Quantitative Information Flow as Network Flow Capacity

Stephen McCamant   Michael D. Ernst

MIT Computer Science and AI Lab
{smcc,mernst}@csail.mit.edu

Abstract

We present a new technique for determining how much information about a program's secret inputs is revealed by its public outputs. In contrast to previous techniques based on reachability from secret inputs (tainting), it achieves a more precise quantitative result by computing a maximum flow of information between the inputs and outputs. The technique uses static control-flow regions to soundly account for implicit flows via branches and pointer operations, but operates dynamically by observing one or more program executions and giving numeric flow bounds specific to them (e.g., "17 bits"). The maximum flow in a network also gives a minimum cut (a set of edges that separate the secret input from the output), which can be used to efficiently check that the same policy is satisfied on future executions. We performed case studies on 5 real C, C++, and Objective C programs, 3 of which had more than 250K lines of code. The tool checked multiple security policies, including one that was violated by a previously unknown bug.

Categories and Subject Descriptors D.2.4 [*Software/Program Verification*]; D.2.5 [*Testing and Debugging*]; E.4 [*Coding and Information Theory*]; G.2.2 [*Graph Theory*]

General Terms Languages, Measurement, Performance, Security, Theory, Verification

Keywords Information-flow analysis, dynamic analysis, implicit flow

1. Introduction

The goal of information-flow security is to enforce limits on the dissemination of information. For instance, a confidentiality property requires that a program that is entrusted with secrets should not "leak" those secrets into public outputs. Absolute prohibitions on information flow are rarely satisfied by real programs: if a sensitive input does not affect a program's output at all, it is better to simply omit it, and unrelated computations at different security levels should be performed by separate processes. 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, measures the bits a program actually reveals, and detects a violation if the measured flow exceeds the policy. The problem we address here is how to measure, by observing an execution of a program (dynamic analysis), how much information about a subset of its inputs (the "secret inputs") can be inferred from a subset of its outputs (the "public outputs"). The text of the program itself is always considered public, and other techniques must be used to prevent inferences from observable aspects of the program's behavior other than its output, such as its use of time or system resources. The measurement produced is a sound upper bound on the actual information flow, so that our technique can overestimate the amount of information revealed, but can never underestimate it. The bound applies only to the examined execution: other executions might reveal either more information or less.

Most previous research on quantitative information flow has focused on 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. We focus on a broader class of problems in which a flow of many bits may be acceptable (though a quantitative policy is still only applicable if the allowable flows are all less than the undesirable ones).

In some violations of information-flow policies, confidential data is exposed directly, for instance if the memory containing a user-provided password is not cleared before being reused by the operating system. A number of existing techniques can track such direct data flows. However, in many other cases information is transformed among formats, and may eventually be revealed in a form very different from the original input. Our research aims to soundly 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 our technique must account for *implicit flows* in which the value of a variable depends on a previous secret branch condition or pointer value.

Most previous approaches to information-flow program analysis are based on some kind 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

PLDI'08, June 7–13, 2008, Tucson, Arizona, USA.
Copyright © 2008 ACM 978-1-59593-860-2/08/06. . . $5.00

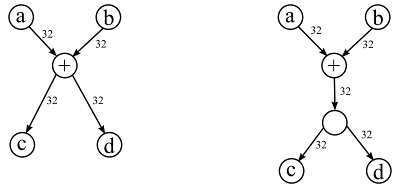

Figure 1. Two possible graphs representing the potential information flow in the expression `c = d = a + b`, where each variable is a 32-bit integer. The graph on the left permits 32 bits of information to flow from `a` to `c`, and a different 32 bits to flow from `b` to `d`. To avoid this, our tool uses the graph on the right.

the operands is. 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 the present work is to measure information-flow not 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. Then the maximum rate at which fluid can flow through the network corresponds to the amount of secret information the execution can reveal. According to the classic max-flow-min-cut theorem, this capacity also corresponds to the weight of a minimum cut: a set of edges whose removal disconnects the secret input from the public output, representing a set of secret intermediate values that (along with public information) determine the program's output.

The rest of this paper is organized as follows. Section 2 describes how to construct a flow network representing the propagation of secrets in a program execution, and Section 3 defines soundness and describes how to ensure it between results from different runs. Section 4 gives an implementation of the technique that operates at the instruction level. Then, Section 5 discusses efficiently computing the maximum flow in a large network, and Section 6 describes how a flow bound, once found, can be checked on future program runs. Section 7 discusses the situations in which our tool is most appropriate, and Section 8 evaluates it on confidentiality properties in a number of real applications. Finally, Section 9 surveys related research, Section 10 discusses future work, and Section 11 concludes.

2. Dynamic maximum-flow analysis

Our basic technique is to construct a graph that represents the possible flows of secret information through a program execution. This section describes that construction, including how to account for implicit flows, and how to assign capacities to edges in the flow graph.

2.1 Basic approach

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 giving how many bits of data they can hold. Nodes represent basic operations on those values, where the in-degree of a node is the operation's arity. For the case when the result of an operation is 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 (see Figure 1); this is also equivalent to giving a capacity limit on a node.

Copying a piece of data without modifying it does not lead to the creation of new nodes or edges, but because memory is byte-oriented, loads and stores of larger values are split into bytes for stores and recombined after loads. 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.

2.2 Implicit flows

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. These operations lead to indirect or *implicit* flows which do not correspond to any direct data flows. 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 index used in a previous store was not equal to 5. To account for such situations, a sound graph representation must have edges that represent all possible 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. The tool adds edges to connect each implicit flow operation to the outputs of the enclosed computation. 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 with edges to the result. For precision, the capacity of these implicit flow edges corresponds to the number of possible different executions: for instance, a two-way branch yields an edge with capacity one bit, while the capacity for a pointer operation such as an indirect load, store, or jump is as many bits as are secret in the pointer value. (Multi-way branches show up as either nested two-way branches or jump tables at the instruction level.)

To achieve soundness, it is sufficient to consider the entire program as being enclosed in this way: our tool's default behavior connects each implicit flow operation to the program's output. 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 annotations which mark a single-exit control-flow region and declare all of the locations the enclosed code might write to (see Figure 2 for examples). These annotations can be inferred using standard static analysis techniques; Section 8.6 describes a pilot study examining what is required. 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.

For precision and efficiency, the graph structures our tool constructs are somewhat more complex than simple edges from each implicit flow operation to each output. Each enclosure region has a distinguished node, and our tool adds edges from each implicit flow operation to that node, and then from that node to each output. For the enclosure of the entire program, our tool takes advantage of the time sequence in outputs by building a chain of nodes each corresponding to an output operation. Each implicit flow is connected to the then-current end of the chain, so that information leaked by an implicit flow can escape via any subsequent output one, but not an output that occurred earlier.

```
 1 /* Print all the "."s or "?"s,
 2    whichever is more common. */
 3 void count_punct(char *buf) {
 4     unsigned char num_dot = 0, num_qm = 0, num;
 5     char common, *p;
 6     ENTER_ENCLOSE(num_dot, num_qm);
 7     while (p = buf; *p != ’\0’; p++)
 8         if (*p == ’.’)
 9             num_dot++;
10         else if (*p == ’?’)
11             num_qm++;
12     LEAVE_ENCLOSE();
13     ENTER_ENCLOSE(common, num);
14     if (num_dot > num_qm) {
15         /* "."s were more common. */
16         common = ’.’; num = num_dot;
17     } else {
18         /* "?"s were more common. */
19         common = ’?’; num = num_qm;
20     }
21     LEAVE_ENCLOSE();
22     /* print "num" copies of "common". */
23     while (num--)
24         printf("%c", common);
25 }
```

Figure 2. 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 "`........`". As detailed in Section 2.4, our tool reports that this execution reveals 9 bits of information about the input.

2.3 Bit-capacity analysis

Subsections 2.1 and 2.2 described the structure of the graph our tool computes for a program execution, but to compute a maximum flow, each edge must also be labelled with a bound on the amount of information it can convey. To compute these bounds, our tool simultaneously performs a dynamic bit-width analysis to determine which of the bits in each data value might contain secret information. This analysis is essentially implemented as dynamic tainting, but at the level of bits. The analysis maintains, for every location in memory or a register, a shadow bit vector of the same size representing which data bits might be secret. For each basic program operation the analysis computes conservative secrecy bits for its results based on the secrecy bits of the operands. The amount of secret information that might flow through a value is bounded by the number of its bits that are marked secret. This analysis is very similar to the analysis that the Valgrind Memcheck tool [47] uses to track undefined values, so we were able to reuse much of its implementation (as did, independently, the Flayer tool [17]).

2.4 Example

As a concrete example of the techniques of this section, consider the code shown in Figure 2. 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 the when run on its own source the program prints 8 periods. Our tool measures that such an execution reveals 9 bits of information about the secret input: 1 bit from the selection of which character is more common, and 8 bits from the count. The corresponding minimum cut consists of 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).

This example shows the importance of several of the techniques introduced above. 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 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. The maximum flow computation also improves precision; without it, simple tainting would determine that all of the bits in the output might depend on the input, giving a bound of 64 bits. Though we chose this example as a simple illustration, similar situations occurred commonly in real applications such as those described in Section 8.

3. Soundness and consistency

The technique of Section 2 often gives a good bound based on only a single program execution. But from a theoretical perspective, the amount of information a program reveals should be defined in terms of multiple possible runs. This section first provides a more specific definition of what our technique computes, then describes how to merge flow graphs to compute a bound that is sound in that sense across multiple program executions.

3.1 Soundness for a dynamic analysis

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. 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. Throughout, we use the perspective of expressing information with a code that represents possible messages via bit strings of variable length.

Other quantitative information-flow analyses have commonly treated the secret to be protected as being drawn from a fixed (e.g., uniform) distribution. Though this is appropriate in some circumstances, such assumptions limit a technique's usability: in practice, the distribution of an input is often unknown, or worse, might be controlled by an adversary. Instead, 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 one assumes that the inputs are uniformly distributed, the expected information revealed is a very small fraction of a bit, since a zero divisor would almost never occur by chance (though when it does, it should be counted as revealing 32 bits). But 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.

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 an upper bound on the channel capacity.

The perspective of a fixed uniform distribution, 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.

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 our tool measures to reveal at most k bits, we imagine that Alice sends a string of up to 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 input $i \in I$ that Alice sends, our tool reports an information-flow bound $k(i)$. We will say that that result is sound if there is also a code 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 there to be many distinct outputs, none of which reveal much information. The precise characterization of this relationship is Kraft’s inequality, 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 [12].) Several 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.

3.2 Achieving consistency over multiple runs

As explained above, 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.

For instance, consider the final phase (lines 22-24) of the example program of Figure 2, 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: $\sum_{n=0}^{255} 2^{-\min(8,n+1)} = \frac{503}{256} > 1$.

We have seen 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.

In outline, this graph combining process merges all the edges that correspond to the “same” program location. More precisely, it 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 [4]. 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”.

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.

4. Machine-level implementation

We have implemented the information-flow analysis described in the previous sections as a dynamic binary analysis for executables on Linux/x86 systems, called Flowcheck (`http://people.csail.mit.edu/smcc/projects/secret-flow/flowcheck.html`).

4.1 Dynamic instruction rewriting

Our tool instruments a program by dynamically rewriting its instruction stream, using the Valgrind framework [40]. Valgrind translates each basic block of instructions into a simple compiler-like intermediate representation; our tool adds instrumentation operations in that format; and then Valgrind translates the IR back into x86 instructions for execution. This translation insulates our analysis from most of the complexities of the large x86 instruction set: Valgrind automatically handles features such as complex addressing modes, implicit operands, string instructions, condition codes, and conditional moves, which require special treatment in tools that operate directly on instructions [10]. Valgrind’s automatic register allocation also makes it easier to insert instrumentation operations.

One architectural complexity of the x86 that is not abstracted by Valgrind is the presence of overlapping registers: for instance, the 16-bit register `%dx` consists of the lower-order bits of the 32-bit register `%edx`. In order to be able to treat each register as distinct, we have changed Valgrind's translation of such sub-registers so that instructions that access them instead read or write from the full register, selecting the relevant portion using bitwise operations.

4.2 Graph construction by value tagging

To build the flow graph described in Section 2, the tool associates a positive integer, which we call a *tag*, with each execution-time value that might contain secret information; values that are not reachable from the secret input have a tag of 0. These tags represent the identities of nodes in the flow graph; a tag is associated with each register, and each byte in memory. The tags are maintained in parallel with the secrecy bit-masks described in Section 2.3; if a value's tag is 0, its bit-mask is necessarily all public, and it is omitted from the graph. If at least one operand of a basic operation has a non-zero tag, the instrumentation code for the operation assigns a fresh tag for the result of the operation, and creates edges linking the inputs to the result.

The representation of edges depends on whether the graph-combining feature of Section 3.2 is in use. If every edge is to be considered unique, they do not need any in-memory representation: each edge is output to the graph immediately, as an ordered pair of node tags. In this mode, the memory usage of the tool is bounded by a multiple of the memory usage of the original program: it does not grow as the graph becomes larger. On the other hand, if edges are to be combined based on their program locations, it is more efficient to keep a representation of each class of equivalent nodes in the tool's memory. However, it is not necessary to retain the entire original graph: instead, all that is needed is the combined graph (whose size is bounded by the program size), and information about those nodes that still correspond to values in registers or memory. The tool implements an algorithm similar to mark-and-sweep garbage collection to identify when tags can be reclaimed. These techniques differ from previous implementations because of our constraint that memory usage must not grow with the length of a program's execution. For instance, Redux [39] builds a similar graph with an in-memory linked data structure, which facilitates computing a backward slice from the output but is less scalable.

4.3 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 tag 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 a flow to 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.

4.4 Other issues

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 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 Section 8 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.

5. Efficient maximum-flow

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, and unconditionally efficient algorithms with the potential to be precise. Empirically, the later approach seems to work better.

5.1 Exact 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 [11], 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. What is needed is an algorithm that is likely to run in close to linear time on the graphs that arise in flow analysis.

Flow graphs often have many nodes connected in series or parallel, for which flow computation is linear-time, so we explored the use of SPQR trees, an incremental graph representation that can capture series-parallel structure [3]. In our case studies, the graphs have a mixture of series-parallel and non-series-parallel structure, with neither one dominant. For instance, in `bzip2`, the largest non-series-parallel structure represents 16% of the graph size over a range of input sizes, and this constant fraction of the graph still requires super-linear processing time. Therefore, while SPQR trees capture some useful regularities, they do not appear sufficient to allow the technique to scale to very large graphs. More details of these experiments can be found in a technical report [33] and the first author's thesis [31].

5.2 Graph collapsing by code location

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 3.2: 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).

The size of the collapsed graph grows not with the runtime of the original execution, but with its code coverage; since the latter

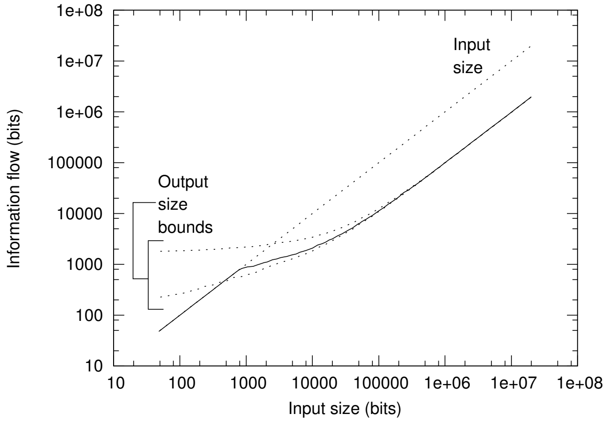

Figure 3. The amount of information revealed in compressing files with `bzip2`, as measured by our tool (note log-log scale). The solid line shows the flows measured by our tool, in bits. The dotted lines represent other functions that would be expected to bound the flow: The straight line through the 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.

tends to plateau (and is bounded by the program size), much longer executions can be analyzed. Graph collapsing can potentially reduce precision, for instance if two calculations in a loop had negatively correlated flows on different iterations. However, we have not observed this to be a practical problem (graph collapsing was enabled in all of our case studies).

5.3 Maximum-flow performance in practice

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, compressing inputs marked as entirely secret. `bzip2` was not intended 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 4.3. (Section 8 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 3 compares the flow measured by our tool to the expected bound, which 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 to the lazy range operation implementation and graph collapsing techniques. 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.

6. Checking a flow bound

Once the main flow measurement technique discussed in this paper has been used to determine the amount of information a program reveals under testing, users of a program would also like to check that the same bound always holds as the program is used in deployment. Checking a bound is a simpler and faster than discovering it. Section 6.1 describes how to compute a cut of the flow graph from a maximum flow, then Sections 6.2 and 6.3 give two checking techniques that use such a cut.

6.1 Computing a minimum cut

A cut (an *s-t cut*, to be precise) 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. There is duality between flows and cuts, captured by the classic max-flow-min-cut theorem: the value of any flow is bounded by the capacity of any cut, and the maximum flows are those with the same value as the minimum-capacity cuts, since there is no way to augment them [11].

Once a maximum flow has been discovered, our tool computes a cut by first enumerating the nodes on the source side of the cut by depth-first search: they are the nodes that are reachable from the source along a path in which each edge has excess capacity. Then, the cut edges are those that connect nodes reached in the DFS to nodes not reached.

On the analyzed run(s), the edges of the cut carried an amount of data equal to our tool's estimate of the amount of information the execution revealed, and all information flows from the secret inputs to the public outputs passed through them. On future executions, the amount of data corresponding edges carry will be a sound measure of the information revealed, as long as no other flows occur. Therefore, a static representation of the edges can be used to efficiently check when an analogous policy holds on future executions, reducing detection to a reachability check; Sections 6.2 and 6.3 describe specific implementations. Such checking will soundly detect any leaks other than, or larger than, those allowed by the cut; the price paid for reduced overhead is that novel leaks may not be measured precisely.

6.2 Tainting-based checking

Checking that no secret information reaches the output other than across a given cut is a simple tainting problem. We have implemented this as an alternate mode of our tool, reusing the bit-level tainting analysis described in Section 2.3. The cut edges correspond to annotations that clear the taint bits on data, while simultaneously incrementing a counter of information revealed. If any other tainted bits reach the output or an implicit flow operation, they are conservatively counted in the same way, and the location reported: enclosure regions are still required. The runtime overhead of this approach is comparable to that of Memcheck: between 10 and 100 times the uninstrumented execution time.

6.3 Output-comparison checking

An even more efficient 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 is less than any binary-level dynamic tainting system, and using two copies can take advantage of multiple processors.

A simpler version of this technique (without a cut, for checking only complete non-interference) has been implemented in an operating-system-level tool called TightLip [57]. We previously suggested the extension to quantitative policies by sending information at a cut, but in a theoretical context to convert an information-flow property into a safety property that could be more easily proved by induction [34].

7. Discussion

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 a complete security policy—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 or less than $40,000?", since it carries the same amount of information as an acceptable query like "Was the average income of Boston residents more or less 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 Section 6, other techniques can be used to check for violations of a policy on future executions, such as after a system has been deployed.

		# of	secret
Program	KLOC	libraries	data
KBattleship	6.6	37	ship locations
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 4. Summary of the programs examined in the case studies of Section 8. 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.

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; 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 2.4, 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. Case studies

To learn about the practical applicability of our tool, we used it to test a different security property in each of five open-source applications. The programs and the secret information protected are summarized in Figure 4. 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.

To obtain precise results, all of the programs required enclosure region annotations. Section 8.6 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 local code refactorings, such 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.

8.1 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 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 [35]) 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.

8.2 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 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 [49].) 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.

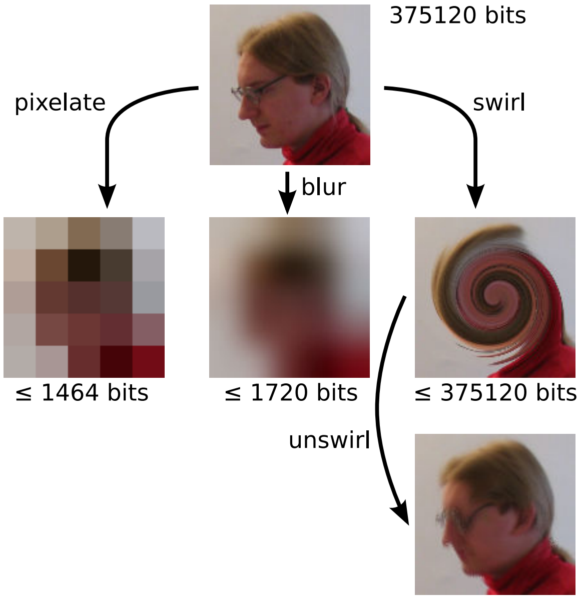

Figure 5. Image transformations vary in how much information they preserve. Our tool verifies that pixelating (left) or blurring (middle) the original image (top, 375120 bits), reveals only 1464 or 1720 bits respectively. By contrast, the bound our tool finds for the information revealed by a twisting 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).

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.

8.3 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 information would prevent the original face from being reconstructed.

Figure 5 shows an original 125-pixel square image, which is represented by 375120 bits in an uncompressed PPM 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).

8.4 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 tainting 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.

8.5 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 [55]. 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 [24] 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`.

8.6 Inferring enclosure regions

Enclosure regions, introduced in Section 2.2 (and illustrated in Figure 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 [7, 46]: in our case, the annotation requires

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

Figure 6. Summary of the results of the static analysis discussed in Section 8.6 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).

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. [44].

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 [38]. 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 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 6. (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 6 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.

9. Related work

Our technique combines some of the attributes of static analyses (including type systems) that check programs for information-flow security ahead of time, and of dynamic tainting analyses that track data flow in programs as they execute.

9.1 Static information-flow

Static checking aims to check the information-flow security of programs before executing them [13]. The most common technique uses a type system, along with a declassification mechanism to allow certain flows. It is also possible to quantify information flows in a static system, though this has been difficult to make practical.

Despite advances such as selective declassification [20, 36], barriers remain to the adoption of information-flow type checking [53] extensions to general purpose languages [35, 48, 27]. 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 [35] (closest are the poker game of [2] and the email client of [25]) or Flow Caml [48] dialects. 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).

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 [22, 53]. 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 placed, a program can pass a type check but still leak arbitrary information. The minimum cuts described in Section 6 could be used to choose the placement of declassification annotations, since they would be a minimal interface between secret and declassified data. However, we do not envision them to be a trusted representation of the information flow policy: rather, the policy is a numeric flow bound, and a cut is an untrusted hint to assist enforcement.

Quantitative measurements based on information theory have often been used in theoretical definitions of information-flow security [23, 16, 28]. Clark et al.'s system for a simple while language [9] 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. A formula giving precise per-iteration leakage bounds for loops [29] may be difficult to automate. Our technique's results reflect the number of iterations that occur on a particular execution.

9.2 Dynamic tainting

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, it is important to examine the flow of information through calculations, including comparisons and branches that cause implicit flows. Several recent projects dynamically track data flow for data confidentiality and integrity, but 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 [19], and Gat and Saal propose reverting writes made by secret-using code [21] to prevent flows. 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 [14]. However, these techniques are described as architectures for new systems, 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 [8] 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. [30] 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. However, it appears that other implicit flows are simply ignored, and their case studies do not involve implicit flows. DYTAN [10], 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`, but do not measure whether the output was tainted.

Accounting for all implicit flows requires static information, as provided by enclosure regions in our system. Several projects have combined completely automatic static analyses with dynamic checking: the key challenge is making such analysis scalable and sufficiently precise. The RIFLE project [52] is an architectural extension that 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 [37, 6] apply to Java programs, making static analysis somewhat easier: their experimental results show low performance overheads, but do not measure precision. None of these tools enforce a quantitative security policy.

In an earlier technical report [32], we presented a tainting-based quantitative information-flow analysis that was the predecessor to the implementation described here. That system had no maximum flow or minimum cut analysis; instead it used manual annotations, called "preemptive leakage" annotations, that played the role of a (not necessarily minimal) cut. Enclosure regions in that system were also manually supplied, and were unsound because they propagated tainting only to locations that were dynamically accessed. More recently, we gave a soundness proof [34] for a simple formalized system that can be seen as modelling our tainting based implementation, with enclosure regions modified to be sound by specifying outputs in the same way those in the present paper do. The simulation proof technique used there could be extended to the present system by treating the minimum cut corresponding to a maximum flow as a preemptive leakage annotation. As long as all possible information flows are captured as edges in the graph, any cut in that graph represents a sound flow bound. (The leakage annotations in [34] are static; we plan to extend that result to a cut that is dependent on the program input by formalizing of the soundness definition of Section 3.)

Restrictions on information flow can also be enforced by an operating system. Traditional mandatory access control (MAC) techniques [15] at the granularity of processes and files are too coarse for the examples we consider. A new operating system architecture with lightweight memory-isolated processes, such as the "event processes" of the Asbestos system [18] or similar mechanisms in HiStar [58], is more suitable for controlling fine-grained information flow, but is not compatible with existing applications. Like our technique's enclosure regions, Asbestos event processes provide isolation of side effects, but they are implemented using hardware memory protection.

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 [42], 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.

The most active area of 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 [41] is based on the same Valgrind framework as our tool, while other researchers have suggested using more optimized dynamic translation [26, 45], source-level translation [56], or novel hardware support [50] 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" [54]) or SQL statements [43].

10. 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.

10.1 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 [1].

10.2 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 2.3 has a close static analogue [5]. 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.

10.3 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 [54, 43]. 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 [51]. 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.

11. Conclusion

We have presented a new approach for determining how much information a program reveals, based on the insight that maximum flow is a more precise graph model of information propagation than reachability (as implemented by tainting) is. Using a practical quantitative definition of leakage, the technique can measure the information revealed by complex calculations involving implicit flows. By applying that definition with an instruction-level bit tracking analysis and optimized graph operations, it is applicable to real programs written in languages such as C and C++. In a series of case studies, our implementation 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.

Acknowledgments

This research was supported in part by DARPA under contracts FA8750-06-2-0189 and HR0011-06-1-0017, and by an NSF grant CCR-0133580.

References

[1] R. Ahlswede, N. Cai, S.-Y. R. Li, and R. W. Yeung. Network information flow. *IEEE Transactions on Information Theory*, 46(4):1204–1216, 2000.

[2] A. Askarov and A. 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.

[3] G. D. Battista and R. 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.

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

[5] M. Budiu, M. Sakr, K. Walker, and S. C. 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.

[6] D. Chandra and M. 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.

[7] J.-D. Choi, M. Burke, and P. 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.

[8] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding data lifetime via whole system simulation. In *13th USENIX Security Symposium*, pages 321–336, San Diego, CA, USA, August 11–13, 2004.

[9] D. Clark, S. Hunt, and P. 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.

[10] J. Clause, W. Li, and A. 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.

[11] T. H. Cormen, C. E. Leiserson, and R. 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.

[12] T. M. Cover and J. A. Thomas. *Elements of Information Theory*. John Wiley, 1991.

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

[14] D. E. R. Denning. *Secure Information Flow in Computer Systems*. PhD thesis, Purdue University, May 1975.

[15] Department of Defense Computer Security Center. *Trusted Computer System Evaluation Criteria*, August 1983. CSC-STD-001-83.

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

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

[18] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. 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.

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

[20] E. Ferrari, P. Samarati, E. Bertino, and S. 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.

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

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

[23] J. 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.

[24] M. 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`.

[25] B. Hicks, K. Ahmadizadeh, and P. 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.

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

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

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

[29] P. 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.

[30] W. Masri, A. Podgurski, and D. 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.

[31] S. McCamant. *Quantitative Information-Flow Tracking for Real Systems*. PhD thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, June 2008.

[32] S. McCamant and M. 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.

[33] S. McCamant and M. 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.

[34] S. McCamant and M. 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.

[35] A. 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.

[36] A. C. Myers and B. 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.

[37] S. K. Nair, P. N. Simpson, B. Crispo, and A. 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.

[38] G. C. Necula, S. McPeak, S. Rahul, and W. 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.

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

[40] N. Nethercote and J. 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.

[41] J. Newsome and D. 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.

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

[43] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. 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.

[44] J. Qian, B. Xu, and H. 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.

[45] F. Qin, C. Wang, Z. Li, H.-S. Kim, Y. Zhou, and Y. 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.

[46] A. Sălcianu and M. 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.

[47] J. Seward and N. 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.

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

[49] S. Smith and M. 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.

[50] G. E. Suh, J. W. Lee, D. Zhang, and S. 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.

[51] G. T. Sullivan, D. L. Bruening, I. Baron, T. Garnett, and S. 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.

[52] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. 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.

[53] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, December 1996.

[54] L. Wall and R. L. Schwartz. *Programming Perl*. O'Reilly & Associates, 1991.

[55] D. P. Wiggins. *Security Extension Specification*. X Consortium, Inc., November 1996.

[56] W. Xu, S. 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.

[57] A. R. Yumerfendi, B. Mickle, and L. 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.

[58] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. 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.