goal in two respects. First, its results could be used as a hypothesis to be proved or disproved by a static tool, relieving developers of the task of creating such a specification from scratch. Second, our dynamic approach suggests an analogous static approach that could take advantage of a number of previously-known static techniques, clarifying why static quantitative information flow remains difficult and suggesting more specific targets for future research (Section 8.2.2).

Our implementation of these ideas in the Flowcheck tool meets the three requirements we outlined earlier. Flowcheck produces a quantitative flow measurement, so a program that allows some inference about the secret information can still be acceptable, as long as the amount of information revealed is not too much. Flowcheck's results are a sound measurement of the flow in a particular execution or set of executions: it never underestimates a flow that occurs. Finally, Flowcheck is practical for developers to use: we have run it on C, C++, and Objective C programs of up to half a million lines of code, and its running time scales linearly with the length of the program execution. Flowcheck currently uses a small number of developer-provided annotations to improve the precision of its results, but we found that only a few annotations (based on local reasoning that was easy even when we were not previously familiar with the code) were sufficient (Chapter 7).

1.3 Outline

The remainder of this thesis studies these ideas in more detail. Chapter 2 begins by surveying previous research on systems and tools to support information-flow security. Next, Chapter 3 describes the most fundamental information-flow analysis techniques we use: tainting analysis, including at the level of individual bits, and a way to account for implicit flows. Chapter 4 covers cuts in a flow network, and explains two ways they can be used to more efficiently check whether a quantitative policy is respected. Chapter 5 describes how to obtain more precise flow measurements by explicitly constructing a graph representation of flows in a program execution, and computing the maximum flow in that graph. Chapter 6 provides a more detailed formal justification of claims made earlier by giving a definition of soundness for a dynamic quantitative information-flow analysis, and then introducing a simulation-based proof technique to demonstrate how a model of the previously discussed analyses satisfies this soundness definition. Chapter 7 discusses our implementation of a quantitative information-flow analysis for Linux/x86 binary programs, and evaluates it on security policies from a series of open-source applications. Finally, Chapter 8 provides further discussion, suggests directions for future research, and concludes.

Chapter 2

Related work

This chapter surveys previous techniques for information-flow security that work at the operating system level (Section 2.1), and then two broad classes of techniques that work at the programming language level: static analyses (including type systems) that check programs for information-flow security ahead of time (Section 2.2), and dynamic tainting analyses that track data flow in programs as they execute (Section 2.3).

As described in Chapter 1, our goal is a flow measurement tool that gives a quantitative result, whose result are sound, and which can be applied to large preexisting programs. As summarized in Figure 2-1, several of the surveyed tools meet one or two of these requirements, but none meets all three.

2.1 Operating system techniques

It is relatively straightforward for an operating system to enforce an information-flow policy at the granularity of processes and files. In this context, such enforcement is referred to as "mandatory access control," "mandatory" in the sense that the restrictions on access to a file are chosen by the operating system, rather than by the user or processes that created the file. Each file or other data source has a secrecy level. To implement an information-flow policy, the system keeps track of a level of secrecy per process which is equal to the most secret information the process has ever read: any data written by the process is then marked as at least that secret. Because

	Partial flows	Soundness	Scalability
Operating system	Trusted declassifier processes	Yes, dynamic checking always enabled	Many applications require redesign
Static type system	Declassification or quantification	Almost always	Hard to retrofit to existing programs
Dynamic tainting	Trusted declassification	Requires additional static analysis	Often quite good
Our approach	Quantitative flow policy	Upper bound for observed executions	Large pre-existing C and C++ programs

Figure 2-1: Previous approaches to information-flow analysis support subsets of our goals as introduced in Figure 1-1. For instance, operating system techniques are generally sound because they explicitly check all of a program's interactions; some static analysis approaches support partial flows via quantification; and many dynamic tainting tools are scalable in terms of developer effort and runtime overhead. However, no single previous approach satisfies all three of our goals.

the operating system can be aware of all of the interactions between processes, such a policy ensures that any information produced under the influence of a secret is secret.

However, such a policy often treats too many results as secret to be useful. An operating system generally cannot know how programs use the information they read, so it must assume that any part of the program's input might influence any part of its output; unfortunately, such pessimistic assumptions rule out whole classes of applications. For instance, if the goal is to prevent an e-commerce application from releasing credit card information, an operating system might prevent a process that has read the information from using the network, or transmitting any data to a process that does. While this would indeed keep the credit card number secret, it would also make it impossible to send any reply at all to a user in response to a purchase request.

The traditional use of mandatory access control is in the context of multilevel-secure operating systems, which are designed to protect classified information primarily in military or intelligence contexts. Such systems can provide a high degree of assurance that execution will respect an information-flow policy, through a combination of providing a limited set of capabilities, carefully auditing an implementation, and formally verifying the underlying security model (for instance, as in the classic work of Bell and La Padula [BP76]). More recent work has applied similar models to provide mandatory access control as an addition to standard operating systems, for instance in the Security-Enhanced Linux (SELinux) project [LS01]. Though civilian applications so far have focused on integrity concerns (e.g., protecting systems from subversion via attacks on network servers), a system such as SELinux could in principle provide very robust confidentiality protections. However, such protections would not be very useful for any application in which secret and public data are both used, because they would apply at too coarse a level of granularity.

To protect systems while still allowing useful work, a finer-grained tracking mechanism is required. For instance, the Asbestos operating system [EKV+05] provides a lightweight abstraction ("event processes") for provide memory isolation of computations on data belonging to a particular user. Similarly, the HiStar [ZBWKM06] operating system provides a new architecture whose processes are suitable for containing secret information, and Flume [KYB+07] shows how such processes could be hosted inside a legacy operating system. In all of these systems, the processes that confine information must have a single purpose (since they have a single information-flow label), and their possible outputs are constrained. This is compatible with some application models, such as CGI scripts that execute a new process for each web request. However, current systems often use a few large server processes that handle multiple requests and maintain state between requests for efficiency. Such systems would need to be redesigned to use process-level flow confinement. Language-level isolation mechanisms, like the ones in our system, can be applied to existing software without reimplementing.

2.2 Fully static analysis

Static checking aims to verify the information-flow security of programs before executing them [Den76]. The most common technique uses a type system, along with a declassification mechanism (a type loophole) to allow certain flows. It is also possible to quantify information flows in a static system as a mechanism for allowing certain flows, though this has been difficult to make practical.

Despite significant advances, barriers remain to the adoption of information-flow type checking [VSI96] extensions to general purpose languages [Mye99, Sim03, LZ06]. Static type systems may also be too restrictive to easily apply to pre-existing programs: for instance, we are unaware of any large Java or OCaml applications that have been successfully ported to the Jif [Mye99] or Flow Caml [Sim03] dialects. (Closest are three Jif programs: the poker game of Askarov and Sabelfeld [AS05], 4,500 lines of code ported from Java over the course of 230 hours; the email client of Hicks et al. [HAM06], 6,000 lines of code written from scratch over "hundreds" of hours; and the Civitas electronic voting system [CCM08], 13,000 lines written from scratch.) Techniques based on type safety are inapplicable to languages that do not guarantee type safety (such as C) or ones with no static type system (such as many scripting languages). By contrast, our current implementation is for C and related languages, and could also be extended to scripting languages (discussed in Section 8.2.3).

Information-flow type systems generally aim to prevent all information flow. Many type systems guarantee non-interference, the property that for any given public inputs to a program, the public outputs will be the same no matter what the secret inputs were [GM82, VSI96]. Because it is often necessary in practice to allow some information flows, such systems often include a mechanism for *declassification*: declaring previously secret data to be public. Such annotations are trusted: if they are poorly written, a program can pass a type check but still leak arbitrary information. In other words, the particular possibilities for declassification in a system are regarded as part of the security policy that the program must satisfy. There has been significant research on ways to put additional policy restrictions on declassification: restricting it not just to certain code [FSBJ97], but to certain principals [ML97], to certain runtime conditions [CM08], according to the integrity of the condition triggering it [MSZ04], or a number of other restrictions [SS05].

However, declassification still introduces a difficult tension between defining the declassification policy narrowly, and keeping the policy independent of a particular implementation. Developers may be left with a false sense of security if declassification is allowed under a condition on program state that is intended to reflect a specification condition, but is really dependent on other untrusted code. (Just the fact that a boolean variable named `authorized` is set to true does not mean that a request was actually properly authorized.) Purely quantitative policies, such as the ones used with our tool, have a clear implementation-independent interpretation, but it is left to developers to connect them to application-specific security goals.

The flow-graph cuts described in Chapter 4 are the closest analogue to declassification annotations in our system, and they embody the same intuition that a partial-flow policy can be enforced by allowing flow at a certain program location, and prohibiting it everywhere else. The minimum cut locations that our tool finds automatically could be used as suggested locations for declassification annotations. However, the fundamental difference between declassification annotations and our cuts is in their relation to policy. A declassification annotation becomes part of the policy the type system guarantees (i.e., is trusted), while in our system the policy is a numeric flow bound expressed independently of the program, and the cut is an untrusted hint pointing out one way the program might satisfy the policy. Both declassification annotations and cuts can be used to focus programmer attention on a part of a large program that is important for information-flow security, but declassification annotations ask the programmer to understand the declassification point completely, so that any information passing through it can be deemed legitimate. In our system the amount of information revealed is the ultimate criterion of acceptability, and examining the code around a cut can help the programmer understand why the program reveals the information it does (perhaps suggesting ways it could be changed to reveal less).

Quantitative measurements based on information theory have often been used in theoretical definitions of information-flow security [Gra91, DHW02, Low02], and some more recent work has attempted to build practical program analyses based on them. For instance, Lowe [Low02] begins with the same abstract definition of channel capacity that we argue for in Chapter 6, but explores it in the context of a process algebra rather than a programming language. The definition he formulates in that context accounts for subtleties of nondeterminism and fine-grained interaction between concurrent processes, but it is not clear even how to translate it to an imperative context, much less automate its application. Clark et al.'s system for a simple while language [CHM04] is the most complete static quantitative information flow analysis for a conventional programming language. Any purely static analysis is imprecise for programs that leak different amounts of information when given different inputs. For instance, given an example program with a loop that leaks one bit per iteration, but without knowing how many iterations of the loop will execute, the analysis must assume that all the available information will be leaked. Malacaria [Mal07] gives a formula for precise per-iteration leakage bounds for loops, but it appears difficult to apply automatically. By contrast, a dynamic technique like ours can simply count the number of iterations that occur on a particular execution.

2.3 Dynamic analysis

Processing sensitive information often involves a sequence of calculations that transform sensitive input into a different-looking output that contains some of the same information (Chapter 7 describes examples from games, graphical and web user interfaces, image processing, and network protocols). To catch violations of confidentiality policies in such software, it is important to examine the flow of information through calculations, including comparisons and branches that cause implicit flows, not just to track data that is copied directly. Several recent projects dynamically track data flow for data confidentiality and integrity, but (unlike our approach) without a precise and sound treatment of implicit flows.

Some of the earliest proposed systems for enforcing confidentiality policies on programs were based on run-time checking: Fenton discovered the difficulties of implicit flows in a tainting-based technique [Fen74], and Gat and Saal propose reverting writes made by secret-using code [GS76] to prevent unintended flows (implicit and otherwise) The general approach closest to ours, in which run-time checking is supplemented with static annotations to account for implicit flows, was first suggested by Denning [Den75]. However, these techniques are described as architectures for new languages or hardware, rather than for as tools evaluating existing software, and they do not support permitting acceptable flows or measuring information leakage.

Many recent dynamic tools to enforce confidentiality policies do not account for all implicit flows. Chow et al.'s whole-system simulator TaintBochs [CPG+04] traces data flow at the instruction level to detect copies of sensitive data such as passwords. Because it is concerned only with accidental copies or failures to erase data, TaintBochs does not track all implicit flows. Masri et al. [MPL04] describe a dynamic information-flow analysis similar to dynamic slicing, which recognizes some implicit flows via code transformations similar in effect to our simple enclosure region inference (Section 7.8). However, it appears that other implicit flows are simply ignored, and their case studies do not involve implicit flows. DYTAN [CLO07], a generic framework for tainting tools, applies a similar technique at the binary level, where the difficulties of static analysis are even more acute. In case studies on Firefox and `gzip`, they found that their partial support for implicit flows increased the number of bytes that were tainted in a memory snapshot, but they did not evaluate how close their tool came to a sound tainting. For instance, they mark the input to `gzip` as tainted, much as we do with `bzip2` in Section 5.3.2, but do not measure whether the output was tainted.

Accounting for all implicit flows requires static information (in our approach, provided by the enclosure regions described in Section 3.2) Several projects have combined completely automatic static analyses with dynamic checking; the key challenge is to make such analysis both scalable and sufficiently precise. The RIFLE project [VBC+04] proposes an architectural extension in which dedicated hardware tracks direct and indirect information flow with compiler support. The authors demonstrate promising results on some realistic small programs, but their technique's dependence on alias analysis leaves questions as to how it can scale to programs that store secrets in dynamically allocated memory. Our approach also uses a mix of static analysis and dynamic enforcement, but our static analysis only needs to determine which locations might be written, while RIFLE attempts to match each load with all possible stores to the same location, which is more difficult to do precisely in the presence of aliasing. Two recent tools [NSCT07, CF07] apply to Java programs, where static analysis is somewhat easier: their experimental results show low performance overheads, but do not measure precision. None of these tools enforces a quantitative security policy.

In attacks against program integrity, the data bytes provided by an attacker are often used unchanged by the unsuspecting program. Thus, many such attacks can be prevented by an analysis that simply examines how data is copied. Quantitative policies are rarely used for integrity; one exception is recent work by Newsome and Song [NS08], which measures the channel capacity between an input and a control-flow decision to distinguish between legitimate influence and malicious subversion. Their measurement technique, based on querying the space of possible outputs with a decision procedure, is very different from ours, and their similar choice of channel capacity as a goal was independent.

The most active area of tainting research is on tools that prevent integrity-compromising attacks on network services, such as SQL injection and cross-site scripting attacks against web applications and code injection into programs susceptible to buffer overruns. These tools generally ignore implicit flows or treat them incompletely. Newsome and Song's TaintCheck [NS05], and the Flayer tool [DO07] are based on the same Valgrind framework as our tool, while other researchers have suggested using more optimized dynamic translation [KBA02, QWL+06], source-level translation [XBS06], or novel hardware support [SLZD04] to perform such checking more quickly. The same sort of technique can also be used in the implementation of a scripting language to detect attacks such as the injection of malicious shell commands (as in Perl's "taint mode" [WS91]) or SQL statements [NTGG+05].

Chapter 3

Basic techniques

This thesis considers several combinations of mechanisms for measuring and checking information flows, but the two techniques described in this chapter are the most fundamental; both will be used as the basis for later techniques and in all the case studies. First, *tainting* (Section 3.1) is the standard dynamic analysis technique for tracking the flow of secret data; we describe the somewhat unusual choice of performing this analysis at the level of individual bits. Second, *enclosure* (Section 3.2) is the key technique our tool uses to soundly measure implicit flows. We then illustrate these techniques with an example (Section 3.3), and discuss an optimized implementation technique (Section 3.4).

3.1 Tainting

Operationally, a taint analysis computes whether each piece of data in a program is public or secret, according to the rule that if any of the inputs to a basic operation is secret, the output is. (The term comes from the intuition that even a small amount of secret data can "taint" a mixture in which all the other data values are non-secret.) Alternatively, tainting can be thought of as reachability: a piece of data is tainted if it is reachable from the secret inputs by some sequence of operations. There are both static tainting analyses, which compute the secrecy of program variables (e.g., [HYH$^+$04, LL05, XA06]), and dynamic tainting analyses, which can assign a different secrecy status to each data value that occurs during execution (a number of previous tools are discussed in Section 2.3). In this thesis, "tainting" will refer to dynamic tainting unless otherwise qualified.

Tainting can be applied to data at any level of granularity, with the choice reflecting a trade-off between analysis effort and precision. Since our goal is precise numeric estimates of the number of bits of secret information that flow, our tool's results benefit from the finest-grained tainting: we thus choose to track the secrecy of each bit separately (this could also be described as a *bit-tracking analysis*). For instance, the flows that our tool measures for several of the case studies in Chapter 7 are no more than a dozen bits, so its results would be much less precise if they were constrained to be multiples of 8 or 32 bits (as in byte- or word-level tainting). Bit-level tainting can also be loosely described as a *bit-width analysis*, because it computes how many bits worth of secret data are stored in a machine word, but it is not limited to situations where the secret bits are contiguous and start at the least-significant position, nor does it require that the non-secret bits have value 0 (though these are all the most common situations).

Our tainting analysis is an instance of a *shadow-value analysis* [NS07]: for each of the values the original program uses, it maintains a parallel value containing metadata used by the analysis. In our case, for each data bit in the original program (i.e., in a register or in memory), it maintains a *secrecy bit* which is `1` if the data bit might contain secret information, or `0` if it is non-secret.

In theory, a very precise bit-level tainting analysis could be derived by expressing each basic program operation with a circuit (e.g., addition using a ripple-carry adder), and then applying the basic principle of tainting to each bit operation. Another way to think about the desired results is that they come from a Kleene-style three-valued logic, in which the third logic value, traditionally glossed as "unknown," instead represents "secret": the analogy is that from the perspective of public results, the values of secret bits should be irrelevant. A result bit is secret if it might be either 0 or 1 depending on the values of secret input bits. However, it would be more precise to call the tainting analysis a four-valued logic (with values `0`, `1`, $\texttt{?}_0$, and $\texttt{?}_1$), since

$$
\begin{aligned}
s_{a\oplus b} &= s_a \mid s_b \\
s_{a\&b} &= (s_a \mid s_b) \mathbin{\&} (s_a \mid v_a) \mathbin{\&} (s_b \mid v_b) \\
s_{a+b} &= s_a \mid s_b \mid (((v_a \mathbin{\&} \overline{s_a}) + (v_b \mathbin{\&} \overline{s_b})) \oplus ((v_a \mid s_a) + (v_b \mid s_b))) \\
s_{a\cdot b} &= s_a \mid s_b \mid -(s_a \mid s_b)
\end{aligned}
$$

Figure 3-1: Examples of formulas for computing the secrecy bits of for the results of a basic operation, in terms of the secrecy bits of the operands (s_a and s_b) and the values of the operands (v_a and v_b). (& and | represent bitwise-AND and -OR, as in C, $\oplus$ is XOR, and an overline indicates bitwise complement.) The first formula says that a bit in the XOR of two values is secret if either of the corresponding operand bits was secret. The second formula for AND is similar, but a bit is also public if either one of the operand bits was both public and `0`. To motivate the expression for s_{a+b} in the third line, note that $v_x \mathbin{\&} \overline{s}_x$ and $v_x \mid s_x$ are the smallest and largest values that can be formed by replacing the secret bits in an operand x with either `0` or `1`; the bits set in the XOR expression are those carry-in bits that might be vary depending on the secret bits. The formula for multiplication in the last line is an imprecise approximation that allows tainting to propagate maximally leftward ($x \mid -x$, where $-$ represents twos-complement negation, has a `1` in every position not to the right of the rightmost `1` in x).

$$
\begin{aligned}
&\texttt{01010111} \oplus \texttt{0000?}_1\texttt{?}_0\texttt{?}_1\texttt{?}_1 = \texttt{0 1 0 1 ?}_1\texttt{?}_1\texttt{?}_0\texttt{?}_0 \\
&\texttt{01010111} \mathbin{\&} \texttt{0000?}_1\texttt{?}_0\texttt{?}_1\texttt{?}_1 = \texttt{0 0 0 0 0 ?}_0\texttt{?}_1\texttt{?}_1 \\
&\texttt{01010111} + \texttt{0000?}_1\texttt{?}_0\texttt{?}_1\texttt{?}_1 = \texttt{0 1 ?}_1\texttt{?}_0\texttt{?}_0\texttt{?}_0\texttt{?}_1\texttt{?}_0 \\
&\texttt{01010111} \cdot \texttt{0000?}_1\texttt{?}_0\texttt{?}_1\texttt{?}_1 = \texttt{?}_1\texttt{?}_0\texttt{?}_1\texttt{?}_1\texttt{?}_1\texttt{?}_1\texttt{?}_0\texttt{?}_1
\end{aligned}
$$

Figure 3-2: Examples of secrecy-aware computations using the formulas of Figure 3-1. `0` and `1` represent public zero and one bits, while $\texttt{?}_0$ and $\texttt{?}_1$ represent secret bits whose secret values are zero or one respectively.

even if a bit is secret, the analysis still remembers what its value is.

The circuit- or logic-level models of the previous paragraph identify the ideal that our real implementation approximates, but for practicality, our tool computes tainting results a full machine word at a time, using the hardware's word-sized operations. The secrecy bits for the result of a basic operation are computed by a formula that in general depends on both the secrecy bits of the operands and their data bits. Depending on the operation, this formula may or may not involve a loss of precision compared to the best result representable in the tainting abstraction; for instance, the formulas our tool uses for bitwise operations, addition, and subtraction are maximally precise, but those for multiplication and division are not. Some representative examples of these formulas are shown in Figure 3-1 (a complete discussion is in [SN05]), and some concrete examples are shown in Figure 3-2.

The bitwise tainting analysis is essentially the same as the analysis that the Valgrind Memcheck tool [SN05] uses to track undefined values; in fact, our observation of this similarity was one of the original motivations for this research project. Our implementation reuses most of the code from Memcheck. The independently-conceived Flayer tool [DO07] also re-purposes Memcheck's undefined value analysis as a taint analysis, but for finding input-processing vulnerabilities (a class of non-quantitative integrity properties without implicit flow).

3.2 Enclosing leaks from implicit flows

The tainting intuition is a good match for programs represented as circuits, but general programs are more complex than circuits because of operations such as branches, arrays, and pointers that allow data to affect which operations are performed or what their operands are. When the branch condition, array index, or pointer value is secret, these operations can lead to indirect or *implicit* flows which do not correspond to any direct data flows, but may nonetheless reveal secret information. For instance, later execution might be affected by a branch that caused a location not to be assigned to, or the fact that the 5th entry in an array is zero might reveal that the secret index used in a previous store was not equal to 5. To deal properly such situations, a sound flow measurement tool must account for all implicit flows.

To recover the intuitive perspective of execution as a circuit, our tool treats each operation (e.g., branch) that might cause an implicit flow as being *enclosed* as part of a larger computation with defined outputs. For instance, consider computing a square root. If a single hardware instruction computes square roots, then there is no implicit flow, but the square root of a secret value is itself secret. On the other hand, if the square root is computed by code that uses a loop or branches on the secret value, these implicit flows can be conservatively accounted for by assuming that they might all affect the computed square root value, so our tool can represent the implicit flows by ensuring the computed square root is tainted. Our tool applies this concept of enclosure in two ways: the complete program is always enclosed with its explicit outputs, and smaller regions of code can be enclosed with an annotated set of output locations.

To achieve soundness, it is sufficient to consider the entire program as being enclosed in this way. Our tool's default behavior treats all implicit flows as potentially revealing information to the program's public output, and sums the amount of information that might be revealed by each. (The amount of information revealed by the program is also bounded by the total size of all the outputs.)

Better precision results from using additional *enclosure regions* around smaller sub-computations, such as the square-root function mentioned earlier. In our system, enclosure regions are specified using source code annotations that mark a single-exit control-flow region and declare all of the locations the enclosed code might write to (see Section 3.3 for an example). These annotations can be inferred using standard static analysis techniques; Section 7.8 describes a pilot study examining what is required. Missing or poorly placed enclosure annotations can only cause the tool to give imprecise results, but it is necessary for soundness that a region declare all the locations it might write to, which is why it would be desirable for these to come from a sound static analysis. Our tool can also dynamically check that the soundness requirements for an enclosure region hold at runtime, but this is less satisfactory because if a check fails, it is not always possible to continue execution in a way that is both sound and behavior-preserving.

Conceptually, the number of bits of information that can flow from an implicit flow operation corresponds to the number of possible different executions: for instance, a two-way branch on a secret reveals one bit, while a pointer operation such as an indirect load, store, or jump could reveal as many bits as are secret in the pointer value. (At the instruction level, multi-way branches show up as either nested two-way branches or jump tables, and our tool estimates their flow accordingly.) As described so far, our tool can only take advantage of this precision when counting bits of information revealed directly to the final program output. When the flow is represented by tainting intermediate program values, our tool must consider all of the outputs of an enclosed computation as tainted when any implicit flow operation on a secret value occurs inside, since in general all the output bits might be affected by even a single branch. (We will see further ways to use this precision in Section 5.1.)

3.2.1 Enclosure and side channels

In addition to handling simple implicit flows to other data values, enclosure regions can also be used to limit the amount of information revealed via some other kinds of observable behavioral differences (so-called *side channels*; here we discuss program output, termination, and timing), though with varying degrees of practicality. In each case, the goal is to prevent the decisions made within the enclosure region from being visible outside (other than via the enclosed outputs).

In addition to the information contained in values that a program prints, its choices of when and how many values to output may also convey information. So that these channels are properly counted, the control flow decisions that decide when values are output should occur outside any enclosure regions: our tool does this by prohibiting output routines inside enclosure regions.

The termination of a program can also be considered a distinguished kind of output for which the decision of when to terminate may carry information (on Unix, termination also can pass a one-byte exit status value). It is somewhat more difficult to prohibit exits inside enclosure regions, since a program can be killed by an exceptional event like a out-of-bounds memory access as well as an explicit call to `exit`. However, it is possible to catch such events and continue execution at the end of the enclosure region; our implementation of dynamic checking for regions maintains a log of memory changes made inside a region that can be rolled back to ensure execution can continue from the enclosure end.

A particularly notorious side channel is the timing of a program's execution of which non-termination can be thought of as a special case. In theory, this could also be addressed by requiring that each execution of an enclosure region take the same amount of time, but such a policy would be impractical to specify and enforce in practice, so we have not implemented anything along these lines. (For short code

```
/* Print all the "."s or "?"s,
   whichever is more common. */
void count_punct(char *buf) {
    unsigned char num_dot = 0, num_qm = 0, num;
    char common, *p;
    ENTER_ENCLOSE(num_dot, num_qm);
    while (p = buf; *p != '\0'; p++)
        if (*p == '.')
            num_dot++;
        else if (*p == '?')
            num_qm++;
    LEAVE_ENCLOSE();
    ENTER_ENCLOSE(common, num);
    if (num_dot > num_qm) {
        /* "."s were more common. */
        common = '.'; num = num_dot;
    } else {
        /* "?"s were more common. */
        common = '?'; num = num_qm;
    }
    LEAVE_ENCLOSE();
    /* print "num" copies of "common". */
    while (num--)
        printf("%c", common);
}
```

Figure 3-3: C code to print all the occurrences of the most common punctuation character (`.` or `?`) in a string. For instance, when run on its own source code, the program produces the output “........”. We might intuitively describe this program as revealing 9 bits of information about its input: 1 bit giving the identity of the most common punctuation mark, and 8 bits from the count, which is computed modulo 256. Later chapters will describe how to make this intuition precise (Chapter 6), and techniques that obtain that result (Chapters 4 and 5).

segments, a code translation approach is possible [MPSW05], but this would probably not scale to enclosure regions that might execute a wide variety of operations.)

3.3 Punctuation count example

As a concrete example of the techniques introduced in this chapter, consider the code shown in Figure 3-3. This function counts the number of periods and question marks in a string, and then whichever was more common, prints as many as appeared in the string (modulo 256, because it uses an 8-bit counter). For instance, the source code contains 8 periods and 4 question marks, so that when run on its own source the program prints 8 periods.

The only relationships between the input buffer and `num_dot`, between `num_dot` and `common`, and between `num` and the output, are implicit flows. A tool that did not account for all of them could give an unsound (too small) result.

The example contains two enclosure regions, spanning lines 6–12 and 13–21 respectively. The enclosure regions marked by `ENTER_ENCLOSE` and `LEAVE_ENCLOSE` improve the precision of the results: without them, the default treatment of enclosing the entire program would cause the tool to measure a leak of 1 bit each time a value from the input buffer was compared to a constant, 1855 in total when run on its own source. However, even when enhanced with enclosure regions, tainting still gives a fairly imprecise result for this example: all of the output bytes are tainted, since they all depend on the input string. For instance, when run on its own source code, producing a string of 8 periods as output, this means that the program would be estimated as revealing 64 bits of information. In later chapters we will introduce additional techniques to allow a much more precise flow bound (9 bits in this example).

A simple tainting analysis like the one described in this chapter is still most appropriate if it is more important to know which output bits contain secret information than to know how much information they contain. The voting software case study of Section 7.3 is such an example; for it we used only these basic techniques.

3.4 Optimizing large-region operations

Because the output of an enclosure regions can be an entire array or other large data structure, the tool often needs to represent the fact that a piece of information might flow to any byte in a large memory region. It would be too slow to do this by modifying the secrecy of each memory location individually. For instance, consider a loop operating on an array in which each iteration might potentially modify any element (say, if the index is secret). Operating on each element during each iteration would lead to quadratic runtime cost.

Instead, the tool performs operations on large memory regions lazily. It maintains a limited-size set (default size: 40) of region descriptors, each of which describes a range of more than 10 contiguous memory locations, along with another list of up to 30 addresses excepted. Operations such as tainting an entire region are recorded just by modifying the descriptor, and operations on single addresses are marked as exceptions. However, if a region accumulates more than 30 exceptions, it is either shrunk to exclude them (if they are all in the first half), or eliminated.

Chapter 4

Cut-based techniques for more efficient checking

A cut (sometimes more specifically called an *s-t cut* [CGK+97]) is a way of dividing a flow graph into two pieces, one containing the source and the other the sink. A cut can be defined as the set of nodes that lie in the half containing the source, but we are interested in the set of edges that cross from that set to its complement; their removal disconnects the source from the sink. The capacity of a cut is the sum of the capacities of these edges. There is a close relationship between cuts of a graph and flows through it: the amount of any flow is less than or equal to the capacity of any cut, since the flow must traverse the cut on its way from the source to the sink. In this chapter we describe two techniques that use this relationship to bound the information flow in a program: as long as a tool can ensure that no secret information reaches the sink other than by traversing a set of locations in a program execution, those locations form a cut, and their capacity is a bound on the information revealed. These techniques are applicable however such a cut is determined (for instance, a developer might supply it by hand), but later in Section 5.4, we will explain how a cut of minimal capacity can be determined automatically.

These runtime mechanisms can be used to check whether a desired information-flow policy, as embodied in a cut, is obeyed during a particular execution. It is a small additional step to enforce such a policy, in the sense of terminating the program or taking another corrective action right before the program reveals more information than it should, but there is the additional subtlety that such action may itself reveal some information. This is another instance of the termination channel mentioned in Section 3.2, and the flow it causes can be bounded in the same way.

4.1 Preemptive leakage

Checking that no secret information reaches the output other than across a given cut is an instance of a tainting problem, so one approach is to add a cut to the tainting technique of Chapter 3. The only new behavior required is that when tainted bits reach the cut, they are cleared, and the count of leaked bits incremented; the same treatment given to bits that reach the output. We call this treatment *preemptive leakage* since the bits are counted as leaked at the moment they reach the cut, rather than later when they reach the output. If the cut (or *preemptive leakage annotation*) is placed at a location where the secret information has a compact representation, then counting the leakage early can give a more precise flow estimate than waiting until it reaches the output.

We implemented preemptive leakage annotations in the original version of our information-flow measurement tool [ME06], and performed the preliminary versions of some of the case studies described in Chapter 7 with manually placed annotations. Running our tool in this mode still requires the fairly expensive runtime mechanisms of bit-tracking and enclosure regions, though the measurement they provide is unnecessary for bits counted at a cut. Thus, if it not important to retain the capability to report where flows outside the cut appear, a more efficient implementation of cut checking, such as the one described in Section 4.2, would be preferable.

4.2 Two-process simulation

A more efficient cut-checking technique is based on running two copies of a program. The basic idea is to run two copies of a program in lockstep, one which initially has access to the secret input, and the other which operates on a non-sensitive input of the same size. At the point when the programs reach a cut annotation, the program with the real secret input sends a copy of the values on the cut to the second copy. If the programs produce the same output, then the data that the second program received from the first at the cuts is the only secret information needed to produce the output, and the flow policy is satisfied. If the outputs diverge, then another flow is present and execution should be terminated. (The disadvantage compared to a tainting approach is that detecting a violation at output time is of less help in tracking down its cause.)

The key advantage of this technique is that the execution of the two programs can be mostly uninstrumented: they only need to behave unusually at the cut points. Enclosure regions are also not required, as long as the non-sensitive input is such that the program can execute the code that would be enclosed without crashing or looping. A factor of two overhead compares favorably with binary-level dynamic tainting systems, and using two copies can take advantage of multiple processors.

It may happen that some of the bits of the non-sensitive input happen to match the corresponding bits of the secret input purely by chance, so that this technique produces some outputs dependent on the secret before recognizing a divergence. However, its output in this case is still only a prefix of the result of running the program on the non-sensitive input, which is already available to an attacker and conveys no information. The best way for a two-process simulation to recover from a divergence is to hide it completely by continuing to return the outputs produced by the non-secret copy of the program.

A simpler version of this technique (without a cut, for checking only complete non-interference) has been implemented independently in an operating-system-level tool called TightLip [YMC07]. We will revisit the idea of two-process simulation, in a theoretical context, in Section 6.3.

Chapter 5

Algorithms for flow graphs

Previous chapters used the intuition of flow or cuts in a graph to motivate techniques for measuring information flow in program executions. This chapter extends this connection to graphs further by describing techniques that build a graph representation at runtime and operate on it directly. Most crucially, explicitly building this flow graph allows a tool to automatically compute the maximum flow and corresponding minimum cut for an execution, achieving optimal precision as compared to the manually-selected cuts described in Chapter 4, without the need for developer insight. We first discuss how to build a flow graph (Section 5.1), and then how to combine flow graphs for several executions (Section 5.2), efficiently compute the maximum flow in a graph (Section 5.3), and compute a minimum cut from a maximum flow (Section 5.4).

5.1 Building a flow graph

Our tool builds a graph representing the possible channels for information flow in a program execution using a dynamic shadow value analysis similar to the tainting analysis described in Section 3.1. In fact, the two analyses run in parallel: for each value, the tool associates a tag representing a node identity in the graph being constructed, and a set of secrecy bits. The node identities are used to build the structure of the graph, while the secrecy bits are counted to give the capacities on its edges.

```
Node regNode[NUM_REGISTERS]; /* Node identities for each register */
Node public; /* Distinguished node for non-secret values */

for each instruction "insn" {
    switch (insn) {
    case "r_i := c":
        regNode[i] := public;
    case "r_i := r_j":
        regNode[i] := regNode[j];
    case "r_i := r_j + r_k":
        Node n := newNode();
        newEdge(regNode[j], n);
        newEdge(regNode[k], n);
        regNode[i] = n;
    }
    execute insn;
}
```

Figure 5-1: Pseudo-code for the simplest version of the graph construction algorithm of Section 5.1. The array `regNode` contains shadow node identities for the value in each machine register. On each operation, the algorithm builds a new node to represent the result of the operation, and adds new edges connecting the operands to the result.

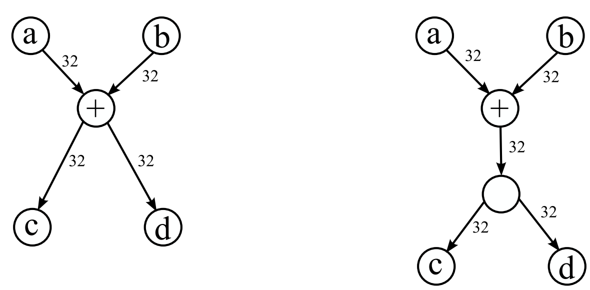

Figure 5-2: Two possible graphs representing the potential information flow in the expression `c = d = a + b`, where each variable is a 32-bit integer. The graph on the left permits 32 bits of information to flow from `a` to `c`, and a different 32 bits to flow from `b` to `d`. To avoid this, our tool uses the graph on the right.

The flow graphs our technique constructs represent an execution in a form similar to a circuit. For efficiency, the graph represents byte or word-sized operations. Edges represent values, and have capacities indicating how many bits of data they can hold. Nodes represent basic operations on those values, where the in-degree of a node is the

```
Node regNode[NUM_REGISTERS]; /* Node identities for each register */
Node public; /* Distinguished node for non-secret values */

for each instruction "insn" {
    switch (insn) {
    case "r_i := c":
        regNode[i] := public;
    case "r_i := r_j":
        regNode[i] := regNode[j];
    case "r_i := r_j + r_k":
        Node n := newNode();
        newEdge(regNode[j], n);
        newEdge(regNode[k], n);
        Node m := newNode();
        newEdge(n, m);
        regNode[i] := m;
    }
    execute insn;
}
```

Figure 5-3: An update to the pseudo-code of Figure 5-1, to give a more precise graph representation of the results of an operation, as described in Figure 5-2. New or changed operations are shown in bold.

operation's arity. Pseudo-code for the most basic version of this graph construction algorithm is given in Figure 5-1. (For simplicity, the pseudo-code omits the computation of edge capacities.) For each machine register, the algorithm maintains a node representing the operation that created its current value, or a distinguished node for literal values, which are considered public. At an operation, such as addition, a new node is created to represent the result of the operation, and it is connected by edges to the operation inputs.

Because of the possibility that the result of an operation may be used in more than one subsequent operation, our tool adds an additional single edge and node, which represents the constraint that the operation has only one output; this is also equivalent to giving a capacity limit on a node. See Figures 5-2 and 5-3 for graphical and pseudo-code illustrations.

Copying a piece of data without modifying it does not lead to the creation of new nodes or edges. Because memory is byte-oriented, each memory byte has its own

```
Node regNode[NUM_REGISTERS]; /* Node identities for each register */
Node memNode[2**32]; /* Node identities for each memory byte */
Node public; /* Distinguished node for non-secret values */

for each instruction "insn" {
    switch (insn) {
    case "r_i := c":
        regNode[i] := public;
    case "r_i := r_j":
        regNode[i] := regNode[j];
    case "r_i := r_j + r_k":
        Node n := newNode();
        newEdge(regNode[j], n);
        newEdge(regNode[k], n);
        Node m := newNode();
        newEdge(n, m);
        regNode[i] := m;
    case "r_v := load r_a":
        Node n := newNode();
        for each b (0 .. 3)
            newEdge(memNode[regs[a]+b], n);
        regNode[v] := n;
    case "store r_a, r_v":
        for each b (0 .. 3)
            memNode[regs[a]+b] := regNode[v];
    }
    execute insn;
}
```

Figure 5-4: An update to the pseudo-code of Figure 5-3, to add support for memory load and store operations. New or changed operations are shown in bold.

shadow value, and loads and stores of larger values are split into bytes for stores and recombined after loads (see Figure 5-4).

The graph is directed, with edges always pointing from older to newer nodes, and so is also acyclic. Inputs and output are represented by two distinguished nodes, a *source* node representing all secret inputs, and a *sink* node representing all public outputs (see Figure 5-5).

Implicit flow operations, such as branches and memory loads and stores, can reveal additional information if the branch condition or address value is secret. The tool represents these flows with additional edges originating from those values. In its

```
Node regNode[NUM_REGISTERS]; /* Node identities for each register */
Node memNode[2**32]; /* Node identities for each memory byte */
Node public, source, sink;

for each instruction "insn" {
    switch (insn) {
    case "r_i := c":
        regNode[i] := public;
    case "r_i := r_j":
        regNode[i] := regNode[j];
    case "r_i := r_j + r_k":
        Node n := newNode();
        newEdge(regNode[j], n);
        newEdge(regNode[k], n);
        Node m := newNode();
        newEdge(n, m);
        regNode[i] := m;
    case "r_v := load r_a":
        Node n := newNode();
        for each b (0 .. 3)
            newEdge(memNode[regs[a]+b], n);
        regNode[v] := n;
    case "store r_a, r_v":
        for each b (0 .. 3)
            memNode[regs[a]+b] := regNode[v];
    case "input r_i":
        Node n := newNode();
        newEdge(source, n);
        regNode[i] := n;
    case "output r_i":
        newEdge(regNode[i], sink);
    }
    execute insn;
}
```

Figure 5-5: An update to the pseudo-code of Figure 5-4, to add distinguished source and sink nodes that are the source and target of program inputs and outputs respectively. New or changed operations are shown in bold.

default behavior, our tool creates paths representing the possibility of flow from these operations directly to the program output. However, our technique still captures two constraints on such flows: the total amount of information revealed can never be more than the total amount of output, and information that leaks from an implicit flow

```
Node regNode[NUM_REGISTERS]; /* Node identities for each register */
Node memNode[2**32]; /* Node identities for each memory byte */
Node public, source, sink;
Node leak = newNode(); /* Implicit flows go here */

for each instruction "insn" {
    switch (insn) {
    case "r_i := c":
        regNode[i] := public;
    case "r_i := r_j":
        regNode[i] := regNode[j];
    case "r_i := r_j + r_k":
        Node n := newNode();
        newEdge(regNode[j], n);
        newEdge(regNode[k], n);
        Node m := newNode();
        newEdge(n, m);
        regNode[i] := m;
    case "if r_i > 0 goto l":
        newEdge(regNode[i], leak);
    case "r_v := load r_a":
        Node n := newNode();
        for each b (0 .. 3)
            newEdge(memNode[regs[a]+b], n);
        regNode[v] := n;
        newEdge(regNode[a], leak);
    case "store r_a, r_v":
        for each b (0 .. 3)
            memNode[regs[a]+b] := regNode[v];
        newEdge(regNode[a], leak);
    case "input r_i":
        Node n := newNode();
        newEdge(source, n);
        regNode[i] := n;
    case "output r_i":
        Node n := newNode();
        newEdge(regNode[i], n);
        newEdge(leak, n);
        newEdge(n, sink);
        Node l := newNode();
        newEdge(leak, l);
        leak := l;
    }
    execute insn;
}
```

Figure 5-6: An update to the pseudo-code of Figure 5-5, to add paths from each implicit flow operation to the program output. New or changed operations are shown in bold.

```
Node regNode[NUM_REGISTERS];
Node memNode[2**32];
Node public, source, sink;
Node leak = newNode(), enclosed_leak;
bool enclosed = false; /* In encl. region? */
Set<Addr> outputs; /* Locations written to */



for each instruction "insn" {
    switch (insn) {
    case "r_i := c":
        regNode[i] := public;
    case "r_i := r_j":
        regNode[i] := regNode[j];
    case "r_i := r_j + r_k":
        Node n := newNode();
        newEdge(regNode[j], n);
        newEdge(regNode[k], n);
        Node m := newNode();
        newEdge(n, m);
        regNode[i] := m;
    case "if r_i > 0 goto l":
        if (enclosed)
            newEdge(regNode[i], enclosed_leak);
        else
            newEdge(regNode[i], leak);
    case "r_v := load r_a":
        Node n := newNode();
        for each b (0 .. 3)
            newEdge(memNode[regs[a]+b], n);
        regNode[v] := n;
        if (enclosed)
            newEdge(regNode[a], enclosed_leak);
        else
            newEdge(regNode[a], leak);
    case "store r_a, r_v":
        for each b (0 .. 3)
            memNode[regs[a]+b] := regNode[v];
        if (enclosed)
            newEdge(regNode[a], enclosed_leak);
        else
            newEdge(regNode[a], leak);
        assert(regs[a] in outputs);
    case "input r_i":
        Node n := newNode();
        newEdge(source, n);
        regNode[i] := n;
    case "output r_i":
        Node n := newNode();
        newEdge(regNode[i], n);
        newEdge(leak, n);
        newEdge(n, sink);
        Node l := newNode();
        newEdge(leak, l);
        leak := l;
    case "enter_enclose(memLocs)":
        enclosed := true;
        outputs := memLocs;
        enclosed_leak := newNode();
    case "exit_enclose":
        enclosed := false;
        for each addr in (outputs) {
            Node n := newNode();
            newEdge(memNode[addr], n);
            newEdge(enclosed_leak, n);
            memNode[addr] := n;
        }
        assert(all registers are dead);
    }
    execute insn;
}
```

Figure 5-7: An update to the pseudo-code of Figure 5-6, to add support for enclosure regions. New or changed operations are shown in bold.

can only escape via an output that occurs later in program execution. To embody these constraints, our tool does not direct implicit flows directly to the graph sink; instead, it creates a chain of nodes, one for each output operation, that accumulate all the information that might be revealed by all preceding implicit flow operations (see Figure 5-6).

Enclosure regions (Section 3.2) allow the tool to give a more precise treatment of implicit flows, by directing their leakage not to the output of the entire program but to the results computed by a particular self-contained computation. For each such region, our tool creates a distinguished node that receives flows from each implicit flow operation, and has outgoing edges to all of the declared outputs of the region (see Figure 5-7).

```
/* Disjoint sets containing either original graph nodes or
   pairs of locations and SOURCE or TARGET markers */
UnionFind uf;

enum EndpointType { SOURCE, TARGET };

/* One edge class for each code location */
Set<Location> combined;

/* Total capacity of all the edges in a class */
Map<Location, int> total;

/* Input loop: collect edges and sum capacities */
for each original edge (u, v, location, capacity) {
    combined.add(location);
    total[location] := total[location] + capacity;
    uf.union(u, [location, SOURCE]);
    uf.union(v, [location, TARGET]);
}

/* Construct representative for each set of merged nodes */
Map<UnionFindSet, Node> node;
for each set s in uf.sets() {
    node[s] := fresh_label();
}
node[uf.find(source)] := source;
node[uf.find(sink)] := sink;

/* Output combined edges in terms of merged nodes */
for each loc in combined {
    Node u := node[uf.find([loc, SOURCE])];
    Node v := node[uf.find([loc, TARGET])];
    output_edge(u, v, total[loc]);
}
```

Figure 5-8: Pseudocode for the graph-combining algorithm described in Section 5.2. Each edge (u, v) is labelled with a program location and a flow capacity. The output is a smaller graph that allows at least as many flows, formed by replacing all the edges with the same location with a single edge, and merging the corresponding endpoints.

5.2 Combining and collapsing flow graphs

A second operation that our tool performs on flow graphs is combining the graphs from multiple executions of the same program into a single graph, which it does by merging all of the edges that correspond to a particular program location, adding their capacities. In fact, it is also sensible to apply this merging operation to a single flow graph, to collapse a large original graph into a smaller graph that allows all the same flows, an application we will revisit in Section 5.3.2.

More precisely, our graph combining algorithm labels each edge with a value that includes a static location (i.e., instruction address), and optionally a 64-bit hash of the calling context (stack backtrace), similarly to Bond and McKinley's probabilistic calling context [BM07]. Then, any number of labelled graphs can be combined by identifying edges with the same label (replacing them with a single edge whose capacity is the sum of the original capacities), and unifying all of the nodes the original edges are incident upon. This can be done in almost-linear time with a union-find structure: for each edge (u, v) with location l, merge the sets containing u and a placeholder for "source of edges at l", and similarly for v and "target of edges at l". This operation is shown in pseudo-code form in Figure 5-8.

When flow graphs are combined in this way, any sum of possible flows in the original graphs is possible in the combined graph, so a bound computed for the combined graph is still sound. On the other hand, the possible cuts in the combined graph correspond only to sets of cuts that appear in the same places in each original graph, excluding the possibility of lower flow bounds corresponding to inconsistently placed cuts. Section 6.4 will discuss in more detail the importance of this kind of consistency to the soundness of the tool's results.

5.3 Efficient maximum-flow computation

Computing the maximum flow in a network is a long-studied computational task, but the flow graphs constructed by our technique are both very large and fairly well-structured, so specialized optimizations are both necessary and possible. We have investigated both exact algorithms with the potential to be efficient (Section 5.3.1), and unconditionally efficient algorithms with the potential to be precise (Section 5.3.2). Empirically, the latter approach seems to work better.

5.3.1 Precise and potentially efficient approaches

The best general algorithms for computing a maximum flow have time complexity at least $O(VE)$, where V and E are the number of vertices and edges in the input graph [CLR90], but a dynamic program analysis is usually only feasible if its running time is close to linear in the running time of the original program. Therefore, a more specialized flow algorithm is called for. The flow graphs produced by our technique have a number of special features that could guide the choice of an algorithm. Because vertices and edges are added at the same time, the graphs are sparse; i.e., $E = O(V)$. The capacities of edges are all small integers; e.g., no more than 32 if the program only uses word-sized operations. Intuitively, the graph from a long program execution will be deep but not very wide: the length of a path from the source to the sink is unbounded, but the number of vertices that are in use at any moment is bounded by the size of the original program's memory.

In theory, this last "narrowness" property is be sufficient to give an algorithm that is linear in the execution length, since for any flow graph with at most k outputs, there is a bounded-size graph that allows the same flows. However, the best upper bound we have found on the size of such a mimicking graph is 2^{2^k} vertices [HKNR98, CSWZ00], clearly impractical if k is the size of memory. A related approach is to bound the treewidth of a flow graph: graphs with bounded treewidth can be hierarchically decomposed in a way that again gives a linear-time maximum flow algorithm [HKNR98]. Unfortunately, the well-known algorithms that are efficient for fixed treewidth k all apparently have exponential dependencies on k that make them impractical for treewidths as small as 4 (e.g., the algorithm of Bodlaender [Bod93]). We suspect that the flow graphs produced by our tool have small treewidth, but we have not been able to verify this because even computing treewidth is very expensive in practice [Bod05]. However, a further specialization of this idea is within the realm of experiment, using the class of series-parallel graphs, whose treewidth is at most 2.

A series-parallel graph is one that can be formed using only the operations of series and parallel composition familiar from electrical circuits. The maximum flow in a series-parallel graph can be computed easily, since series and parallel composition correspond to the operations of minimum and addition on the maximum flows of the subgraphs.

Our flow graphs are not generally series-parallel, but they often contain large series-parallel portions, which suggests the use of a data structure called an SPQR tree [BT89]. An SPQR tree is a tree that represents a hierarchical decomposition of a directed acyclic graph with exactly one source and sink (an *s-t DAG*). The nodes in the tree are of four kinds labelled S, P, Q, or R: S nodes represent a series composition of their children, P nodes represent a parallel composition, Q nodes are leaves that represent single edges, and R nodes represent any composition that is not series-parallel. An SPQR tree can be constructed efficiently, and maintained incrementally as vertices and edges are added [BT89]. Depending on the structure of the graph, the tree can range between having no R nodes (for a series-parallel graph), and representing the entire graph by a single R node with all Q nodes attached directly.

If our flow graphs have SPQR trees without large R nodes, then the maximum flow can be computed quickly, since a super-linear general algorithm would only be needed inside R nodes. Moreover, the hierarchical nature of an SPQR tree allows for a convenient incremental flow algorithm: if an edge is added to the graph, then only the flows in the subtree corresponding to the two endpoints would need to be recomputed.

To test the efficacy of SPQR tree decomposition on our flow graphs, we computed SPQR trees for them using the batch algorithm from the AGD library [GJK+01]. (OGDF [CGJ+07], the successor library to AGD, also includes incremental SPQR tree construction and is open-source, but was not yet available when we began these experiments.) The results are shown in Figure 5-9 (for OpenSSH, the flow graph was too large for the SPQR tool to process; for ImageMagick we used a smaller image size). Series-parallel structure occurs across all the programs, as shown by the large number of S and P nodes. However, most of the trees also had a large R node at the root, indicating that the high-level structure of the flow is not series-parallel. Comparing the two runs of `bzip2`, notice that the R node at the root grows as a

Program	vertices	edges	S and P nodes	R nodes	largest R node
KBattleship	49135	57049	10583	8	5485
ImageMagick twist	1324145	1516765	286409	2639	151483
ImageMagick pixelate	225294	271464	43882	82	28714
ImageMagick blur	1898289	2354791	343095	2376	359440
OpenGroupware.org	550	647	21	2	45
X server	2010	2253	380	3	174
bzip2, 1KB input	1254073	1559800	196937	1011	197911
bzip2, 2KB input	2352727	2916519	362728	1993	373783

Figure 5-9: Experimental evaluation of SPQR trees for representing flow graphs. An SPQR tree is an efficient representation for maximum flow computation if the size of its largest R node (last column, measured in vertices) is small.

constant fraction of the graph size; if this is the general pattern, it means that an SPQR tree will not provide an asymptotic performance advantage for this program. Therefore, while SPQR trees capture some useful regularities, they do not appear sufficient to allow the technique to scale to very large graphs.

5.3.2 Efficient and potentially precise approaches

An alternative to exactly computing the maximum flow in large graphs is to simplify the graph in a way that makes it much smaller, while still being sound and not greatly increasing the maximum flow. The most important regularities in large graphs seem to come from loops in the original program, and are most easily exploited by using information about the program. Our tool does this using the same implementation of edge labelling and node collapsing that was described in Section 5.2: even the graph of a single run can be simplified by combining edges with the same context-sensitive code location, since the context does not distinguish different loop iterations. A graph can be collapsed even further by combining edges based on their code location (context-insensitive). With either variant, the size of the collapsed graph grows not with the runtime of the original execution, but with its code coverage; since the latter tends to plateau, much longer executions can be analyzed. A disadvantage of this collapsing technique is that it undoes some of the properties that make computations on the original graph easy: the summed capacities on edges can be unbounded, and collapsing can introduce cycles. For instance, collapsed graphs cannot be represented with SPQR trees.

To test the scalability of our graph construction and maximum-flow computations on large graphs, we ran our tool on `bzip2`, a general-purpose (lossless) compression tool based on block sorting. We used `bzip2` to compress inputs files marked as entirely secret. We do not intend `bzip2` as a realistic target for security analysis (obviously its output contains the same information as its input). We chose it because it represents a worst-case for our analysis's performance: it is computationally intensive, almost all of the computation operates on data derived from the input, and it makes extensive use of large arrays that necessitate the laziness described in Section 3.4. (Chapter 7 discusses larger, more security-relevant programs; for them the tool's overhead is less, because many operations are not connected to the secret data.) Also, it is easy to select inputs of various sizes, and the expected amount of information flow can be computed a priori to give a bound on the expected results. We chose a class of inputs that are highly compressible: the digits of π, written out in English words, as in "`three point one four one five nine`".

We ran our tool with context-sensitive edge collapsing, and `bzip2` in verbose mode `-vv` with a 100k block size. The computer was a 1.8GHz AMD Opteron 265 running Linux; `bzip2` and our tool ran in 32-bit mode.

Figure 5-10 compares the flow measured by our tool to the expected bound. That expected bound is the minimum of the size of the input, and the size of that portion of the output that depends on the input. The exact value for the latter is somewhat uncertain, because part of the output format consists of fixed headers, and the commentary printed to the terminal is only partially input-dependent; so we estimate it with lower and upper bounds (curved dotted lines in the figure). The results match our expectations: very small inputs cannot be compressed by `bzip2`, but for inputs that `bzip2` can compress, our tool's flow bound matches the size of the compressed output.

The running time of our tool grows linearly over this range of input sizes, thanks

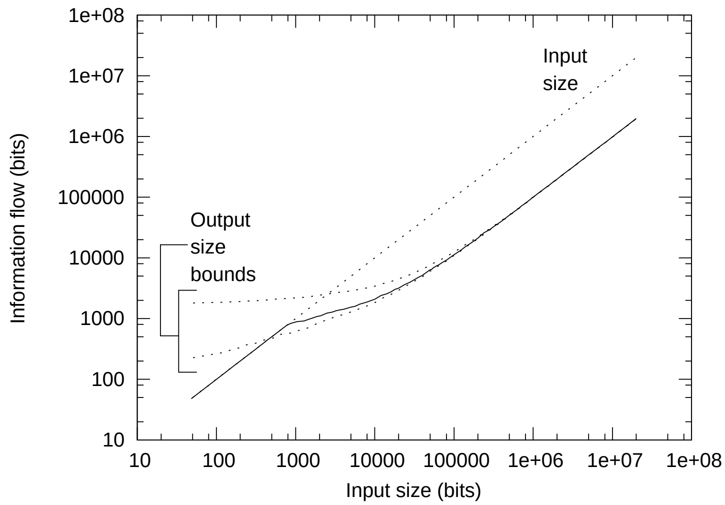

Figure 5-10: The amount of information revealed in compressing files with `bzip2`, as measured by our tool (note log-log scale). The solid line shows the flows measured by our tool, in bits. The dotted lines represent other functions that would be expected to bound the flow. The straight line through the origin represents the input size. The two curved lines (which are close to linear but do not pass through the origin) represent the size of the program's output, minus upper and lower approximations of the amount of output (such as fixed headers and progress messages) that does not depend on the input.

to the lazy range operation implementation and graph collapsing techniques, as shown in Figure 5-11. For the largest input, 2.5MB, the tool's running time was 1.5 hours. Though still quite slow compared to an uninstrumented execution, this time reflects processing a graph (before collapsing) with 3.6 billion nodes, since almost all of `bzip2`'s time is spent operating on secret data. (After collapsing, the graph had only about 22000 nodes and 30000 edges.) When tracing code that is not operating on secrets, no graph is constructed, so the tool's is overhead less, though still more than Memcheck's. The time to compute a maximum flow on the collapsed graph was less than a second in all cases.

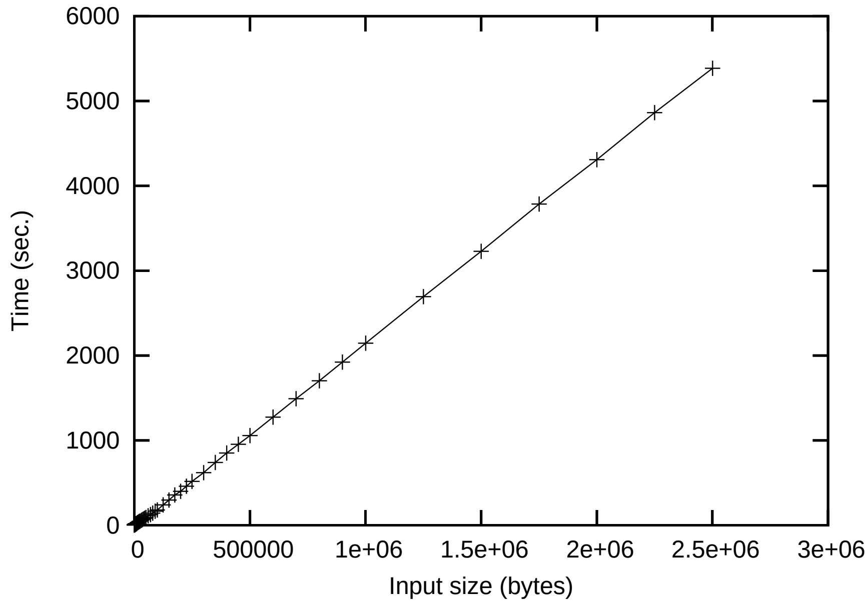

Figure 5-11: The running time of our analysis, with graph collapsing, on `bzip2` running on a range of input sizes. For a 2.5 megabyte file, the tool took about 1.5 hours. For small inputs, the performance is also linear, but dominated by a constant startup time of about 4 seconds.

5.4 Computing a minimum cut from a flow

If all that is required is a flow measurement for one or more executions, it is enough to calculate the amount of the maximum flow in a single or combined graph. In other cases, however, it is useful to know a minimum cut corresponding to the maximum flow, either to understand which program locations are information flow bottlenecks, or in order to use one of the more efficient checking techniques of Chapter 4 on future executions. Such a cut can be easily computed from the flow values for individual edges produced by the maximum flow algorithm.

An algorithm for computing a minimum cut from a maximum flow is implicit in the textbook proof of the max-flow-min-cut theorem [CLR90]. Our tool first enumerates the nodes on the source side of the cut by depth-first search: they are the nodes that are reachable from the source along an *augmenting path*, one in which each edge is either traversed in the forward direction and has excess capacity, or is traversed in the backward direction and has non-zero forward flow that can be cancelled. Then, the cut

```
/* Print all the "."s or "?"s,
   whichever is more common. */
void count_punct(char *buf) {
    unsigned char num_dot = 0, num_qm = 0, num;
    char common, *p;
    ENTER_ENCLOSE(num_dot, num_qm);
    while (p = buf; *p != '\0'; p++)
        if (*p == '.')
            num_dot++;
        else if (*p == '?')
            num_qm++;
    LEAVE_ENCLOSE();
    ENTER_ENCLOSE(common, num);
    if (num_dot > num_qm) {
        /* "."s were more common. */
        common = '.'; num = num_dot;
    } else {
        /* "?"s were more common. */
        common = '?'; num = num_qm;
    }
    LEAVE_ENCLOSE();
    /* print "num" copies of "common". */
    while (num--)
        printf("%c", common);
}
```

Figure 5-12: (Same as Figure 3-3.) C code to print all the occurrences of the most common punctuation character is a string. For instance, when run on its own source code, the program produces the output "........".

edges are those that connect nodes reached in the DFS to nodes not reached. Because the graph collapsing technique of Section 5.2 can introduce cycles, the computed maximum flow can contain extra flow circulating in cycles that do not include either the source or the sink. Though distracting to a human observer, these extra circular flows do not contribute to the total measured flow, or disrupt the cut construction algorithm.

5.5 Punctuation count example

Revisiting the punctuation counting example of Section 3.3 (repeated as Figure 5-

12), we see that using the graph maximum flow techniques of this chapter instead of a simple tainting analysis greatly increases our tool's precision. Recall that simple tainting finds that all the bits of the program's output might depend on the secret input: for instance, on an execution when the output is 8 characters, tainting measures it as revealing 64 bits of information. On the other hand, the maximum flow our tool computes for such an execution is 9 bits: intuitively, 1 bit telling for telling whether the most common punctuation character is a period or a question mark, and 8 bits counting the number of times the most common character appeared (modulo 256). This corresponds to a minimum cut with two edges: one for the implicit flow from the comparison between `num_dot` and `num_qm` on line 14 (capacity 1 bit), and one for the value of `num` after the second enclosure region on line 21 (capacity 8 bits).

Chapter 6

Formalization and soundness

Though the information flow measurement techniques described in previous chapters can be understood using just an intuitive notion of information content (and in most cases this was the way we originally developed them), it can be valuable to have a more precise statement of what the tool's results mean. This chapter formulates such a statement, first by specifying what it means for a dynamic information-flow analysis to give sound results for particular executions, and then giving a soundness proof for a core version of our technique, using a new simulation-based proof technique.

6.1 Soundness and codes

A key challenge in information-flow measurement is that information can be encoded in many different and incomparable ways. In particular, an attacker who is misusing a program can use its computations to encode information differently than the developers intended. Therefore we propose a definition of soundness that treats the program as a channel for transmitting messages, and bounds the way the information that can be sent using any encoding.

Specifically, we describe what it means for an analysis that examines a subset of possible executions to give an acceptable flow bound in two steps. First, we present a general attack model in which an adversary uses the program to communicate a secret message of her choosing to a confederate (Section 6.1.1). Second, we define a sound flow bound in this model: in summary, a bound of k bits is sound if an adversary could have communicated the same information by sending a k-bit message directly (Section 6.1.2). Throughout, we use the perspective of expressing information with a code that represents possible messages via bit strings of variable length.

6.1.1 A communications channel attack model

Rather than assuming that the secret to be protected is drawn from a fixed (e.g., uniform) distribution, we have found it more natural to consider a more powerful adversary who can choose the secret inputs to reveal as much information as possible. For instance, consider a division function for 32-bit words that hides its normal output, but has an observably different behavior on a divide by zero error. If an adversary could influence the divisor, she might cause it to be 0 with probability one half, in which case each execution would reveal one bit of information. This result does not depend on how likely the divisor is to be 0 during normal program execution, since not only is that difficult to know, the adversary may not be constrained to act normally.

In more detail, consider a pair of spies, Alice and Bob. Alice wants to use the program to send a message to Bob, by choosing the program's secret inputs to cause some change to the public outputs that Bob observes. Alice and Bob have prior knowledge of the program, and they have agreed in advance on a set of possible messages they might want to communicate. The public inputs might be out of Alice and Bob's control, or Alice and Bob might have chosen them, but we will treat them as being fixed in advance: the analysis's results and soundness will be with respect to a particular set of public inputs. We will also assume that Alice and Bob are interested in error-free communication (the program is deterministic), and have no computational limits, so their strategy is to choose a set of possible program inputs that Alice might send, each of which will cause a distinct public output. For instance, in the division example above, they might choose the following code: Alice gives the input `5/3`, causing normal program output, to convey "attack at dawn", while she gives `2/0`, causing an error report, to convey "no attack".

In essence, we treat the program's execution as a channel for transmitting messages, and are interested in an upper bound on the amount of information the channel can convey under any coding scheme: its *channel capacity*. The channel capacity is determined by the total number of different public outputs the program can produce, but counting them directly would be impractical. Instead, our tool's measurements correspond to a natural coding scheme suggested by the structure of the analyzed program, which will always be at least as much as the channel capacity over the observed program behaviors.

Other quantitative information-flow analyses have commonly treated the secret to be protected as being drawn from a uniform distribution. This perspective, in which one bit of input data always carries a full bit of information, has an intuitive appeal: for instance, in the division example, it is tempting to argue that finding out that a 32-bit input has all bits zero should always count as discovering 32 bits of information. However, we believe it is ultimately less useful because it is tied to a particular data representation, that of the inputs to the program being analyzed. By contrast, channel capacity abstracts away from a particular representation to more abstractly characterize the computation a program performs. In particular, channel capacity can be more naturally be approximated compositionally, as our graph-based analysis does.

6.1.2 Soundness of the channel model

To Alice and Bob, the originally intended behavior of the program might just be a distraction: they wish to use its input/output behavior as a communications channel. To define how well they can exploit the program, we can compare their results using it to what they could achieve by using a direct communications channel. Instead of an execution of the program that we would like to say reveals k bits, we imagine that Alice sends a string of k binary digits directly to Bob according to a code they have settled on in advance. For instance, `0` might correspond to "attack at dawn", and `1` to "no attack". Suppose that for each program input $i \in I$ that Alice sends to convey as message, a tool reports an information-flow bound $k(i)$. We define that result to be sound if there is also a code of bit strings by which Alice and Bob could have unambiguously communicated the same messages, in which each message was represented by a string of $k(i)$ bits. Thus, in the division example, it would be sound for the tool to report a bound of 1 bit.

There is also an equivalent characterization of soundness as a numeric condition on the amounts $k(i)$. Intuitively, it is impossible for a program to produce many distinct outputs, none of which reveal much information. The precise characterization of this relationship is Kraft's inequality [CT91], which in our notation states that $\sum_i 2^{-k(i)} \leq 1$. (Kraft's inequality holds for any uniquely-decodable code, and conversely, it is straightforward to construct a code to match a set of lengths that satisfy the inequality [CT91].) Two more specific consequences follow from this soundness definition. First, if a sound tool ever reports a flow of 0 bits, then it must be the case that the public output for that execution is the only one that can possibly be produced with any other secret inputs (for that public input). In other words, the case of 0 bits corresponds to the non-interference criterion for no information flow. Second, if there are N messages that all carry the same information, each one must be convey at least $\log_2 N$ bits: k bits are enough to distinguish between 2^k possibilities.

We define soundness with respect to a code, rather a single correct amount of information, because the same information can be coded in several incomparable ways. Our technique can be thought of as inferring one code from the structure of the program, but other equally good ones are possible. To illustrate this, consider an example that makes a three way choice, such as if the modification to the punctuation counting example shown in Figure 6-1. Because of the nested structure of the `if` statements, our tool reports that 1 bit is revealed if the most common punctuation character is a period, or 2 bits each if it is a question mark or exclamation point. But the fact that the period is checked first is arbitrary: one could equally well code `!` with one bit and `.` and `?` each with two; or if the three characters are equally likely, the information in each would be $\log_2 3$ (about 1.58). Since it is generally not possible to know the probabilities of alternate program inputs, we argue that it would not be sensible to try to define the correct amount of information revealed by choosing an exclamation mark to be 1 bit, 1.58 bits, 2 bits, or any other single value.

```
if (num_dot > num_qm && num_dot > num_excl) {
    /* "."s were most common. */
    common = '.';
} else {
    if (num_qm > num_dot && num_qm > num_excl) {
        /* "?"s were most common. */
        common = '?';
    } else {
        /* "!"s were most common. */
        common = '!';
    }
}
```

Figure 6-1: C code to determine which of three punctuation marks appeared most often in a string (as might be used in an extended version of the code in Figure 5-12). There are several different but sound ways of measuring the information revealed in such a three-way choice. Our technique uses one derived from the structure of the program. In this case, it would measure that one bit is revealed if the most common character is a period, while two bits are revealed if either a question mark or an exclamation mark is most common.

6.2 Soundness definition for a formal language

To make our notion of soundness more precise, we will next state it in terms of a completely-specified core subset of our real analysis. This core analysis includes the key techniques of Chapter 3, as well as a cut checked via tainting as described in Section 4.1. Because of the equivalence between maximum cuts and minimum flows, soundness in this model also carries over immediately to the explicitly graph-based analysis described in Chapter 5, as long as the cut location is fixed. We will return to the issue of varying cut locations in Section 6.4.

6.2.1 An unstructured imperative language

Because our real tool operates on programs at the machine language level, it is appropriately modelled by a technique for a language with unstructured control flow. Since the real technique counts the amount of information in machine words in terms of their bits, we restrict the language's variables to single bits.

The syntax of the language is summarized in Figure 6-2. The program's data is

$$
\begin{array}{lll}
\text{stmt} & ::= & \mathtt{x}_i\ \mathtt{=}\ \mathtt{x}_j \\
& & \mathtt{x}_i\ \mathtt{=}\ \mathtt{not}\ \mathtt{x}_j \\
& & \mathtt{x}_i\ \mathtt{=}\ \mathtt{x}_j\ \mathtt{and}\ \mathtt{x}_k \\
& & \mathtt{output}\ \mathtt{x}_i \\
& & \mathtt{if}\ \mathtt{x}_i\ \mathtt{then}\ \mathtt{goto}\ l \\
& & \mathtt{end} \\
& & \mathtt{leak}\ \mathtt{x}_i \\
& & \mathtt{enclose}(l, n, R) \\
& & \mathtt{end\_enclose}
\end{array}
$$

Figure 6-2: Syntax for a simple imperative language used in the soundness proof of Section 6.2.

stored in a finite set of variables $\mathtt{x}_i$ indexed by positive integers i. Each $\mathtt{x}_i$ may take the values `0` or `1`. A subset of the variables, the *secret input variables*, are initialized by the program's secret inputs; the remaining variables are initialized by the public inputs. Six types of statements provide the basic operations of assignment, logical negation and conjunction, output, conditional branches, and a statement denoting the end of execution. Three additional statement types correspond to program annotations in our real system: one to explicitly count the information in a variable as leaked, and two to enter and exit *enclosure regions*. A program consists of a sequence of statements numbered by labels l, which we take to be consecutive integers starting from 1. In an `enclose` statement, l is a label referring to the corresponding `end_enclose` statement, n is a positive integer giving the number of steps for which the enclosure region will execute, and R is a set of variables that we call the results of the enclosure region. There are three syntactic well-formedness constraints on programs: each label l must refer to an existing statement, the last statement must be `end`, and `enclose` and `end_enclose` statements must match up one-to-one according to the labels l (but their locations are not constrained).

Rather than defining the static well-formedness of an enclosure region in terms of possible program side effects, the model enforces enclosure regions dynamically (a capability that we also included in the real tool as a backstop in case an enclosure region is given incorrectly). The purpose of these enclosure regions is to isolate calculations that occur within the region from the rest of a program: their results are visible only via the result variables R. These enclosure regions are a dynamic mechanism to ensure that even though a computation may be implemented using branches and side-effects to arbitrary memory locations, it can be reasoned about as if it were a pure function. To achieve this, the contents of memory are recorded when the region is entered, and, except for R, restored on exit. (In the real system, a more efficient logging mechanism is used to the same effect.) To prevent other side-effects and termination of the enclosed code from being visible, the execution of the region must end at a pre-specified label, and `output` and `end` statements are disabled for the duration. To avoid problems of non-termination, the enclosure region can execute for at most n steps; to make the proof simpler, we enforce that it execute for exactly n steps, waiting at the `end_enclose` if necessary. (If timing were visible in the model, fixing the execution time of enclosure regions would also avoid a timing channel.) The intent is that entries to and exits from enclosure regions should match up during execution, but with unstructured control flow, this cannot easily be enforced ahead of time (this difficulty applies to the real system as well). Instead, the system simply lets an enclosure region continue until the matching end is seen. Also, in the real system it is convenient to allow enclosure regions to dynamically nest, but this does not increase expressiveness, so we omit it from the model for simplicity.

More formally, we refer to the behavior of a program (without the dynamic information-flow analysis) as the *regular semantics*, given as a state transition relation in Figure 6-3. The state of a program consists of a program counter pc holding the label of the next instruction to execute, a store S holding the values of variables, and a structure E holding information about the current enclosure region. E is either a distinguished value $\perp$ if execution is not in an enclosure region, or a tuple (l, k, R, S) where l is a label, k is a nonnegative integer, R is a set of variables, and S is a store. l is the label at which the enclosure region will end, k is a counter decremented once per step to control how long the region executes, R is the set of result variables, and S is a saved copy of (non-result) program variables to restore at the end of the region. The notation $S[i \leftarrow b]$ represents a new store in which i is bound to b, and all other variables have the same bindings as in S; we also abuse notation by using sets as

$(pc, S, E) \rightarrow (pc', S', E')$ where:

1. $pc' = l$ if code(pc) = "$\texttt{if x}_i\ \texttt{then goto}\ l$" and $S(i) = \texttt{1}$
2. $pc' = pc + 1$ if code(pc) = "$\texttt{if x}_i\ \texttt{then goto}\ l$" and $S(i) = \texttt{0}$
3. $pc' = pc$ if $pc = E.l$ and $E.k > 1$
4. $pc' = E.l$ if $E.k = 1$
5. $pc' = E.l$ if $E \neq \perp$ and code(pc) = "$\texttt{end}$"
6. $pc' = pc + 1$ otherwise
7. $E' = \perp$ if $E.k = 0$ and $E.l = pc$
8. $E' = \perp$ if code(pc) $\neq$ "$\texttt{enclose}(l, n, R)$" and $E = \perp$
9. $E' = (l, n, R, S)$ if code(pc) = "$\texttt{enclose}(l, n, R)$" and $E = \perp$
10. $E' = (E.l, E.k - 1, E.R, E.S)$ otherwise
11. $S' = S[i \leftarrow S(j)]$ if code(pc) = "$\texttt{x}_i\ \texttt{=}\ \texttt{x}_j$"
12. $S' = S[i \leftarrow \neg S(j)]$ if code(pc) = "$\texttt{x}_i\ \texttt{= not x}_j$"
13. $S' = S[i \leftarrow S(j) \wedge S(k)]$ if code(pc) = "$\texttt{x}_i\ \texttt{= x}_j\ \texttt{and x}_k$"
14. $S' = S[\overline{E.R} \leftarrow E.S(\overline{E.R})]$ if $E.k = 0$ and $E.l = pc$
15. $S' = S$ otherwise
16. output $S(i)$ if code(pc) = "$\texttt{output x}_i$" and $E = \perp$
17. stop if code(pc) = "$\texttt{end}$" and $E = \perp$

Figure 6-3: Regular semantics for the language given in Section 6.2.1. For brevity, we follow the convention that any condition involving a field of E is false if $E = \perp$.

indexes and/or values to indicate that each i is bound to the corresponding or only value from b. Initially, pc points to the first statement, $E = \perp$, and S is populated with the program inputs.

After most statements, execution continues with the following statement (rule 6 in Figure 6-3), but an `if` statement causes a branch if its condition is true (2), and an enclosure region jumps to its end if the counter $E.k$ runs out (4) or an `end` is encountered (5). E is initialized at the beginning of an enclosure region (9) and cleared at the end (7, 8); when it is present, $E.k$ is decremented on each step (10). The assignment statements modify the store in the expected way (11–13); `end_enclose` also restores the contents of all variables except the results of the region (14). The `output` and `end` statements have the expected effects, but they are disabled inside enclosure regions (16, 17).

6.2.2 Analysis semantics

Recall, as discussed in Sections 3.1 and 4.1, that in measuring the information flow though a program with a fixed cut, our tool counts the potential propagation of secret information in two ways. A bit may be marked as secret (tainted) if it might contain secret information, and a counter keeps track of other leaks outside the set of variables. The secrecy status of bits is propagated conservatively: a copy of a secret bit is secret, and the output of an operation is secret if any of the inputs that contributed to it were. (But the result of an operation on a public value and a secret value can be public if the result does not depend on the secret value: for instance, multiplying a public `0` by a secret number yields a public `0`.) Preemptive leakage via a `leak` statement erases the secrecy of a bit and simultaneously increments the counter to compensate. To account for implicit flows, a bit is also counted as leaked in the same way if a secret bit is used as a branch condition, and of course secret bits are counted as leaked if they are output. Branches on secret data are not counted as leaks inside an enclosure region, but to compensate, the output of an enclosure region is always marked as secret.

We formalize the operation of the analysis with an *instrumented semantics* that extends the regular one, as shown in Figure 6-4. To the state of the system, we add secrecy store SS parallel to the regular store S, holding `1` if the corresponding variable is secret and `0` otherwise, and an integer counter c. We also add a saved secrecy store $E.SS$ to the enclosure-related information. Initially, $S(i)$ is `1` for the secret input variables and `0` for the others, and c is zero. The most complex rule (3 in Figure 6-4) describes the result of an `and` operation: the result is secret if either input is, and neither input is a public zero. Observe that because the instrumentation semantics are a pure addition to the regular semantics, the behavior of an instrumented program is the same as the behavior without analysis.

The value of c at the end of execution is the formal analysis's measurement of the amount of information that has been revealed on that execution. If c bits are enough to learn whatever the attacker might learn from the program's public outputs, then

$(pc, E, S, SS, c) \to (pc', E', S', SS', c')$ where:

1. $SS' = SS[i \leftarrow SS(j)]$ if $\text{code}(pc) =$ “$\texttt{x}_i\ \texttt{=}\ \texttt{x}_j$”
2. $SS' = SS[i \leftarrow SS(j)]$ if $\text{code}(pc) =$ “$\texttt{x}_i\ \texttt{= not}\ \texttt{x}_j$”
3. $SS' = SS[i \leftarrow (SS(j) \vee SS(k)) \wedge (S(j) \vee SS(j)) \wedge (S(k) \vee SS(k))]$ if $\text{code}(pc) =$ “$\texttt{x}_i\ \texttt{=}\ \texttt{x}_j\ \texttt{and}\ \texttt{x}_k$”
4. $SS' = SS[i \leftarrow \mathtt{0}]$ if $E = \perp$ and $\text{code}(pc) =$ “$\texttt{output}\ \texttt{x}_i$”
5. $SS' = SS[i \leftarrow \mathtt{0}]$ if $E = \perp$ and $\text{code}(pc) =$ “$\texttt{if}\ \texttt{x}_i\ \texttt{then goto}\ l$”
6. $SS' = SS[i \leftarrow \mathtt{0}]$ if $E = \perp$ and $\text{code}(pc) =$ “$\texttt{leak}\ \texttt{x}_i$”
7. $SS' = E.SS[E.R \leftarrow \mathtt{1}]$ if $E.k = 0$ and $E.l = pc$
8. $SS' = SS$ otherwise
9. $c' = c + 1$ if $E = \perp$ and $SS(i)$ and $\text{code}(pc) =$ “$\texttt{output}\ \texttt{x}_i$”
10. $c' = c + 1$ if $E = \perp$ and $SS(i)$ and $\text{code}(pc) =$ “$\texttt{if}\ \texttt{x}_i\ \texttt{then goto}\ l$”
11. $c' = c + 1$ if $E = \perp$ and $SS(i)$ and $\text{code}(pc) =$ “$\texttt{leak}\ \texttt{x}_i$”
12. $c' = c$ otherwise
13. $E'.SS = SS$ if $\text{code}(pc) =$ “$\texttt{enclose}(l, n, R)$” and $E = \perp$
14. $E'.SS = E.SS$ otherwise

Figure 6-4: Instrumented semantics describing secrecy tracking for the core language, as described in Section 6.2.2. The rules for pc, E, and S are the same as in Figure 6-3.

c is a sound information-flow measurement. If we could define a single number I that represented a perfect information-flow measurement, soundness for the analysis would amount to a simple inequality $c \geq I$, but as argued in Section 6.1, no single such value I would be appropriate. Instead, we compare the measurement c to the size of a coded message conveying the same information. In the next section, we will describe how to obtain such a result: in fact we will describe a way to use the program itself to encode and decode a c-bit message that suffices (along with the public inputs) to reconstruct the program's output.

6.3 A simulation-based soundness proof technique

In this section, we first describe the construction of a simulation of the instrumented program by a pair of programs (Section 6.3.1), then prove by induction that the simulation is a faithful one (Section 6.3.2). This result then implies (Section 6.3.3) that the analysis meets the definition of soundness we introduced in Section 6.2: the information conveyed by a program execution could also be conveyed by a coded binary message with as many bits as the flow bound the analysis reports.

6.3.1 Simulation construction

Given an instrumented program execution under the semantics described in Section 6.2.2, we wish to prove that the counter c is an upper bound on the size of a coded message needed to convey the same information about the secret inputs that is present in the program's output. To do this, we connect two copies of the instrumented program by a unidirectional channel called a *pipe*; we call the two copies the *writer* and the *reader* according to the way they use the pipe. The two copies have the same program text and public inputs, but initially only the writer has the secret input data. The two copies execute in lockstep; on some steps, the writer writes a bit to the pipe, and the reader reads it. The goal is that the two copies of the program should produce the same results; this demonstrates that the information sent via the pipe is the only potentially-secret information needed to produce the program output.

In order for the reader to simulate the writer, the reader needs access to secret data whenever it affects control flow, or is output. In fact, we choose to have the writer send a secret bit exactly whenever it is counted as leaked in the instrumented program. The writer and the reader both maintain the same secrecy bits as the instrumented program, which tell the reader when to use a value from the pipe. Enclosure regions can make decisions based on secret bits without leaking them, so the reader is unable to simulate them; instead, it simply waits for them to complete.

The writer semantics are purely an addition to the instrumented semantics, just as the instrumented semantics added to the regular semantics, so the writer's behavior is the same. By contrast, the reader's semantics are different; proving that the reader's behavior is similar is the main task of the proof below. The modified semantic rules for the writer and reader are given in Figure 6-5, where the pipe operations are represented as write($\mathtt{x}_i$) and read(). (When read() appears in the definitions of two post-state variables for a single state transition, the intended meaning is that a single bit is read, and used in multiple places.) As mentioned earlier, the reader does not start with any secret information. It does not matter what its copies of the secret

Writer:
$(pc, E, S, SS, c) \to (pc', E', S', SS', c')$ where:
1. write($\mathtt{x}_i$) if $E = \bot$ and $SS(i)$ and code(pc) = "$\mathtt{output\ x}_i$"
2. write($\mathtt{x}_i$) if $E = \bot$ and $SS(i)$ and code(pc) = "$\mathtt{if\ x}_i\ \mathtt{then\ goto}\ l$"
3. write($\mathtt{x}_i$) if $E = \bot$ and $SS(i)$ and code(pc) = "$\mathtt{leak\ x}_i$"

Reader:
$(pc, E, S, SS, c) \to (pc', E', S', SS', c')$ where:
4. output read() if $SS(i)$ and code(pc) = "$\mathtt{output\ x}_i$" and $E = \bot$ (*)
5. output $\mathtt{x}_i$ if $\neg SS(i)$ and code(pc) = "$\mathtt{output\ x}_i$" and $E = \bot$
6. $pc' = l$ if code(pc) = "$\mathtt{enclose}(l, n, R)$"
7. $pc' = l$ if $SS(i)$ and code(pc) = "$\mathtt{if\ x}_i\ \mathtt{then\ goto}\ l$" and read() = $\mathtt{1}$ (*)
8. $pc' = l$ if $\neg SS(i)$ and code(pc) = "$\mathtt{if\ x}_i\ \mathtt{then\ goto}\ l$" and $\mathtt{x}_i = \mathtt{1}$
9. $pc' = pc + 1$ if $SS(i)$ and code(pc) = "$\mathtt{if\ x}_i\ \mathtt{then\ goto}\ l$" and read() = $\mathtt{0}$ (*)
10. $pc' = pc + 1$ if $\neg SS(i)$ and code(pc) = "$\mathtt{if\ x}_i\ \mathtt{then\ goto}\ l$" and $\mathtt{x}_i = \mathtt{0}$
11. $S' = S[i \leftarrow \text{read}()]$ if $SS(i)$ and code(pc) = "$\mathtt{output\ x}_i$"
12. $S' = S[i \leftarrow \text{read}()]$ if $SS(i)$ and code(pc) = "$\mathtt{if\ x}_i\ \mathtt{then\ goto}\ l$"
13. $S' = S[i \leftarrow \text{read}()]$ if $SS(i)$ and code(pc) = "$\mathtt{leak\ x}_i$"

Figure 6-5: Modified semantics for the pipe writer and reader, as described in Section 6.3. The writer rules are in addition to those given in Figures 6-3 and 6-4. The reader rules also extend those, except that the three rules marked (*) replace the corresponding ones from Figure 6-3; the reader rules add the condition $SS(i)$ and use read() in place of $\mathtt{x}_i$. Because the reader skips directly to the end of enclosure regions, the effect is as if each of the new reader rules included the condition $E = \bot$, but we omit it for space.

input variables are initialized to, but for concreteness, say they all start as `0`.

6.3.2 Simulation lemma

Section 6.3.1 described the construction of a pair of programs intended to give the same results as an instrumented secret-using program, but with the use of secret data by the second (reader) program rationed by a special channel. To use this construction to obtain a soundness result for the analysis, we must first prove that the reader faithfully simulates the instrumented program, and then relate the information disclosed by the reader to the leakage count maintained by the instrumented program.

To relate the behavior of the writer and the reader, we define a relation $\sim$ between states of the writer and states of the reader. The definition captures the intuition that the writer and reader should generally run in lockstep, but that the contents of secret store locations may be different in the reader, and the correspondence is broken while executing enclosure regions. For convenience, we use subscripts of W and R to distinguish the state variables of the writer and reader. Two states are related by $\sim$ if all the following hold:

- The program counters are the same: $pc_{\mathrm{W}} = pc_{\mathrm{R}}$
- The secrecy bits are the same: $\forall i, SS_{\mathrm{W}}(i) = SS_{\mathrm{R}}(i)$
- The store contents that are public are the same: $\forall i, \neg SS_{\mathrm{W}}(i) \Rightarrow S_{\mathrm{W}}(i) = S_{\mathrm{R}}(i)$
- The writer is not in an enclosure region: $E_{\mathrm{W}} = \bot$

Given this definition of $\sim$, the key simulation lemma states that each writer state for which $E_{\mathrm{W}} = \bot$ is related to the corresponding (simultaneous) reader state by $\sim$. We prove this by induction over the execution history of the programs. Clearly the initial states are related by $\sim$: the program counters are both 1, both programs are outside enclosure regions, and except for the reader's missing secret bits (for which $SS_{\mathrm{R}}(i) = SS_{\mathrm{W}}(i) = \mathtt{1}$), their initial store contents are the same.

For the inductive step, suppose that the current states are related by $\sim$, and let primed state variables represent the next states. (Note we can omit the subscripts on pc, SS, and E without ambiguity.) We take one case for each of the potential next statement types:

- $\mathtt{x}_i$ `=` $\mathtt{x}_j$: Only the value stored at location i is modified, so we must check that it is either the same or secret in both post-states. If $SS(j) = 0$, then $S_{\mathrm{W}}(j) = S_{\mathrm{R}}(j)$, and so $S'_{\mathrm{W}}(i) = S'_{\mathrm{R}}(i)$. On the other hand if $SS(j) = 1$, then $SS'_{\mathrm{W}}(i) = SS'_{\mathrm{R}}(i) = \mathtt{1}$.
- $\mathtt{x}_i$ `= not` $\mathtt{x}_j$: Similarly, if $SS(j) = 0$, then $S_{\mathrm{W}}(j) = S_{\mathrm{R}}(j)$, and so $S'_{\mathrm{W}}(i) = \neg S_{\mathrm{W}}(j) = \neg S_{\mathrm{R}}(j) = S'_{\mathrm{R}}(i)$. On the other hand if $SS(j) = 1$, then $SS'_{\mathrm{W}}(i) = SS'_{\mathrm{R}}(i) = \mathtt{1}$.

- $\texttt{x}_i$ `=` $\texttt{x}_j$ `and` $\texttt{x}_k$: Here there are three kinds of cases. If both values are public, $SS(j) = SS(k) = \texttt{0}$, then both arguments are the same by assumption, $S_{\mathrm{W}}(j) = S_{\mathrm{R}}(j)$ and $S_{\mathrm{W}}(k) = S_{\mathrm{R}}(k)$, so the results are also the same: $S'_{\mathrm{W}}(i) = S_{\mathrm{W}}(j) \wedge S_{\mathrm{W}}(k) = S_{\mathrm{R}}(j) \wedge S_{\mathrm{R}}(k) = S'_{\mathrm{R}}(i)$. If either argument is public and zero, say $SS(j) = 0$ and $S_{\mathrm{W}}(j) = S_{\mathrm{R}}(j) = 0$, then both results must be zero, and so equal: $S'_{\mathrm{W}}(i) = \texttt{0} \wedge S_{\mathrm{W}}(k) = \texttt{0} = \texttt{0} \wedge S_{\mathrm{R}}(k) = S'_{\mathrm{R}}(i)$. Otherwise, at least one argument is secret, and neither argument is both public and zero, so all three of the conjuncts in the rule for SS' are true, and $SS'_{\mathrm{W}}(i) = SS'_{\mathrm{R}}(i) = \texttt{1}$.

- `output` $\texttt{x}_i$: If $SS(i) = \texttt{0}$, then the state is unchanged. Otherwise, note that $E = \perp$, so the writer writes a bit b which is read by the reader. $SS'_{\mathrm{W}}(i) = SS'_{\mathrm{R}}(i) = 0$, but $S'_{\mathrm{W}}(i) = S'_{\mathrm{R}}(i) = b$. Also, observe that the writer and reader output the same bit in either case.

- `if` $\texttt{x}_i$ `then goto` l: As in the `output` case, the branch condition is either the same by assumption if it is public, or if it is secret, the same because it is written by the writer and read by the reader. Thus, either $pc'_{\mathrm{W}} = l = pc'_{\mathrm{R}}$ if the branch is taken, or $pc'_{\mathrm{W}} = pc + 1 = pc'_{\mathrm{R}}$ if not.

- `end`: Note that $E = \perp$, so both programs stop and this case is satisfied vacuously.

- `leak` $\texttt{x}_i$: Also like the `output` case, if $SS(i) = \texttt{0}$, then the state is unchanged. Otherwise, $E = \perp$, so the writer writes a bit b which is read by the reader, and $S'_{\mathrm{W}}(i) = S'_{\mathrm{R}}(i) = b$.

- $\texttt{enclose}(l, n, R)$: Uniquely in this case, it is not the next states that are related by $\sim$, but the next non-enclosed states, $n+1$ steps later. Because `end` is disabled in an enclosure region, there are sure to be such subsequent states: when the countdown $E.k$ reaches zero, control will have reached the `end_enclose` at l, so the next states have pc referring to the next statement after that. Using primes for this state, we clearly have $E'_{\mathrm{W}} = E'_{\mathrm{R}} = \perp$. The secrecy store in this state consists of the saved secrecy store SS, with all of the locations in R marked as secret, but R is the same for both programs, so $SS'_{\mathrm{W}} = SS'_{\mathrm{R}}$. For the regular store, locations not in R were saved and restored, so match the values in S_{W} and S_{R}, which are either equal or secret by the induction hypothesis. Values in R are marked as secret, so may be different, but $\sim$ holds.

- `end_enclose`: The usual situation of the end of an enclosure region was described in the previous case. Here, $E = \perp$; if an `end_enclose` is encountered outside an enclosure region, it has no effect.

This completes the induction. As a corollary, observe that writes to and reads from the pipe are always made on the same step by both programs, so there are never any left-over bits or blocking. Also, on each `output` statement, the bits output by the two programs are the same.

6.3.3 Using the simulation

If we take the intuitive view of secret information as a kind of substance that can be transformed but not created by computation, the simulation property leads immediately to a bound on the amount of information in the program output. We simply look at the reader process as a unit, and observe that the only secret information entering it is via the pipe; thus at most the same amount of information can come out in the output, which by construction is the same as the output of the original program. (The intuition that processing cannot create information is formalized as the Data Processing Theorem of information theory [McE02].)

The same conclusion holds in the more rigorous model of Alice, Bob, and their secret messages. The c bits of information that traverse the pipe in fact exactly give a binary-coded message, and using the reader as a decoder shows that the message contains any information in the program's output, since the exact program output can be recovered from it using only public information.

6.4 Consistency for multiple cut locations

```
1  int ident(int i) {
2      i &= 3;
3
4      x1 = i;
5      x2 = 0;
6      ENTER_ENCLOSE(x1, x2);
7      while (x1 > 0) { x1--; x2++; }
8      LEAVE_ENCLOSE();
9
10     x2 = (x2 + 1) & 3;
11     x3 = 0;
12     ENTER_ENCLOSE(x2, x3);
13     while (x2 > 0) { x2--; x3++; }
14     LEAVE_ENCLOSE();
15
16     x3 = (x3 + 1) & 3;
17     x4 = 0;
18     ENTER_ENCLOSE(x3, x4);
19     while (x3 > 0) { x3--; x4++; }
20     LEAVE_ENCLOSE();
21
22     x4 = (x4 + 1) & 3;
23     x5 = 0;
24     ENTER_ENCLOSE(x4, x5);
25     while (x4 > 0) { x4--; x5++; }
26     LEAVE_ENCLOSE();
27
28     return (x5 + 1) & 3;
29 }
```

Figure 6-6: An artificial example that demonstrates a particularly bad case of the unsoundness that is caused by choosing a cut location dependent on the secret input. This code is an unusual implementation of the identity function on integers between 0 and 3, so we would expect it to reveal 2 bits of information; and that is the value corresponding to many cut locations. But a cut location at implicit flows in the first `while` loop on line 7 would measure the flow according to the number of times the test executes: 1 bit for 0, 2 bits for 1, 3 bits for 2, and 4 bits for 3. The remaining loops are similar, but operate on $(i + 1) \bmod 4$, $(i + 2) \bmod 4$, or $(i + 3) \bmod 4$ for input i. Thus for any value between 0 and 3, there is a loop such that if the cut is chosen at that loop, the flow is measured as 1 bit.

As explained in Section 6.2, soundness is best defined as a property about sets of inputs, even if the tool examines only a single execution. But if a tool analyzes a set of executions, soundness requires that the results taken together correspond to a single possible code. As described so far, the maximum flow values our technique produces would only be guaranteed to be sound in this sense if the minimum cut always occurred at the same place in the flow graph. (In the simulation proof, this can be seen in the fact that the location of the preemptive leakage annotations is required to be known by both writer and reader: if the writer chose a different cut for different secret inputs, the reader would not know the right way to decode the pipe bits.)

For instance, consider the final phase (lines 22–24) of the example program of Figure 5-12, in which a character is printed n times ($0 \leq n \leq 255$). If the analysis chooses a cut before the loop, n will be measured in its binary representation, and so will be counted as revealing 8 bits. Alternatively, if it chooses a cut at the implicit flow edges corresponding to each loop test, then printing n characters will be counted as revealing $n+1$ bits. Either of these choices is sound on its own (they correspond to binary and unary encodings of n), but always choosing the smaller one (i.e., $\min(8, n+1)$) gives measurements that are too small. Kraft's inequality confirms this unsoundness: $\sum_{n=0}^{255} 2^{-\min(8,n+1)} = \frac{503}{256} > 1$. Another even more stark example of this problem is shown in Figure 6-6.

These examples show that if our maximum-flow analysis is run independently on different executions of a program, the results may be inconsistent with each other: some of the variation between the executions may cause the tool to pick different cut locations, rather than contributing to the estimated information flow. To get sound results from multiple executions, our tool combines the graphs from multiple executions and analyzes them together using the algorithm of Section 5.2. (Another possibility would be to count the choice of a cut itself as leaking some information, but this seems difficult to do precisely because there are exponentially many possible cuts, even though usually only a few are ever selected.) When flow graphs are combined, any sum of possible flows in the original graphs is possible in the combined graph, so a bound computed for the combined graph is still sound. On the other hand, the possible cuts in the combined graph correspond only to sets of cuts that appear in the same places in each original graph, excluding the possibility of lower flow bounds corresponding to inconsistently placed cuts.

Chapter 7

Case studies

A key aim of this research was to be able to measure information flows in real software. To this end, we used our implementation to test a different confidentiality policy in each of six open-source applications. The programs and the secret information protected are summarized in Figure 7-1. In each program the secret information participates in implicit flows, and is partially disclosed in ways that are nonetheless acceptable; thus both a quantified policy and a sound treatment of implicit flows are needed.

Playing the role of a developers interested in whether their software adequately controlled secret information, we used our tool in each case to measure the amount of information the program revealed on an execution of a typical use case. Then using our understanding of the intended use of the system, and the location of a cut in the program where the tool measured the flow, we reasoned about whether the measured information flow was legal to allow. In some cases, an excessive flow indicated a bug or a questionable design choice, so we investigated changes to the code to reduce the flow. In other cases, an excessive flow indicated that an operation, though implemented as intended, was not appropriate for use with secret data. In the remaining cases, our results verify that the programs satisfy what we believe to be a reasonable confidentiality policy, on the executions we examined.

To start, Section 7.1 covers some additional implementation details of our tool not directly related to any of the previously discussed main techniques. Next, one

Program	KLOC	# of libraries	secret data
KBattleship	6.6	37	ship locations
eVACS	9.3	7	authentication barcode
OpenSSH client	65	13	authentication key
ImageMagick	290	20	original image details
OpenGroupware.org	550	34	schedule details
X server	440	11	displayed text

Figure 7-1: Summary of the programs examined in the case studies of Chapter 7. The program sizes, measured in thousands of lines of code (KLOC), include blank lines and comments, but do not include binary libraries (3rd column, measured with `ldd`) that were included in the analysis but not directly involved with the security policy.

section will describe each of the six case-studies in turn.

To obtain precise results, all of the programs required enclosure region annotations. Section 7.8 describes a pilot experiment with a very simple static analysis for C which was able to infer a majority of the annotations used, and discusses how to improve its results by adding other standard techniques. We supplied the remaining enclosure annotations by hand: we found the locations where they were needed by running the tool in a mode in which every implicit flow operation caused a warning message. Because of limitations in our current syntax for specifying such regions, this sometimes required minor local code refactorings, such as introducing a temporary variable to hold a return value. Writing annotations was easy: we spent about as much time writing such annotations as compiling and configuring the programs to run on our system and developing test cases for the relevant policies.

7.1 Implementation details

Because the analysis operates at the binary level, all of the libraries that a program uses are included automatically. It would be possible to treat `malloc` as part of the instrumented program, though we currently inherit Memcheck's behavior of replacing the program's allocator. Doing so leaves the possibility of information flow via the addresses returned from `malloc`; this channel could be blocked by using a separate arena for allocations inside enclosure regions, or by randomizing the addresses.

Inputs and outputs are recognized based on system calls, such as `read` and `write` respectively. Memory-mapped I/O is not recognized, though doing so would not be difficult because every memory operation is already instrumented.

We have not studied the best extension of our technique to multi-threaded programs, since Valgrind implicitly serializes the programs it executes; it would likely suffice to execute enclosure regions atomically. (The case studies of this chapter are all single-threaded.)

Many aspects of a program's interactions with its environment might reveal information about its internals, such as how long it takes to execute or how much power the CPU draws. If such *side channels* are reflected in the program's output, they can be included in our approach: for instance, the result of `gettimeofday` could be treated as secret. However, observations made outside the program are beyond this scope of our technique.

7.2 KBattleship

In the children's game Battleship, successful play requires keeping secrets from one's opponent. Each player secretly chooses locations for four rectangular ships on a grid representing the ocean, and then the players take turns firing shots at locations on the other player's board. The player is notified whether each shot is a hit or a miss, and if a hit has sunk a complete ship. A player wins by shooting all of the squares of all of the opponent's ships. In a networked version of this game, one would like to know how much information about the layout of one's board is revealed in the network messages to the other player. If the program is written securely, each missed shot by the opponent should reveal only one bit, since "hit" and "miss" represent only two possibilities. KBattleship is an implementation of the game that is part of the KDE graphical desktop. We used our tool to measure how much information about the player's ship locations is revealed when playing KBattleship.

We were inspired to try this example because Jif, a statically information-flow secure Java dialect (the latest descendant of the work described in [Mye99]) includes as an example a 500-line Battleship game. Apparently unlike Jif Battleship, however, the version of KBattleship we examined (3.3.2) contains an information leak bug. In responding to an opponent's shot, a routine calls a method named `shipTypeAt` to check whether a board location is occupied, and returns the integer return value in the network reply to the opponent. However, as the name suggests, this return value indicates not only whether the location is occupied, but the type (length) of the ship occupying it. An opponent with a modified game program could use this fact to infer additional information about the state of adjacent board locations. The KBattleship developers agreed with our judgement that this previously unrecognized leakage constituted a bug, and our patch for it appears in version 3.5.3. Though this bug shows up as excessive flow under our tool, we discovered it by inspection while considering whether to use the program as a case study (before the tool was implemented).

Our tool can verify that the bug is eliminated in a patched version: we mark the position and orientation of each of the player's ships as secret, and measure how much of this information reaches the network. In response to a miss, the program reports one bit of information; a non-fatal hit reveals two bits, one indicating the shot is a hit and a second indicating it is non-fatal. These flows can be observed in real time by running our tool in a mode that recomputes the flow on every program output, or each second, whichever is less frequent. Information about the ship locations is also revealed via the program's graphical interface, but we excluded that code from the analysis by explicitly declassifying some data passed to drawing routines; thus this analysis could miss leaks that occurred through the GUI libraries.

7.3 eVACS

The eVACS system is a client-server implementation of electronic voting developed for use in elections in the Australian Capital Territory starting in 2001. The client software that runs on each voting terminal is a graphical application using a small keypad; voters are authenticated using a unique barcode that they scan at the beginning and end of the voting session. If an attacker gained access to a valid barcode, it could be used to cast a false vote, and if the association between a barcode and a voter's identity were known, it could be used together with information on the server to determine the voter's vote. If the client program is working correctly, the only uses of the authentication data are to verify it with an internal checksum and to send it to the server on two occasions. We used our tool to measure how much information derived from the barcode is revealed, and to where.

An eVACS barcode consists of 128 bits of data and a 4-bit checksum, encoded in 22 ASCII characters. We modified the client so it can run by itself on a regular Linux workstation (for instance, the modified version simulates the scanning of a barcode in response to a key press) and marked the barcode data as tainted.

For this application, it was more important to have precision in the assessment of where information leaked, and less important to measure the amount precisely, so we used only the basic tainting and enclosure techniques of Chapter 3, and not the minimum-cut or maximum-flow based techniques of Chapters 4 and 5. We let the barcode data propagate taint through the program, so that we could count separately all the ways in which it is used (though this leads to a larger total bound). In total, our tool reports that at most 403 bits of barcode-derived data are revealed. Specifically, our tool counts the 176 bytes of the barcode in each of two network socket writes, one in an authentication request to the server at the beginning of the voting process, and one along with the votes and other information in the final commit message.

Our tool also shows a few other disclosures of barcode-related information. Each time the barcode is read, a computed checksum is compared to the one encoded at the end of the barcode and, similarly, the two scanned barcodes are verified to be equal; each of these comparisons reveals one bit. Though the barcode is of a fixed length, it is represented as a null-terminated string in the code used to send network messages using HTTP. The lengths of these messages reveal the length of the barcode; we use a preemptive annotation to leak this value. In sum, this analysis of the results indicates that all of the ways eVACS might reveal barcode information on the observed executions are acceptable: the two network writes of the barcode match the intended protocol, and the additional flows are small and appropriate.

7.4 OpenSSH client

OpenSSH is the most commonly used remote-login application on Unix systems. In one of the authentication modes supported by the protocol, an SSH client program proves to a remote server the identity of the host on which it is running using a machine-specific RSA key pair. For this mode to be used, the SSH client program must be trusted to use but not leak the private key, since if it is revealed to the network or even to a user on the host where the client is running, it would allow others to impersonate the host. (We were inspired to consider this example by the discussion of it by Smith and Thober [ST06].) We used our tool to measure how much information about the private key is revealed by a client execution using this authentication mode, by marking the private key (a number of arbitrary-precision integers) as secret as it is read from a file.

Our tool finds that 128 bits of information about the secret key are revealed. The cut location reveals that this is the MD5 checksum of a response that includes a value decrypted with the public key, as expected under the protocol. Of course, our tool is not able to verify that MD5 is a secure one-way function, though that belief is part of why revealing those particular 128 bits is acceptable. Our tool demonstrates that if the 218-line MD5 implementation is secure, the entire execution obeys the confidentiality property: no information leaks from the rest of the program.

7.5 ImageMagick

ImageMagick is a suite of programs for converting and transforming bitmap images. We evaluated some of its transformations to assess how much information about the original they preserve. For instance, if one tries to anonymize a photograph by obscuring the subject's face, using a transformation that preserves very little

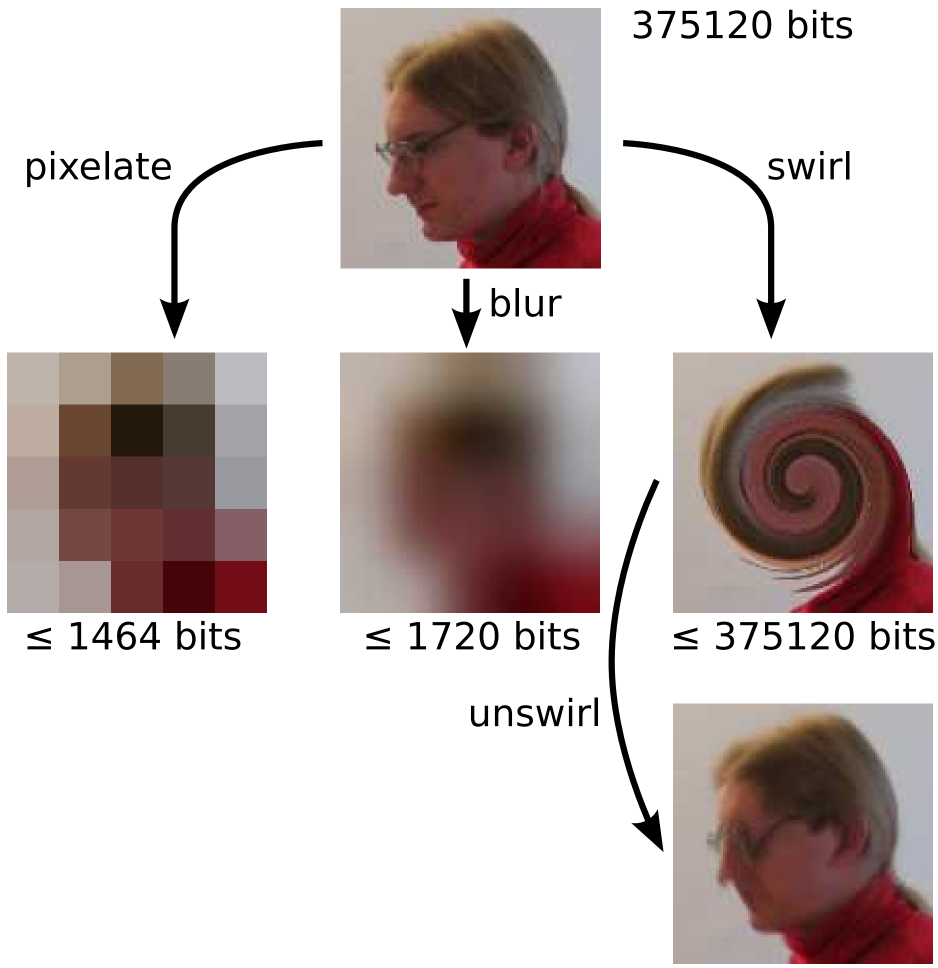

Figure 7-2: Image transformations vary in how much information they preserve. Our tool verifies that pixelating (left) or blurring (middle) the original image (top, 375120 bits), reveals only 1464 or 1720 bits respectively. By contrast, the bound our tool finds for the information revealed by a twisting transformation (right) is 375120 bits, no less than the input size. Applying the same transformation with the opposite direction to the twisted image gives back an image fairly close to the original (lower right).

information would prevent the original face from being reconstructed.

Figure 7-2 shows an original 125-pixel square image, which is represented by 375120 bits in an uncompressed format, and the output of three different transformations. Pixelation to a 5x5 grid uses the options `-sample 5x5 -sample 125x125`, while blurring uses `-resize 5x5 -resize 125x125`, and the twisting transformation uses `-swirl 720`. Though all three transformed images are visually unidentifiable, they differ greatly in the amount of information they preserve, as our tool verifies. Pixelation and blurring both involve shrinking the image to a small intermediate form and then enlarging it, so the maximum flow is dominated by the size of the intermediate form. Since ImageMagick uses 16-bit pixel component values internally, a 5-pixel square image is represented by 1200 bits. In addition there are some implicit flows, since the header of the file, which includes its size and other metadata, is also considered secret. In total our tool gives bounds of 1464 bits revealed for pixelation and 1720 bits for blurring.

On the other hand, the twist transformation computes each output pixel by finding the corresponding input image location under a continuous transformation, and interpolating between the four input pixels near it. There is no apparent bottleneck in this computation, so our tool's bound is the same as the input and output size, 375120 bits. Though the result is only an upper bound, and does not prove that no information is lost, it accords with the intuition that a continuous transformation is reversible, aside from blurring caused by the interpolation. In fact, a twist of the same magnitude in the opposite direction gives back an image fairly close to the original (and more sophisticated inversion techniques are possible).

7.6 OpenGroupware.org

OpenGroupware.org is a web-based system for collaboration between users in an enterprise, providing email and calendar features similar to Microsoft Outlook or Lotus Notes. We focused specifically on its appointment scheduling mechanism. Each user may maintain a calendar listing of personal appointments, and the program allows one user to request a meeting with a second user during a specified time interval. The program then displays a grid that is colored according to what times the second user is busy or free. This grid is intended to provide enough information about the second user's schedule to allow choosing an appropriate appointment time, but without revealing all the details of the schedule: for instance, the boundaries of appointments are not shown, and the granularity of the display is only 30 minutes.

We used our tool to measure the amount of information about the user's calendar this grid reveals, marking the starting and ending times of appointments as tainted when the program reads them with a SQL query.

For instance, for a proposal for a one hour appointment between 9:00am and 6:00pm, when the target user has an appointment from 10 to noon, our tool bounds the amount of information revealed as 12 bits. In previous experiments using the basic version of our tool, we had discovered that a loop that computes time period intersections unnecessarily considered times every minute, and fixed it to use the same half-hour interval as the final display; the 12-bit measurement corresponds to a cut at checks made in this loop.

This example also demonstrates the possibility of different flow estimates that are equally correct, but differ in when they are more precise. Later in the code, the objects created in the intersection-checking loop are used to decide whether each of the 18 squares in the grid should be colored beige or red; a cut there would measure every one-day appointment search as revealing 18 bits. For the case of a single morning appointment, a cut at the intersection loop gives a more precise bound, but if the user had many appointments, later in the day, an 18-bit bound from the display routine would be more precise.

7.7 X Window System server

In the X Window System commonly used on Unix, a single program called the X server manages the display hardware, and each program (X client) that wishes to display windows communicates with the server over a socket. The X server's mediating role makes it a significant potential source of security problems: programs can use it to communicate with each other (including using the same mechanisms that support cut and paste), and any information displayed on the screen also passes through the server. The original design of X addressed security only with respect to access control; more recently, the protocol has been extended with mechanisms that can enforce information-flow policies, by dividing clients into trusted and untrusted classes and restricting what untrusted clients can do [Wig96]. However, it can be difficult in a large monolithic system like the X server to ensure that enough permissions checks have been added. Since the X server is written in C, there is also the danger that an attack such as a buffer overflow could allow any checks to be subverted. As an alternate approach, we examined whether it is possible to avoid trusting most of the server implementation, and instead enforce our information flow goals directly. We used our tool to measure how much information from client programs is revealed to other clients or otherwise leaked from the server, by marking text data as secret when it arrived in requests used for cut-and-paste or drawing text on the screen.

Data bytes provided for cut-and-paste are uninterpreted by the server, and cause no implicit flows. By contrast, drawing text on the screen involves a number of computations: looking up bitmaps from a font, computing the width of the area drawn, and drawing each pixel according to the current rendering mode. The main effect is to change pixels in the framebuffer, which we do not count as a public output; but as a side effect, the server also computes a bounding box for the text that was drawn, for use in later redrawing calculations. The dimensions of this bounding box reveal information about the text that was drawn, in the same way that the dimensions of a black redaction rectangle in a declassified document would, by constraining the sum of the widths of the characters drawn inside.

For instance, our tool estimates (somewhat imprecisely) that in one font and drawing context, the bounding box generated from the string `Hello, world!` could reveal up to 21 bits about the characters of the string. However, on examining the location of this possible leak, it was clear to us that it could be eliminated by using a more conservative bounding box (not dependent on the contents of string), perhaps at the expense of requiring more redrawing later. Once the expected leaks are accounted for, either with cut annotations or algorithmic changes, a dynamic checking tool can catch any other information flows that violate the policy. For instance, we used our tainting-based checker with a single policy to catch both leaks caused by user errors, like pasting text from a secret application into an untrusted one, and code injection attacks, like a simulated exploitation of a an integer overflow vulnerability [Her07] in which code supplied via a network request walks through memory, looks for strings of digits that resemble credit card numbers, and writes them to a hidden file in `/tmp`.

7.8 Inferring enclosure regions

Enclosure regions, introduced in Section 3.2, are static program annotations that improve our tool's precision by directing the implicit flows from a code region to the locations holding results used by the rest of the program. This section discusses how they can be inferred by static analysis. We first describe the general approach, then describe a pilot study with a simple analysis tool. Even our very simple analysis tool discovered most of the annotations needed in our case studies, and the aspects it did not cover could be handled by other standard static analysis techniques.

An enclosure region delimits particular starting and ending program locations, and lists locations, which we call *outputs*, that hold results used in the rest of the program. If no implicit flows occur within them, enclosure regions have no effect, so an inference can simply choose starting and ending points enclosing every possible implicit flow operation in a program. Also, there is no harm in including extra outputs that might not be read. Therefore, the key challenge in inferring enclosure regions is, given a fragment of code in a program, to conservatively determine a list of data locations it might write to: essentially a kind of side-effect analysis. As with other kinds of side-effect analysis, it is necessary to take aliasing into account [CBC93, SR05]: in our case, the annotation requires an expression valid at the enclosure entrance that must-aliases the lvalue expression in a later assignment, similar to the interstatement must-alias pairs used by Qian et al. [QXM07].

For an initial assessment of the prospects for automatic inference of enclosure regions, we built a very simple pilot implementation, and compared its results to the complete hand-checked annotations used in the case studies above. The inference is a static analysis for C source code, based on the CIL framework [NMRW02]. It is intraprocedural, syntax-directed, and context-insensitive, operating as a single pass that disregards control flow except as implied by block structure. It does not use an

Program	hand annot.	need length	pilot analysis missed exp'n	interproc.	found
bzip2	79	17	17	13	49
eVACS	10	2	0	3	7
OpenSSH client	2	0	0	1	1
ImageMagick	23	1	1	0	22
X server	19	2	0	2	17

Figure 7-3: Summary of the results of the static analysis discussed in Section 7.8 to compute which locations a code region (containing an implicit flow) might modify. Overall, the pilot analysis found 72% ("found" column) of the hand-verified output annotations used in the case studies ("hand annotations" column).

alias analysis, so it only finds locations that can be named by the same expression at the region entrance as at the modification location.

Treating the set of output annotations used in the case studies as our target, we measured how many of the region outputs annotated by hand were found correctly by the pilot analysis. The results of the comparison are shown in Figure 7-3. (The remaining case studies are written in C++ or Objective C, so CIL cannot parse them.) Overall, even this very simple analysis found 72% of the required annotations: in most cases, the implicit flow, side-effect, and annotation were all close together, and no aliasing was involved.

We then further classified the remaining missed outputs, determining that more sophisticated analysis in two areas would be required to infer a full set of annotations: arrays, and interprocedural aliasing. The column "need length" in Figure 7-3 counts the outputs where the location being written to was a dynamically allocated array, and the enclosure annotation has a bound (currently supplied by hand) on the size of the array. These bounds would not be required in a language like Java whose arrays keep track of their own size. Among the output annotations the tool missed, the column "missed / expansion" counts cases where the inferred enclosure region referred to only a single element in an array, but it needed instead to refer to the entire array, commonly because the index expression was not constant. Finally, the column "missed / interprocedural" counts cases where the annotation we added by hand was in a different function than the side-effecting operation. While we found no cases in which an intraprocedural alias was required, interprocedural annotations often required that the modified location be referred to with a different expression in the annotation, such as by substituting an argument expression in place of a parameter in an lvalue expression.

Comparing the results between the various case study programs, `bzip2` is an outlier in the complexity of its annotations, because of its sophisticated use of arrays and pointers: for instance, to conserve space, many of its main data structures are allocated as subranges of two large arrays.

Chapter 8

Conclusion

Now that we have presented the main technical results of the thesis, this final chapter discusses some further issues about the utility of the technique, suggests directions for future research, and wraps up with a summary of our contributions.

8.1 Utility and applicability

This section provides some additional discussion of the ways in which a dynamic quantitative analysis would be useful in developing secure software, including which policies can be quantified, how to use a dynamic tool, and a comparison between our technique and standard tainting.

A quantitative policy may only be an approximation to the complete security policy one might specify—the projection of a set of acceptable and unacceptable behaviors onto a single axis—but it is usually sufficient to catch large categories of attack. For instance, in a system protecting privacy in a census database, a simple quantitative policy could not prevent the query "Was Stephen McCamant's income more than \$40,000?", since it carries the same amount of information as an acceptable query like "Was the average income of Boston residents more than \$40,000?". But it could prevent a query from requesting the incomes of everyone in Boston. Since the flow bounds our tool supports are whole numbers, it is also important to control the number of times an attacker might repeat a process, since even a small bound would become large if multiplied by a large number of repeated requests; but if the executions are analyzed together, our tool can be used to determine whether they are revealing the same or different information.

Our tool measures the flows in particular executions, and is intended for testing or debugging: its results do not say anything about other possible executions, which might leak either more information or less. As with any other kinds of testing, developers must choose inputs that exercise program behaviors relevant to a policy. It is still important for a dynamic tool that its results never underestimate the amount of flow that has occurred on a single run, even though this soundness for a dynamic analysis is different from soundness for a static analysis that describes all possible executions. As discussed in Chapter 4, other techniques can be used to check for violations of a policy on future executions, such as after a system has been deployed.

No matter how automated a flow measurement tool is, it is still the responsibility of a developer to decide which flows are acceptable, and how to resolve any violations. Using a tool like ours can be seen as a kind of machine-checked auditing: the developer conjectures a security policy the program is expected to satisfy, and the tool checks whether it really is satisfied in a particular case. Mismatches might either represent a policy that is too restrictive, or a bug in the program. The same kind of understanding and policy specification would be required to annotate a program with an information-flow type system: the difference is that a dynamic tool can be used to examine one program execution at a time, while a static approach requires that a policy covering every possibility be provided up-front.

Our analysis has a close relationship with dynamic tainting: the graph it constructs contains all the values that a tainting analysis would mark as secret. Our tool reports a flow of 0 bits in exactly the cases when a (sound) tainting analysis would allow a program. Conversely, any program with non-zero flow would be rejected by a taint analysis (counting the number of tainted output bits corresponds to the total capacity of edges to the sink in our graph). Using maximum flows allows our technique to find a more precise flow measurement, but it does not provide any more precise information about *which* parts of the output contain secret information. For instance, in the example of Section 3.3, 64 bits of the output are tainted, and our tool finds that together, these bits carry 9 bits of information about the secret input. But it is not possible to pick out a particular 9 bits out of the 64 that contain the information.

8.2 Future directions

Directions for possible further application of these ideas include interactions between different kinds of secret, replacing the dynamic parts of the current technique to produce a completely static analysis, and supporting interpreted languages without trusting the interpreter.

8.2.1 Different kinds of secret

If a program operates on different classes of secret information, such as Alice's secrets and Bob's secrets, or "classified" secrets and "top secret" secrets, our analysis can be used independently for each kind of secret. This is conceptually straightforward, and possible with our current tool just by running a program repeatedly, but for efficiency and ease of use, it would be better for to run the analyses together. A question is how much of the analysis can be shared between kinds of secret without hurting precision. For instance, would it be enough to have one set of graph capacities for any kind of secret, or should the bit-width analysis be repeated?

There may also be a possibility of increasing precision by analyzing the interactions between different types of secret, because of crowding-out effects: for instance, a certain byte might be able to store 8 bits of Alice's data, or 8 bits of Bob's data, but not both at once. However, the obvious approach of analyzing the flows of multiple kinds of information as multi-commodity flow would not be sound in general, because multiple information flows can share capacity via coding [ACLY00].

8.2.2 An all-static maximum-flow analysis

Since the dynamic analysis considered in the body of this paper already takes advantage of static inference, and we found that a flow graph labelled with static identifiers was fairly precise, it is instructive to consider how the same basic idea of network maximum flow could be applied to an entirely static version of the information-flow task. The flow graphs we consider are similar to the program dependence graphs used in slicing, and the dynamic bit-width analysis of Section 3.1 has a close static analogue [BSWG00]. The key difficulty is likely how to bound the number of times a static flow edge will execute, in terms of a developer-understandable parameter of the program input. The result of a static information flow analysis would need to be a formula in terms of such parameters, rather than a single number.

8.2.3 Supporting interpreters

In the past, information flow tracking for languages such as Perl and PHP has been implemented by adding explicit tracking to operations in an interpreter [WS91, NTGG$^+$05]. However, since such interpreters are themselves written in languages such as C, an alternative technique would be to add a small amount of additional information about the interpreter to make its control-flow state accessible to our tool in the same way a compiled program's is, and then use the rest of the tracking mechanism (for data) unchanged. This technique is analogous to Sullivan et al.'s use of an extended program counter combining the real program counter with a representation of the current interpreter location to automatically optimize an interpreter via instruction trace caching [SBB$^+$03]. Compared to a hand-instrumented interpreter, this technique would exclude most of the scripting language's implementation from the trusted computing base, and could also save development time.

8.3 Contributions

Protecting secret information is a key challenge for computer security, and a particularly difficult aspect is distinguishing when it is acceptable or unacceptable for a program to reveal some data that is derived from a secret. Previous researchers have suggested quantitative measurement of information flows as a way to distinguish acceptable from unacceptable flows: we show for the first time that the technique can be used in practice on real software.

We propose that dynamic tainting applied at the level of individual bits can be used for quantitative information-flow measurement. When combined with a technique such as our enclosure regions, such tainting can soundly account for implicit flows associated with secret-dependent control flow.

We introduce the idea that information flow can be measured as the maximum flow in a network of program operations. Based on this insight, we first describe a simple technique in which a flow bound can be obtained by describing a cut that separates a program's secret inputs from its public outputs. The we give a more complex technique in which a program analysis constructs a flow graph at run time. In support of the latter technique, we give a number of algorithms for operating on flow graphs, including a graph collapsing algorithm based on program structure that is critical to making the graph-based approach scale.

These techniques make it possible to assess a program's potential for information leakage via testing, which was previously difficult because the presence of secret information in a program's output is not directly visible. Some dynamic flow checking techniques also have low enough overheads to be used to catch all policy violations in secrecy-sensitive production applications. In particular, we use a minimum cut, which can be automatically derived from a maximum flow, to extend a technique based on parallel execution to support a quantitative policy.

We investigate formally what it means for a dynamic quantitative information-flow analysis to give sound results, and propose a new definition using concepts from coding theory. This definition is related to the channel capacity of the program for conveying information from an adversary who can choose arbitrary secret inputs, but can be achieved by an analysis that examines only a subset of executions. We also introduce a simulation-based soundness proof technique, and use it to show how the features of a core version of our analysis allow it to achieve soundness.

Finally, we study and evaluate our technique and a prototype implementation by applying it to six pre-existing open-source programs written in C, C++, and Objective C, totaling more than a million lines of code. For each program, we analyzed a the secrecy of a kind of information appropriate to the program. Depending on the programs and the way they were used, our tool's results either verified that the information was appropriately kept secret on the examined executions, or revealed unacceptable leaks, in one case due to a previously unknown bug.

Though ensuring that computer programs keep adequate control of secret information will continue to be a challenge for developers in the years to come, the research described here gives reason to believe that they will be helped in this task by increasingly sophisticated automated support. With the help of tools and techniques like the ones we have described, concerns of information-flow security can move from existing solely as abstract considerations in the minds of developers and users, to being concrete properties that can be tested and evaluated like other kinds of correctness.

Bibliography

[ACLY00] Rudolf Ahlswede, Ning Cai, Shuo-Yen Robert Li, and Raymond W. Yeung. Network information flow. *IEEE Transactions on Information Theory*, 46(4):1204–1216, 2000.

[AS05] Aslan Askarov and Andrei Sabelfeld. Security-typed languages for implementation of cryptographic protocols: A case study. In *Proceedings of the 10th European Symposium on Research in Computer Security (LNCS 3679)*, pages 197–221, Milan, Italy, September 12–14, 2005.

[BM07] Michael D. Bond and Kathryn S. McKinley. Probabilistic calling context. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2007)*, pages 97–112, Montréal, Canada, October 23–25, 2007.

[Bod93] Hans L. Bodlaender. A linear time algorithm for finding tree-decompositions of small treewidth. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing*, pages 226–234, San Diego, CA, USA, May 15–18, 1993.

[Bod05] Hans L. Bodlaender. Discovering treewidth. In *31st Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM)*, pages 1–16, Liptovský Ján, Slovakia, January 22–28, 2005.

[BP76] David E. Bell and Leonard La Padula. Secure computer system: Unified exposition and Multics interpretation. Technical Report MTR-2997 Rev. 1, MITRE Corporation, March 1976.

[BSWG00] Mihai Budiu, Majd Sakr, Kip Walker, and Seth Copen Goldstein. BitValue inference: Detecting and exploiting narrow bitwidth computations. In *European Conference on Parallel Processing*, pages 969–979, Munich, Germany, August 29–September 1, 2000.

[BT89] Giuseppe Di Battista and Roberto Tamassia. Incremental planarity testing. In *30th Annual Symposium on Foundations of Computer Science*, pages 436–441, Research Triangle Park, NC, USA, October 30–November 1, 1989.

[CBC93] Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Proceedings of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 232–245, Charleston, SC, January 1993.

[CCM08] Michael R. Clarkson, Stephen Chong, and Andrew C. Myers. Civitas: Toward a secure voting system. In *2008 IEEE Symposium on Security and Privacy*, Oakland, CA, USA, May 18–21, 2008.

[CF07] Deepak Chandra and Michael Franz. Fine-grained information flow analysis and enforcement in a Java virtual machine. In *23rd Annual Computer Security Applications Conference*, pages 463–475, Miami Beach, FL, USA, December 10–14, 2007.

[CGJ+07] Markus Chimani, Carsten Gutwenger, Michael Jünger, Karsten Klein, Petra Mutzel, and Michael Schulz. The Open Graph Drawing Framework. In *15th International Symposium on Graph Drawing*, Sydney, Australia, September 23–26, 2007.

[CGK+97] Chandra S. Chekuri, Andrew V. Goldberg, David R. Karger, Matthew S. Levine, and Cliff Stein. Experimental study of minimum cut algorithms. In *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 324–333, New Orleans, LA, USA, January 5–7, 1997.

[CHM04] David Clark, Sebastian Hunt, and Pasquale Malacaria. Quantified interference for a while language. In *Proceedings of the 2nd Workshop on Quantitative Aspects of Programming Languages (ENTCS 112)*, pages 149–159, Barcelona, Spain, March 27–28, 2004.

[CLO07] James Clause, Wanchun Li, and Alessandro Orso. Dytan: A generic dynamic taint analysis framework. In *ISSTA 2007, Proceedings of the 2007 International Symposium on Software Testing and Analysis*, pages 196–206, London, UK, July 10–12, 2007.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Electrical Engineering and Computer Science Series. MIT Press and McGraw-Hill, Cambridge, Massachusetts and New York, New York, 1990.

[CM08] Stephen Chong and Andrew C. Myers. End-to-end enforcement of erasure and declassification. In *21st IEEE Computer Security Foundations Symposium*, Pittsburgh, PA, USA, June 23-25, 2008.

[CPG+04] Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. Understanding data lifetime via whole system simulation. In *13th USENIX Security Symposium*, pages 321–336, San Diego, CA, USA, August 11–13, 2004.

[CSWZ00] Shiva Chaudhuri, K. V. Subrahmanyam, Frank Wagner, and Christos D. Zaroliagis. Computing mimicking networks. *Algorithmica*, 26(1):31–49, 2000.

[CT91] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory*. John Wiley, 1991.

[Den75] Dorothy Elizabeth Robling Denning. *Secure Information Flow in Computer Systems*. PhD thesis, Purdue University, May 1975.

[Den76] Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.

[DHW02] Alessandra Di Pierro, Chris Hankin, and Herbert Wiklicky. Approximate non-interference. In *15th IEEE Computer Security Foundations Workshop*, pages 3–17, Cape Breton, Nova Scotia, Canada, June 24-26, 2002.

[DO07] Will Drewry and Tavis Ormandy. Flayer: Exposing application internals. In *First USENIX Workshop on Offensive Technologies*, Boston, MA, USA, August 6, 2007.

[EKV+05] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and event processes in the Asbestos operating system. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, pages 17–30, Brighton, UK, October 23–26, 2005.

[Fen74] J. S. Fenton. Memoryless subsytems. *The Computer Journal*, 17(2):143–147, May 1974.

[FSBJ97] Elena Ferrari, Pierangela Samarati, Elisa Bertino, and Sushil Jajodia. Providing flexibility in information flow control for object-oriented systems. In *1997 IEEE Symposium on Security and Privacy*, pages 130–140, Oakland, CA, USA, May 4–7, 1997.

[GJK+01] Carsten Gutwenger, Michael Jünger, Gunnar W. Klau, Sebastian Leipert, Petra Mutzel, and René Weiskircher. AGD - a library of algorithms for graph drawing. In *9th International Symposium on Graph Drawing*, pages 473–474, Vienna, Austria, September 23–26, 2001.

[GM82] Joseph A. Goguen and José Meseguer. Security policies and security models. In *1982 IEEE Symposium on Security and Privacy*, pages 11–20, Oakland, CA, USA, April 26–28, 1982.

[Gra91] James W. Gray III. Toward a mathematical foundation for information flow security. In *1991 IEEE Symposium on Research in Security and Privacy*, pages 21–34, Oakland, CA, USA, May 20–22, 1991.

[GS76] Israel Gat and Harry J. Saal. Memoryless execution: A programmer's viewpoint. *Software: Practice and Experience*, 6(4):463–471, 1976.

[HAM06] Boniface Hicks, Kiyan Ahmadizadeh, and Patrick McDaniel. From languages to systems: Understanding practical application development in security-typed languages. In *Proceedings of the 2006 Annual Computer Security Applications Conference*, pages 153–164, Miami Beach, FL, USA, December 11–15, 2006.

[Her07] Matthieu Herrb. X.org security advisory: multiple integer overflows in DBE and Render extensions, January 2007. `http://lists.freedesktop.org/archives/xorg-announce/2007-January/000235.html`.

[HKNR98] Torben Hagerup, Jyrki Katajainen, Naomi Nishimura, and Prabhakar Ragde. Characterizing multiterminal flow networks and computing flows in networks of small treewidth. *Journal of Computer and System Sciences*, 57(3):366–375, 1998.

[HYH+04] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing web application code by static analysis and runtime protection. In *Proceedings of the Thirteenth International World Wide Web Conference*, pages 40–52, New York, NY, USA, May 17–20, 2004.

[KBA02] Vladimir Kiriansky, Derek Bruening, and Saman Amarasinghe. Secure execution via program shepherding. In *11th USENIX Security Symposium*, pages 191–206, San Francisco, CA, USA, August 7–9, 2002.

[KYB+07] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard OS abstractions. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, pages 321–334, Stevenson, WA, USA, October 14–17, 2007.

[LL05] V. Benjamin Livshits and Monica S. Lam. Finding security errors in Java programs with static analysis. In *14th USENIX Security Symposium*, pages 271–286, Baltimore, MD, USA, August 3–5, 2005.

[Low02] Gavin Lowe. Quantifiying information flow. In *15th IEEE Computer Security Foundations Workshop*, pages 18–31, Cape Breton, Nova Scotia, Canada, June 24-26, 2002.

[LS01] Peter Loscocco and Stephen Smalley. Integrating flexible support for security policies into the Linux operating system. In *Proceedings of the 2001 USENIX Annual Technical Conference (FREENIX Track)*, pages 29–42, Boston, MA, USA, June 25–30, 2001.

[LZ06] Peng Li and Steve Zdancewic. Encoding information flow in Haskell. In *19th IEEE Computer Security Foundations Workshop*, pages 16–27, Venice, Italy, July 5-6, 2006.

[Mal07] Pasquale Malacaria. Assessing security threats of looping constructs. In *Proceedings of the 34rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 225–235, Nice, France, January 17–19, 2007.

[McE02] Robert J. McEliece. *The Theory of Information and Coding*. Cambridge University Press, second edition, 2002.

[ME06] Stephen McCamant and Michael D. Ernst. Quantitative information-flow tracking for C and related languages. Technical Report MIT-CSAIL-TR-2006-076, MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, November 17, 2006.

[ME07a] Stephen McCamant and Michael D. Ernst. Quantitative information flow as network flow capacity. Technical Report MIT-CSAIL-TR-2007-057, MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, December 10, 2007.

[ME07b] Stephen McCamant and Michael D. Ernst. A simulation-based proof technique for dynamic information flow. In *PLAS 2007: ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, pages 41–46, San Diego, California, USA, June 14, 2007.

[ME08] Stephen McCamant and Michael D. Ernst. Quantitative information flow as network flow capacity. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation*, Tucson, AZ, USA, June 9–11, 2008.

[ML97] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 129–142, St. Malo, France, October 5–8, 1997.

[MPL04] Was Masri, Andy Podgurski, and David Leon. Detecting and debugging insecure information flows. In *Fifteenth International Symposium on Software Reliability Engineering*, pages 198–209, Saint-Malo, France, November 3–5, 2004.

[MPSW05] David Molnar, Matt Piotrowski, David Schultz, and David Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In *8th International Conference on Information Security and Cryptology*, pages 156–168, Seoul, Korea, December 1–2, 2005.

[MSZ04] Andrew C. Myers, Andrei Sabelfeld, and Steve Zdancewic. Enforcing robust declassificaiton. In *17th IEEE Computer Security Foundations Workshop*, pages 172–186, Pacific Grove, California, USA, June 28-30, 2004.

[Mye99] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Proceedings of the 26th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 228–241, San Antonio, TX, January 20–22, 1999.

[NMRW02] George C. Necula, Scott McPeak, S.P. Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Compiler Construction: 11th International Conference, CC 2002*, pages 213–228, Grenoble, France, April 8–12, 2002.

[NS05] James Newsome and Dawn Song. Dynamic taint analysis: Automatic detection, analysis, and signature generation of exploit attacks on commodity software. In *Annual Symposium on Network and Distributed System Security*, San Diego, CA, USA, February 3–4, 2005.

[NS07] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary insrumentation. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, pages 89–100, San Diego, CA, USA, June 11–13, 2007.

[NS08] James Newsome and Dawn Song. Influence: A quantitative approach for data integrity. Technical Report CMU-CyLab-08-005, Carnegie Mellon University CyLab, February 2008.

[NSCT07] Srijith K. Nair, Patrick N.D. Simpson, Bruno Crispo, and Andrew S. Tannenbaum. A virtual machine based information flow control system for policy enforcement. In *The First International Workshop on Run Time Enforcement for Mobile and Distributed Systems*, pages 3–16, Dresden, Germany, September 27, 2007.

[NTGG+05] Anh Nguyen-Tuong, Salvatore Guarnieri, Doug Greene, Jeff Shirley, and David Evans. Automatically hardening web applications using precise tainting. In *20th IFIP International Information Security Conference*, pages 295–307, Chiba, Japan, May 30–June 1, 2005.

[QWL+06] Feng Qin, Cheng Wang, Zhenmin Li, Ho-Seop Kim, Yuanyuan Zhou, and Youfeng Wu. LIFT: A low-overhead practical information flow tracking system for detecting general security attacks. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 135–148, Orlando, FL, USA, December 9–13, 2006.

[QXM07] Ju Qian, Baowen Xu, and Hongbo Min. Interstatement must aliases for data dependence analysis of heap locations. In *ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE 2007)*, pages 17–23, San Diego, CA, USA, June 13–14, 2007.

[SBB+03] Gregory T. Sullivan, Derek L. Bruening, Iris Baron, Timothy Garnett, and Saman Amarasinghe. Dynamic native optimization of interpreters. In *ACM SIGPLAN 2003 Workshop on Interpreters, Virtual Machines and Emulators*, pages 50–57, San Diego, California, USA, June 12, 2003.

[Sim03] Vincent Simonet. Flow Caml in a nutshell. In *First Applied Semantics II (APPSEM-II) Workshop*, pages 152–165, Nottingham, UK, May 26–28, 2003.

[SLZD04] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 85–96, Boston, Massachusetts, USA, October 7–13, 2004.

[SN05] Julian Seward and Nicholas Nethercote. Using Valgrind to detect undefined value errors with bit-precision. In *Proceedings of the 2005 USENIX Annual Technical Conference*, pages 17–30, Anaheim, CA, USA, April 10–15, 2005.

[SR05] Alexandru Sălcianu and Martin C. Rinard. Purity and side-effect analysis for Java programs. In *VMCAI'05, Sixth International Conference on Verification, Model Checking and Abstract Interpretation*, pages 199–215, Paris, France, January 17–19, 2005.

[SS05] Andrei Sabelfeld and David Sands. Dimensions and principles of declassification. In *18th IEEE Computer Security Foundations Workshop*, pages 255–269, Aix-en-Provence, France, June 20-22, 2005.

[ST06] Scott Smith and Mark Thober. Refactoring programs to secure information flows. In *PLAS 2006: ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, pages 75–84, Ottawa, Canada, June 10, 2006.

[VBC+04] Neil Vachharajani, Matthew J. Bridges, Jonathan Chang, Ram Rangan, Guilherme Ottoni, Jason A. Blome, George A. Reis, Manish Vachharajani, and David I. August. RIFLE: An architectural framework for user-centric information-flow security. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 243–254, Portland, OR, USA, December 4–8, 2004.

[VSI96] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, December 1996.

[Wig96] David P. Wiggins. *Security Extension Specification*. X Consortium, Inc., November 1996.

[WS91] Larry Wall and Randal L. Schwartz. *Programming Perl*. O'Reilly & Associates, 1991.

[XA06] Yichen Xie and Alex Aiken. Static detection of security vulnerabilities in scripting languages. In *15th USENIX Security Symposium*, pages 179–192, Vancouver, BC, Canada, August 2–4, 2006.

[XBS06] Wei Xu, Sandeep Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *15th USENIX Security Symposium*, pages 121–136, Vancouver, BC, Canada, August 2–4, 2006.

[YMC07] Aydan R. Yumerfendi, Benjamin Mickle, and Landon P. Cox. TightLip: Keeping applications from spilling the beans. In *4th USENIX Symposium on Networked Systems Design and Implementation*, pages 159–172, Cambridge, MA, USA, April 11–13, 2007.

[ZBWKM06] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. In *USENIX 7th Symposium on OS Design and Implementation*, pages 263–278, Seattle, WA, USA, November 6–8, 2006.