Quantitative Information-Flow Tracking for C and Related Languages

Stephen McCamant and Michael D. Ernst
Massachusetts Institute of Technology
Computer Science and AI Lab
Cambridge, MA 02139
{smcc,mernst}@csail.mit.edu

Abstract

We present a new approach for tracking programs' use of data through arbitrary calculations, to determine how much information about secret inputs is revealed by public outputs. Using a fine-grained dynamic bit-tracking analysis, the technique measures the information revealed during a particular execution. The technique accounts for indirect flows, e.g. via branches and pointer operations. Two kinds of untrusted annotation improve the precision of the analysis. An implementation of the technique based on dynamic binary translation is demonstrated on real C, C++, and Objective C programs of up to half a million lines of code. In case studies, the tool checked multiple security policies, including one that was violated by a previously unknown bug.

1 Introduction

The goal of information-flow security is to enforce limits on the use of information that apply not just to the first access to data, but to all subsequent calculations that use a piece of information. For instance, a confidentiality property requires that a program that is entrusted with secrets should not "leak" those secrets into public outputs. (Dually, integrity policies forbid untrusted inputs from affecting critical data.) Absolute information-flow properties, such as non-interference (a guarantee a system's public behavior is the same no matter what its secret inputs), are rarely satisfied by real programs: generally some information flow exists, and system designers, implementers, and testers wish to distinguish acceptable from unacceptable flows.

Our goal is to apply quantitative information-flow tracking to real legacy systems, by dynamically measuring the number of bits of information that may be leaked by a specific execution of a program. Compared to previous dynamic approaches, which have mainly focused on integrity properties such as preventing code injection, our focus on confidentiality properties requires a more complete and precise treatment of indirect flows. Though more similar in its goals to static information-flow approaches such as type systems, our technique is applicable to type-unsafe languages and larger systems that static systems have been demonstrated to scale to, and avoids the need for trusted declassification annotations.

Our technique is built upon tainting: it traces the flow of data through the program execution at single-bit granularity, tracking which bits are derived directly or indirectly from secret information. Tainting does not address the problem of indirect flows, but our technique handles them in two ways. First, a user may write annotations indicating a single-entry single-exit "enclosure region" of the code. No bit-tracking is performed within the enclosure region, but if any of the inputs is secret, then all of the outputs are secret. Second, at run time a global counter maintains a count that bounds the amount of information that may have been leaked indirectly; the total bound on leakage is the sum of the global counter plus the number of tainted bits that are output. Whenever a non-enclosed branch depends on secret data, the global counter is incremented; similarly for writes

through secret pointers. The bound may be unnecessarily high due to copies of data (outputting the same information repeatedly) or format conversions (outputting a long representation). For a more precise bound, our technique permits preemptive leakage of a compact representation, which erases the taints on n bits and adds n to the global counter. Both types of annotation are untrusted: if omitted or misplaced, the bound becomes less precise.

A static technique can guarantee that no program execution will leak any information (modulo trusted declassification annotations and the covert channels that are outside their scope). By contrast, our dynamic technique gives information about a particular program execution: it gives a quantitative result that upper-bounds the number of bits leaked on this particular execution (modulo covert channels outside its scope). The technique could be used for testing, to find leaks caused by design or implementation errors, or for run-time monitoring to terminate a program that begins to leak too much information because of a bug or an attack.

Our technique does not eliminate all covert channels: it concentrates on information that appears in the program's normal output, and for instance it does not prevent a program from manipulating the timing of externally-visible events (up to and including nontermination). We have also chosen an implementation of enclosure regions that makes them easier to use at the expense of potentially underestimating flows through some limited channels.

We have implemented our technique for x86 binaries and performed case studies on 4 programs written in C, C++, and Objective C, the largest of which is 550 KLOC. We verified an information-flow security property for each program — in one case after correcting an error that caused information leakage (our patch correcting the error has been accepted by the maintainers).

2 Related work

Our approach can be understood as a way of combining some of the attributes of two kinds of tool that have been studied in the past: static analyses (including type systems) that check programs for information-flow security ahead of time, and dynamic tainting analyses that track data flow in programs as they execute.

2.1 Static information-flow

If possible, it would be desirable to certify the information-flow security of programs before executing them; such *static* checking has a long history [Den76], with more recent activity centered on approaches using type systems as pioneered by Volpano, Smith, and Irvine [VSI96]. While the greater part of the contributions in this area have been to a theoretical understanding of information-flow security, there have also been important strides in making a type-system approach more practical, such as allowing for selective declassification [FSBJ97, ML97], and integrating checking with full-scale languages such as Java [Mye99], Caml [Sim03], and Haskell [LZ06]. However, there are still some significant barriers to the adoption of information-flow type-checking in day-to-day use. Even when information-flow-secure languages are built as extensions to general-purpose ones, they may 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 or Flow Caml dialects. A second obstacle is that techniques based on type safety are inapplicable to languages that do not guarantee type safety (such as C and C++) or ones that do not have a static type system in the first place (such as many scripting languages).

The security definitions used by information-flow type systems are generally based on preventing any information flow. For instance, many type systems guarantee non-interference, the property that for any public inputs to program, the public outputs will be the same no matter what the secret inputs were [GM82, VSI96]. However, it is often necessary in practice to allow some information flows, so such systems often include a mechanism for "declassification": declaring previously secret data to be public at certain code locations. Such annotations are a concise and straightforward mechanism for allowing certain information flows, but they have the disadvantage of being trusted: if they are placed incorrectly, a program can pass a type check but still leak arbitrary information. Our approach uses annotations that play a similar role (see Section 3.3), but are untrusted.

Some recent work has attempted to quantify the amount of information leaked in a computation, but the emphasis has been on precise schemes that can quantify leakages of much less than one bit. The classic motivating example for such work is password checking: suppose that passwords in a system are eight characters long, and an attacker who initially knows nothing about my password tries to log in using the password `i102rjD7` and is rejected. Intuitively, the attacker has learned something about my password, but very little. Such calculations can be made precise using information theory, though there is no consensus on the best definition to use. Generally, the result depends on the probability distribution of the secret; Clarkson et al. argue that the attacker's prior beliefs should also be considered, since an attacker can gain an unbounded amount of information from the result of a single yes-no query, if it allows him to correct a previous erroneous belief [CMS05]. However, such distributions are not likely to be available in most situations, so worst-case bounds must be used instead.

The most complete static quantitative information flow analysis for a conventional programming language of which we are aware is Clark et al.'s system for a simple while language [CHM04]. As with any purely static analysis, the requirement to produce a information flow bound that applies to any possible execution means that the technique's results will necessarily be imprecise for programs that leak different amounts of information when given different inputs. For instance, the authors consider 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. A formula giving precise per-iteration leakage bounds for loops is described in follow-on work by Malacaria [Mal07], but its application may be difficult to automate.

2.2 Dynamic tainting

Several other recent projects have applied dynamic checking to track data flow for security applications, but without a precise and sound treatment of indirect flows, such tools are not applicable to the confidentiality policies we focus on. In the best-known attacks against program integrity, such as SQL injection and cross-site scripting attacks against web applications and code injection into programs susceptible to buffer overruns, the data bytes provided by an attacker are used unchanged by the unsuspecting program. Thus, such attacks can be prevented by an analysis that simply examines how data is copied. On the other hand, many of the vulnerabilities that allow programs to inadvertently reveal information involve a sequence of calculations that transform secret input into a different-looking output that contains some of the same information. To catch violations of confidentiality policies, therefore, it is important to examine the flow of information through calculations that include many basic operators, including comparisons and branches that cause indirect flows.

The most active area of research is on tools that prevent integrity-compromising attacks on net-

work services; such tools generally ignore indirect flows or treat them only incompletely, but this does not prevent them from being effective against many attacks. Newsome and Song's Valgrind-based TaintCheck [NS05] is most similar to our proposed implementation, while other researchers have suggested using more optimized dynamic translation [KBA02, QWL+06], source-level translation [XBS06], or hypothetical 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].

Some dynamic tools to enforce confidentiality policies have been proposed recently, but none have satisfactorily accounted for all indirect flows in a way that is scalable to arbitrary programs. Most similar to our system is Chow et al.'s whole-system simulator TaintBochs [CPG+04], which traces data flow at the instruction level to detect failures to destroy sensitive data such as passwords. However, because it is concerned only with accidental copies or failures to erase data, TaintBochs does not track all indirect flows. Haldar, Chandra, and Franz [HCF05] track information flow at the object level for the Java virtual machine, but since their approach does not allow an object to write public data after reading secret data, it would prohibit most useful computations involving secrets. The RIFLE project [VBC+04] is an architectural extension that tracks direct and indirect information flow in concert with a compiler. The authors demonstrate promising results on some realistic small programs, but their technique's dependence on sound and precise alias analysis leaves questions as to whether it can scale to programs that store secrets in dynamically allocated memory.

Restrictions on information flow can also be enforced by an operating system, though the granularity of traditional processes and files is too coarse for the sorts of examples we consider. A new operating system architecture with lightweight memory-isolated processes, such as the "event processes" of the Asbestos system [EKV+05], is more suitable for controlling information flow, but is not compatible with existing applications.

3 Technical approach

This section outlines the information-flow tracking technique we have implemented, starting with our definition of information flow (Section 3.1) and the basic bitwise tainting analysis (Section 3.2), and then describing the two kinds of annotations (Sections 3.3 and 3.4) that are key to obtaining precise results. We give informal justifications for our goal that the tool's output be an upper bound on the amount of information leaked in any single execution, but no formalization or soundness proof. With a running example, we show how refinements to successive refinements to the technique yield increasingly precise information-flow bounds.

3.1 Information flow as entropy

Our technique uses a definition of information leakage based on entropy, specialized to embody conservative assumptions about information not available to a program analysis.

Our fundamental definition of the amount of information leaked by a program is the entropy of the program's outputs as a distribution over the possible values of the secret inputs, with the public inputs held constant. Since we do not envision any sources of randomness internal to the program, this is equivalent to the conditional mutual information between the output and the secret inputs, given the public inputs. This is the same definition used by Clark et al. [CHM04], and analogous

```
char checksum(char *hex_str) {
    unsigned sum = 0;
    for (int i = 0; i < size; i++) {
        if (isdigit(hex_str[i]))
            sum += hex_str[i] - '0';
        else
            sum += hex_str[i] - 'a' + 10;
    }
    sum = (-sum) & 15;
    if (sum < 10)
        return '0' + sum;
    else
        return 'a' + (sum - 10);
}
```

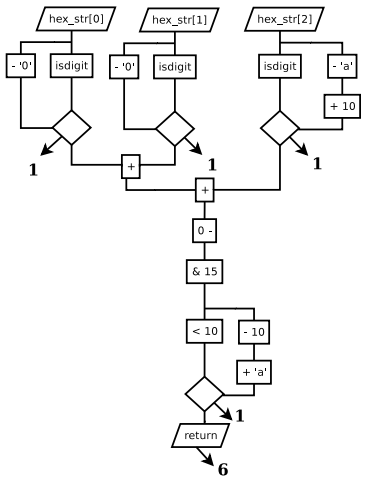

Figure 1: On the left,a C function to compute a checksum for data represented as a hexadecimal string. The output is a hex digit such that the sum of that digit and those in the input a multiple of 16. On the right, a representation in the form of a circuit of the operations performed by an execution of the function. Data flows from inputs (top row parallelograms) through primitive computations (rectangles) to outputs (bottom row parallelograms). Values that depend on a control-flow branch are represented by diamonds, where the top input is the branch condition, the left or right inputs are the values when the condition is true and false, and the output is at the bottom. Diagonal arrows show the 10-bit information flow bound estimated by our tool when operating without annotations.

to other information-theoretic measures used in the literature (e.g., [Mil87]). It is also roughly the logarithm of the effort that knowledge of the output would save an attacker attempting to guess the secret by brute force [Mal07]. Non-interference corresponds to an entropy of 0.

The definition in the preceding paragraph is based on a probability distribution over possible secret inputs, but on a single execution, only one set of inputs is available. To account for this, our technique makes two worst case assumptions: first, that every bit of the secret input might vary, and second, that every bit of data may carry a full bit of information. The first assumption is straightforward: the set of possible secret inputs is unknown, so the safest choice is to assume that any other values might have been possible. (Note, however, that the *size* of the input is still treated as fixed.) The second assumption is required by the tool's ignorance of the relative probabilities of different inputs, but it is applied throughout analysis, not just to the inputs: whenever a bit might be either `0` or `1`, it is considered to contain 1 bit of information (entropy). For instance, in the example of a password-checking routine mentioned in Section 2.1, our technique would estimate that revealing whether a password is correct reveals at most one bit of information. The 1-bit bound corresponds to a worst-case distribution of possible inputs that is not uniform: rather, it is the distribution in which an attacker knows that each of two possible passwords is correct with probability 1/2.

To make the above discussion concrete, and to introduce a running example used in the remainder of this section, consider the function shown in the left half of Figure 1 to compute a checksum for a string of hexadecimal digits. Since the check digit can take one of 16 possible values `0` through `f` based on the secret input string, by our definition it reveals 4 bits about the contents of the string.

In further analysis of this example, it is helpful to consider it as a circuit, in which the con-

nections between basic operations form a data-flow network specialized for particular control-flow choices; this is illustrated in the right half of Figure 1.

3.2 Bit tracking analysis

The basic principle of our analysis is that the entropy of the output is bounded by the number of bits of the output that depend on the secret input; to determine which output bits depend on the input, our tool uses a simple conservative bit-tracking analysis, essentially dynamic tainting applied at the level of individual bits. Bits that are not affected by the secret input carry no information about it, and bits that might be affected by it each reveal at most one bit of information about it (another application of the one-bit-per-bit principle introduced in Section 3.1). Each data bit in the program has an associated *secrecy bit*, which is initially `1` for the bits of the secret input and `0` for other values. For each program operation, an output bit is marked as secret if it could be either `0` or `1` depending on the value of a secret input bit. Though dynamic, this analysis is still conservative in some of the ways that a static analysis would be, since it does not consider all of the potential alternate secret inputs one by one. For instance, subtracting a secret value from itself will give a secret rather than a non-secret 0 as the result, since the analysis does not track which secret bits are correlated with each other.

Based on this level of analysis, which is completely automatic, our tool can compute a relatively imprecise bound on the amount of information that an execution reveals. In the case of the checksum example, the tool computes a bound of 10 bits, of which 6 come from its conservative determination that the six least significant bits of the return value might depend on the secret. (The remaining 4 bits come from the treatment of branches, which will be justified in Section 3.3 and refined in Section 3.4.) This particular bound depends on the way the code is compiled (with the `'a'` and `-10` constant folded, in this case); other semantically equivalent code could give larger or smaller estimates.

In the above discussion, and our current implementation, the secrecy tag for each bit is itself one bit, so that each data bit is either secret or public. This choice makes some implementation aspects easier: for instance, the secrecy bits for a machine word can be stored in a second machine word. However, it would be straightforward to generalize the approach to labels drawn from a larger lattice: bit operations would simply compute the join of the labels of the relevant inputs. Most convenient would be a subset lattice: for instance, to represent which subset of users from the set $\{\text{Alice}, \text{Bob}\}$ require a particular value to be secret, the one-bit analysis can simply be duplicated, with one set of bits representing Alice-secrecy, and the other Bob-secrecy.

3.3 Preemptive leakage

To obtain more precise bounds on the amount of information that program executions reveal, our technique uses additional information about the structure of a program's computations. This information is currently provided by code annotations, though many of these annotations could be inferred automatically using other kinds of analysis. We first introduce *preemptive leakage* annotations, which we the tool uses to measure information at a point where it is represented in a minimal number of bits.

The change to the basic bit-tracking algorithm made by these annotations is justified by a simple but powerful fact about information: in a closed computational system, new information is never created. Thus, if we select any region of our program's execution, and measure the amount of

information going in and coming out, the total entropy of the outputs will always be at most the sum of the entropies of the inputs. As with the analogous results about the flows of electromagnetic fields and fluids, the power of this simple fact about information flow (essentially the Data Processing Theorem of information theory) comes from the scope for creativity in choosing the boundaries of the region to apply it to.

Preemptive leakage applies the no-new-information principle to the high-level structure of a program's secret computation. It is based on the intuition that while information is often represented in many different formats during a program's execution, there is at least one place ("in the middle") when it has its most compact representation. This location of the compact representation is the ideal place to measure the flow of secrets through a program. In particular, none of the computations done subsequently in execution are important: if we take the "second half" of the program to be all of the computations that operate on the compact representation of a secret to produce the final output, nothing in the second half can increase the flow beyond the flow through the compact representation.

To implement this intuition, we give our technique a second way of counting secret bits: besides the tainting mechanism that associates secrecy with particular bits, it also keeps a count in bits of other information that may have leaked. At any point in execution, the program is allowed to remove the taint from a bit, as long as it increments the counter at the same time. Assuming the information in the bit would have eventually been revealed, the advantage of doing this is that subsequent expansions of the data (such as copying) will not multiply tainting and so will not be counted as adding to information leakage. With this approach, it is the sum of the number of tainted bits and the leakage counter that is a bound on the secret information in a program; annotations direct the tool to use one kind of bound or the other to minimize imprecision.

Our tool uses this same mechanism to account for information leaks via indirect flows, which may be present when the program performs a branch or pointer operation that might behave differently if a secret condition or pointer value were different. (A further technique for bounding such flows is discussed in Section 3.4.) The basic bit-tracking analysis of Section 3.2 does not specify the result when secrets are used as the basis for a control-flow branch, inasmuch as it would not be feasible to execute both sides of every branch and then combine the results. The same problem affects indirect memory operations whose effective addresses are secret: array accesses with secret indexes, loads and stores through secret pointers, jump tables, and calls via secret function pointers can also reveal information. Using preemptive leakage, we can achieve the effect of never passing secret data to such an operation by enforcing that any branch conditions or pointers be leaked right before the operation: in practice, the leak is treated as an integral part of the instruction. For instance, to obtain a bound for an execution that includes branches, our tool treats the branch condition (a single bit, for a two-way branch) as being leaked at the point the branch is encountered. The program then executes only the side of the branch it would have taken normally. The same argument as before shows why the count obtained in this way is still an upper bound on information flow, though because only one half of the branch is examined, the result says nothing about what would have happened under a different control-flow path.

In the checksum example, the most compact representation of the secret is the binary value of the checksum, before it is converted back into ASCII: as an integer between 0 and 15, it is easy for our analysis to tell that it contains 4 bits of information. By eliminating a leak from a branch and the imprecision of our tool's analysis of the binary to ASCII conversion, a preemptive leakage annotation there (marked with `/* 1 */` in Figure 2) reduces the tool's estimate of the leakage in

the running example to 7 bits.

The preemptive leakage annotations in our technique have a similar motivation to the declassification annotations in static information-flow type systems: marking program locations that represent the desired information flow in a program. However, the relation of preemptive leakage to our quantitative measure of flow is very different from the relation of declassification to noninterference: in short, our annotations are untrusted rather than trusted. An incorrectly placed leakage annotation will cause our tool's estimate of a program's information flow to be too high, making it clear that the security goal has not yet been met. By contrast an incorrectly placed declassification annotation can cause a static type system to allow a program even when it contains flows the programmer did not intend.

3.4 Enclosure of indirect flows

Leaking a bit for every branch on a secret condition can lead to a quite imprecise information flow bound: the one bit per input character leak in the checksum example is typical of many programs that parse their input in one way or another. However, such branches do not affect the course of execution of the program at a high level: they are just an implementation detail of a function that is executed unconditionally. Intuitively, their contribution to the program's result should be subsumed in the tainting of the result of the conversion. As with preemptive leakage, this intuition can be justified by dividing the program's execution into pieces, in this case looking at just the branch and the code around it separately. In particular, consider a region that encloses both sides of the branch: the computations in this region do not affect the control flow of the rest of the program, so if they affect the output at all, it must be by data flow. This can be accounted for by marking all of the outputs of the region as tainted. This treatment is similar to the rules for `if` statements in static information-flow type systems (if the branch condition is secret, so are the values written in it), but indicated by separate annotations.

These *enclosure regions* are our tool's most powerful approach for accounting for indirect flows. Though the basic idea is simple, some care is required in translating two key notions into an imperative language: how is a single control flow region delimited, and what constitute its outputs? The choices we make are practical, but leave some limited-capacity channels for covert flow which we argue are tolerable.

In order to prevent leaks via control-flow, an enclosure region should be single-exit: if control enters the region, there should only be one way for it to leave. In our current prototype, the entrance and exit of a region are represented as separate annotations connected with a unique identifying number. The tool detects at runtime if entrances and exits are mismatched; if an exit is bypassed, all of the program's future outputs will be counted as leaks. A related but more difficult problem is that a program might reach a state inside an enclosure region which in normal execution would cause it to terminate, for instance because of an exception like a null pointer dereference. By choosing whether to crash based on a secret, a program could reveal information via the amount of normal execution it finished before stopping: for instance, it could leak a number n by crashing after printing the first n characters of its input.

The other class of difficulties relate to measuring the output of one region of a program. Our tool's basic policy is to mark every register and memory location written by the enclosed code as tainted, but this policy does not account for every possible flow. In particular, it does not account for the possibility that enclosed code uses secret information to decide whether or not to write, or which locations to write to. The first problem is the one recognized first by Fenton [Fen73], in

an example like `b := 0; if (a == 1) then b := 1; if (b == 0) ...`: if `a` is secret and `0`, the fact that `b` is still `0` after the first branch reveals information about `a`. The second problem is a generalization made possible by random-access memory: for instance, by setting only one entry in a 2^k-entry array, a program can convey k bits of information though only one tainted value is read. These attacks would both cause the expected value of our tool's information-flow bound to be too small by a constant factor, and in unlucky cases (where "luck" is defined over the secret value) could allow a flow when the computed entropy bound is 0. (Note that both of these cases are only problematic when the branch or array operation is inside an enclosure region: the basic preemptive leakage treatment of indirect flows accounts for them conservatively.)

We have considered some changes to eliminate the above attacks. For the termination channel, the tool could treat enclosed as revealing an appropriate amount of information, or prevent it by ignoring machine exceptions, but it may not be possible for execution to continue normally after an exception, and the code can always loop infinitely. For the output-related channels, a possibility would be requiring enclosure regions to pre-declare their output variables, always marking these as tainted, and rolling back any other changes the region makes to memory. However, we have decided for the moment that our tool is more useful in the current state, in which the annotation burden is less and the security guarantee provided is less complete. In particular, it is often useful to create enclosure regions that are large, and whose outputs, while functionally related, would be cumbersome to enumerate explicitly. The covert channels corresponding to these attacks seem relatively tolerable in part because they seem less likely to be introduced by simple coding mistakes or code injections, but also because even if present, they would require significant effort to exploit: quantities of output exponential in the size of the leaked secret for the termination channel, and time and space exponential in the expected-underestimate-factor for the address channel.

As an additional convenience, the enclosure regions we have implemented have the additional feature that they are delayed in taking effect until the first operation that would otherwise cause a leak. In particular, this gives a simple kind of polymorphism: if a general-purpose piece of code operates sometimes on secret data and sometimes on public data, its outputs will not be marked secret if none of its inputs were.

Using the annotations introduced in this and the previous section, a final version of the checksum example is given in Figure 2; the lines labelled `/* 2 */` show an enclosure annotation around the ASCII to binary conversion of each input character. Together, they allow for a precise estimate of the information revealed by the code's execution: 4 bits.

4 Implementation

We implemented our technique as a modified version of the Valgrind Memcheck tool [NS03, SN05]. It uses dynamic binary translation to insert bit tracking instructions into a program as it executes. The annotations mentioned in Sections 3.3 and 3.4 are preprocessor macros whose expansions signal a Valgrind tool to take a special trusted action. Except for the inclusion of an additional header file, annotated programs can use their standard build process. The runtime overhead of our tool is similar to Memcheck's, making it usable for testing and some specialized applications such as honey-pots, but too slow for full-time production use (slowdown factors between 10x and 100x for CPU-bound programs). A performance-oriented implementation could reduce but not eliminate this overhead (e.g., to 3.6x as in [QWL$^+$06]).

A user must specify which sources of data are considered tainted for a particular application.

```
char checksum(char *hex_str) {
    unsigned sum = 0;
    for (int i = 0; i < size; i++) {
        ENTER_ENCLOSE(42); /* 2 */
        if (isdigit(hex_str[i]))
            sum += hex_str[i] - '0';
        else
            sum += hex_str[i] - 'a' + 10;
        EXIT_ENCLOSE(42); /* 2 */
    }
    sum = (-sum) & 15;
    sum = LEAK(sum); /* 1 */
    if (sum < 10)
        return '0' + sum;
    else
        return 'a' + (sum - 10);
}
```

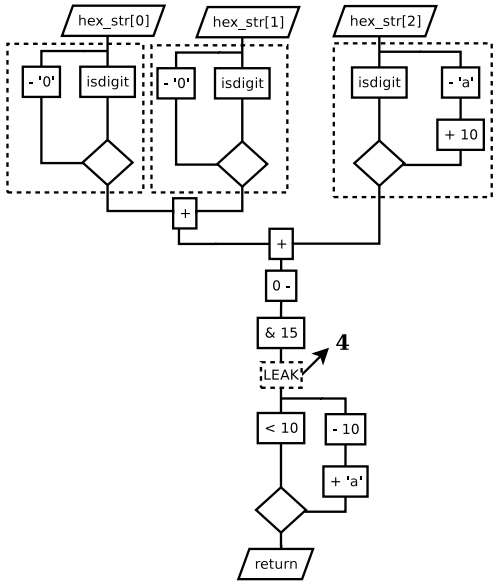

Figure 2: Preemptive leakage and enclosure of indirect flows. Here the hexadecimal checksum example of Figure 1 has been annotated to enclose computations that branch on secret data (`ENTER_ENCLOSE` and `EXIT_ENCLOSE` in the code, dotted rectangles in the circuit), and to preemptively leak the value of the checksum while it is stored in binary (`LEAK` code annotation and circuit operation). With these annotations, the amount of information leaked by the code is be precisely computed as 4 bits per execution.

The tool includes some options to support this, such as treating data as secret if it is read from a file with restricted Unix filesystem permissions; in other cases the information can be marked with an annotation at the code location where it is input. Tainted bits are considered as output if they are used as arguments to a system call. A report is printed to the standard error stream for each leak of bits via a branch, memory address, or output, with a total at the end of execution; each leak may optionally be accompanied by a backtrace giving its code location, which is useful for debugging.

5 Case studies

To learn about the practical applicability of our tool, we used it to test a different security property in each of four open-source applications. The programs illustrate several different kinds of confidentiality goals and are written in three different C-family languages. Some basic statistics of the programs are shown in Figure 3.

In each case, we began by adding tainting to an input to the program, and measured the resulting leaks; we then added additional annotations until the measured leakage matched our expectations of correct behavior, or we found a too-large flow that was real, indicating a secrecy-compromising bug. In some cases, later annotations made earlier annotations unnecessary, but redundant annotations are not harmful, so we have included them in our counts of annotations to better reflect programmer effort.

Program	KLOC	Language	Number of libraries	Number of annotations	
				Leakage	Enclosure
KBattleship	6.6	C++	37	8	9
OpenSSH client	65	C	13	2	4
OpenGroupware.org	550	Objective C	34	0	19
eVACS	9.3	C	7	1	9

Figure 3: Summary of the programs examined in the case studies of Section 5. The sizes of each program, measured in thousands of lines of code (KLOC), include blank lines and comments, but do not include binary libraries (4th column, measured with `ldd`) which were included in the bit tracking but did not require annotation. The annotations in each case were those used for checking a particular policy explained in the text.

5.1 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 they 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 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, only one bit ("hit" or "miss") should be revealed for each of the opponent's shots.

We used our tool to test this property on KBattleship, an implementation of the game that is part of the KDE graphical desktop. We were inspired to try this example because the source code distribution for Jif, a statically information-flow secure Java dialect (the latest descendant of the work described in [Mye99]) includes as an example an implementation of a Battleship game. However, the two versions of the game are not directly comparable: Jif Battleship appears to have been written from scratch in Jif, contains about 500 lines of code, and uses no libraries more complicated than a linked list data structure and a print routine. It does not include a capacity to play across a network, or in fact any user interface; the program simulates both sides of a game in a batch mode using hard-coded ship positions. KBattleship is more than 10 times as large, even before including the many libraries it uses, and has a GUI and a networked multi-player mode which was the one we tested.

Also apparently unlike Jif Battleship, the version of KBattleship we examined (3.3.2) does contain an information leak bug. Specifically, in responding to an opponent's shot, a routine calls a method `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 we have submitted a patch for it which appears in version 3.5.3. Though this bug would have been detected with our tool, we discovered it by inspection before the tool was implemented. So instead, we used our tool to check whether the bug was eliminated in the patched version of the program.

We used three annotations to taint variables representing the position and orientation of each

of the player's ships before they are added to a data structure representing the board. We then added a total of 17 annotations of the following kinds:

(a) Eight enclosure regions around calculations that populate or access the board data structures (for instance, to enclose branches that check whether a ship's orientation is horizontal or vertical)

(b) One enclosure region around the call to the `shipTypeAt` accessor method used in replying to an opponent shot message

(c) One preemptive leakage of a variable used in the reply method to determine whether to send a 'hit' or a 'miss' message

(d) Seven preemptive leaks of other variables with specified bit widths in the calculation of a reply message in the case of a sunken ship

With these annotations the preemptive leakage of a single bit in (c) is the only information flow to the network our tool reports in response to a miss or a non-fatal hit by the opponent, confirming the expected security policy for the patched code (except for a limitation described below). The additional annotations in (d) allow the tool to report that the number of bits leaked in response to a hit that sinks a ship is 11; the response in this case includes the type of the sunken ship and its location. In some places, we also needed to make behavior-preserving code modifications, such as replacing repeated method calls with a temporary variable.

Unfortunately, there was one additional kind of information flow in the KBattleship example that our tool was not able to soundly account for. In addition to being used in network replies, information about the player's ship locations is also used to update the player's graphical display. Since the display is not visible to the opponent, this should not be prohibited, but it presents a difficulty for our tool because the GUI libraries used retain state, and are used for both secret and public data. It would be difficult to prove that displaying secret data does not change the state of the library in a way that might, say, affect future inputs. Since we did not consider it acceptable to annotate the GUI libraries in this case, we instead declassified (untainted) ship location information in six locations where it was passed to drawing routines. This means that our information flow bound does not account for the possibility that information about the ship locations derived via the GUI library might leak to the network, which while unlikely to happen intentionally is hard to rule out given the complexities of the libraries and the fact that the whole program shares an address space.

5.2 OpenSSH

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 an 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. Inspired by a discussion of this example by Smith and Thober [ST06], we use our tool to check how much information about the private key is revealed by a client execution using this authentication mode.

According to the protocol, the server sends a challenge encrypted with the public key; the client decrypts the challenge with the private key, computes the MD5 checksum of the decrypted challenge and a session ID, and sends this hash to the server. We mark the private key, which is represented by a number of arbitrary-precision integers, as tainted right after it is read from a file. Since the information-flow here intimately involves cryptography, we also add annotations to the OpenSSL encryption library that OpenSSH uses: half of the annotations lie there rather than in the main program. However, the total number of annotations remains quite low because the entire calculation of the response, including both the RSA decryption and the MD5 checksum, can be included in a single enclosure region. Each of the bytes of the checksum is then preemptively leaked right before being added to the response packet: this is necessary because the conversation between the client and server is encrypted in CBC mode, which causes every bit output to depend on all of the previous ones. The remaining enclosure regions are required in the code that initializes and clears the key-related data structures. With these annotations, our tool verifies OpenSSH's network reply as including 128 bits (the size of an MD5 checksum) derived from the private key.

Though this result matches our expectations (each bit of the checksum depends on the key), it highlights a way in which a quantitative security definition does not capture the reason the protocol is secure. It would not be acceptable for OpenSSH to send, say, an arbitrary 128-bit segment of the private key in a network response. The program is acceptable because the MD5 checksum is believed to be difficult to invert, but our tool cannot distinguish this feature of a function, nor is there even an obvious way to measure it. However, we have added one feature to our tool that provides some additional assurance in this instance: we have implemented a single trusted operation that computes an MD5 checksum and preemptively leaks it atomically, which explicitly enforces the connection between the hash function and the leak.

5.3 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 which 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 in bits the amount of information this grid reveals.

Most of the calculations relevant to this information flow involve dates, so we included the date processing library the program uses among the code to be annotated. We mark the starting and ending times of appointments as tainted as the program reads them with a SQL query. A relatively large number of enclosure annotations were needed, not because of the large overall size of the program (only few portions needed to be annotated), but because a large number of separate methods operate on dates. In this example, as contrasted with OpenSSH and KBattleship, it was not always possible to enclose a single large calculation with one enclosure region, in part because of an implementation detail of the Objective C runtime. It appears that the version of the GNU Objective C runtime used by OpenGroupware.org sometimes performs state-modifying operations during method dispatch, for instance because of lazy initialization. If a dispatch that modifies

metadata occurs inside an enclosure region, it can taint that metadata and cause future calls to show up as leaks, so our enclosure regions could not span method calls.

By propagating the tainting of secret date information through date operations, our tool's tainting reaches a loop that compares the secret dates with a series of regularly spaced intervals used to later form the grid display; the loop makes a multi-way comparison on each iteration, but exits early when it has exhausted all of the appointments. At this point, it is convenient to take advantage of the tool's default behavior of counting branches as leaks, and to use the comparisons in this loop as a measure of the information included in the grid. The original loop unnecessarily considered potential intervals starting every minute, causing both a large leakage estimate and a slight runtime inefficiency. By improving the loop to use half-hour intervals matching the grid size and making corresponding changes to some boundary conditions, we obtained the same grid with quite precise flow bounds. For instance, for a proposal for a one hour appointment between 9:00am and 6:00pm, when the target user has an appointment from 11:00am to noon and leaves the office at 6, our tool bounds the amount of information in the grid at 16 bits.

5.4 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. Therefore, we would like to check that the barcode data is used in the client program only as intended: verified using an internal checksum and then sent to the server on two occasions. We modified the client so it can run by itself on a regular Linux workstation.

Barcodes in the eVACS system consist of 128 bits of data and a 4-bit checksum, encoded in 22 ASCII characters. We modified the client to simulate the scanning of a barcode in response to a key press, and marked the barcode data as tainted. We are interested particularly in where the barcode data propagates, so we never preemptively leaked the data: our tool tracks it all the way to the `write` system calls used for output. Specifically, our tool counts the 176 bytes of the barcode in 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 that are acceptable. Each time the barcode is read, a computed checksum is compared to the one encoded at the end of the barcode; 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.

6 Future work

The work described so far has shown some of the potential uses of dynamic quantitative information-flow tracking, but research in a number of other directions could make the tool more practical and

allow it to provide higher assurance. Here we mention the inference of annotations, the reuse of the tool for interpreted languages, techniques for visualizing information flow measurements, and the prospects for formalization.

The annotations required by the current version of the tool, though not excessive in number, are somewhat cumbersome to discover one by one. Though some annotations require an understanding of the a program's algorithm and security context, others are routine and local, such enclosure regions around small branching computations. Because the annotations required by our tool are untrusted, it is reasonable to imagine an automated process discovering and automatically applying ones that would be beneficial. For instance, the most compact representation of a secret could be discovered automatically.

In the past, information flow tracking for languages such as Perl and PHP [WS91, NTGG+05] has been implemented by adding explicit tracking to operations in an interpreter. However, since such interpreters are themselves written in languages such as C, an alternative approach 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 approach 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.

Our current tool can pinpoint the location of a leak with a stack backtrace, but provides little support for understanding why a particular value is tainted, or how information flows between components of a large system. While we have argued that information flow has an intuitive graphical structure, the instruction-level flows our tool works with would be too large to visualize directly, as shown by the experience of the Redux tracing tool [NM03] (also based on Valgrind). An effective tool for visualizing flows would make it much easier to understand the errors our tool finds, but only if it was able to aggregate and structure the flows using elements of program structure that developers are already familiar with, such as procedures.

Information flow is a tricky aspect of program behavior to reason about, and it is all too easy to overlook channels for information flow or design mechanisms that are vulnerable to subtle attacks. Though our approach has been to design a tool starting from the constraints of applying it to real programs, the best way to gain confidence in the guarantees that a security technique purports to provide is to prove results about a simplified version of it in a formal context. Though our technique has a different flavor than most that have been considered in more theoretical literature, we believe that all of its basic elements could be shown to be sound in an appropriate formal model. Performing such a formalization would likely also give insights into ways the technique could be simplified or generalized.

7 Conclusion

We have presented a new technique for dynamically tracking information flow in programs. Using a practical quantitative definition of leakage, the technique can measure the information revealed by complex calculations involving indirect flows. By applying that definition with a robust instruction-level bit tracking analysis, it is applicable to real programs written in languages such as C and C++,

requiring only a few developer-supplied annotations. In a series of case studies, our implementation of the technique checked a wide variety of confidentiality properties in real programs, including one that was violated by a previously unknown bug. We believe this technique points out a promising new direction for bringing the power of language-based information-flow security to bear on the problems faced by existing applications.

References

[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.

[CMS05] Michael R. Clarkson, Andrew C. Myers, and Fred B. Schneider. Belief in information flow. In *18th IEEE Computer Security Foundations Workshop*, pages 31–45, Aix-en-Provence, France, June 20-22, 2005.

[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.

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

[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 32–26, 2005.

[Fen73] J. S. Fenton. *Information Protection Systems.* PhD thesis, University of Cambridge, Cambridge, UK, 1973.

[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.

[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.

[HCF05] Vivek Haldar, Deepak Chandra, and Michael Franz. Practical, dynamic information flow for virtual machines. In *2nd International Workshop on Programming Language Interference and Dependence*, London, UK, September 6, 2005.

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

[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*, Nice, France, January 17–19, 2007.

[Mil87] Jonathan K. Millen. Covert channel capacity. In *1987 IEEE Symposium on Security and Privacy*, pages 60–66, Oakland, CA, USA, April 27–29, 1987.

[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.

[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.

[NM03] Nicholas Nethercote and Alan Mycroft. Redux: A dynamic dataflow tracer. In *Proceedings of the Third Workshop on Runtime Verification*, Boulder, CO, USA, July 13, 2003.

[NS03] Nicholas Nethercote and Julian Seward. Valgrind: A program supervision framework. In *Proceedings of the Third Workshop on Runtime Verification*, Boulder, CO, USA, July 13, 2003.

[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.

[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*, Orlando, FL, USA, December 9–13, 2006.

[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.

[ST06] Scott Smith and Mark Thober. Refactoring programs to secure information flows. In *ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, 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.

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

[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.