Computer Science and Artificial Intelligence Laboratory

Technical Report

MIT-CSAIL-TR-2007-057 December 10, 2007

Quantitative Information Flow as Network Flow Capacity

Stephen McCamant and Michael D. Ernst

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

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.

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 of 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 whom I'm meeting, 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.

A slightly abridged version of this paper has been submitted to PLDI 2008.

The approach of *quantitative* information-flow security expresses a confidentiality property as a limit on the number of bits 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 (designated secret) can be inferred from a subset of its outputs (designated public). 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 sometimes overestimate the amount of information revealed, but never underestimate it. Some programs reveal only a negligible amount of information on most executions, and this can be measured by quantitative policy that allows only a small fraction of a bit: for instance, the amount of information revealed by an unsuccessful login attempt, if an attacker had no previous knowledge of the password. Most previous research on quantitative information-flow has focused on very small flows, but a quantitative policy can be applied to any situation in which secret information is partially revealed. We focus on a broader class of problems in which the amount of allowed flow can be arbitrarily large, as long as the all the allowed flows contain less information than the flows that must be prohibited.

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 be eventually 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 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 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 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 net-

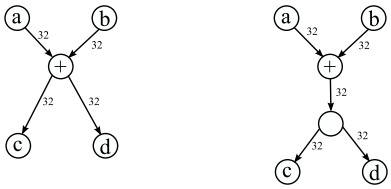

Figure 1. Two possible graphs representing the potential information flow in the expression `c = d = a + b`, where each variable is a 32-bit integer. Our tool uses the graph on the right, which unlike the one on the left excludes the possibility of 32 bits of information flowing from `a` to `c`, and a different 32 bits flowing from `b` to `d`.

work 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 a minimum set of secret intermediate values that (along with public information) determine the program's output.

A dynamic security analysis can be used interactively, to test whether a policy holds and debug violations, or non-interactively, to catch violations in production. In this work, we concentrate on a debugging-style tool that enforces a general policy and gives detailed information about information leaks, in the same way a tool like Purify [24] or Valgrind Memcheck [42] is used to debug memory safety violations. An interactive tool is also best for developing policies that will later be enforced non-interactively. Section 6.3 discusses different techniques for a production context where error reporting is not important and performance overhead is. Most convenient would be a static tool that checked a policy over all possible executions at once, but static information-flow checking is far from scaling to the large pre-existing programs we are most interested in (Sections 8.1, 9.1).

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 discusses how to ensure consistency between results from different runs. Section 4 gives an implementation of the technique that operates at the binary 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 enforced on future program runs. Section 7 evaluates our tool on confidentiality properties in a number of real applications. Finally, Section 8 surveys related research, Section 9 discusses future work, and Section 10 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 are similar to circuits: edges in the graph represent values, and nodes represent basic operations on those values; the in-degree of a node corresponds to the operation's arity. However, for efficiency, the graph represents byte or word-sized operations; edges have capacities giving how many bits of data they can hold. For the case when the result of an operation is used in more than one subsequent operation, our tool adds an addition 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

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

Figure 2. Summary of the results of the static analysis discussed in Section 2.3 to compute which locations a code region (containing an implicit flow) might modify. Overall, the automatic analysis found 72% ("found" column) of the outputs used by the rest of the analysis ("outputs" column).

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 inputs, and a *sink* node representing all outputs.

2.2 Implicit flows

General programs are more complex than circuits because of operations such as branches, arrays and pointers that allow current data to affect which operations will be performed in the future or what their operands will be. In an information flow context, these operations are said to 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, our technique must add edges that represent for all possible implicit flows. One simple approach is to add edges from implicit flow operations (e.g., branches) directly to the sink (output node) of the graph. The capacity of such edges corresponds to the number of possible different executions: for instance, a two way branch corresponds to an edge with capacity one bit, while a pointer operation such as an indirect load, store, or jump might reveal as many bits as are in the pointer value. While sound, this approach can be imprecise, so our technique builds on it in two ways.

First, rather than directing the global leaks from implicit flows directly to the sink, it builds a chain of nodes corresponding to each output operation the program performs, so that the information leaked by an implicit flow can escape via the next output operation, or any subsequent one, but not an output that occurred before the implicit flow did. Second, the technique can use what we call *enclosure regions* to further constrain where implicit leaks can flow. An enclosure region is a single-exit control-flow region of the code, along with a list of all of its outputs, i.e., which parts of the program state it might modify. When an implicit flow happens inside an enclosure region, the leak is directed not to the final program output, but to the outputs of the region. The next subsection describes how our tool constructs enclosure regions.

2.3 Determining enclosure outputs

An enclosure region is a static program annotation, since it must account for all the possible side effects of a code region that generally includes branches. The key task in constructing an enclosure region is to list all of the locations that a code fragment might write to. Though conceptually straightforward, the practical difficulty of this depends on the language the program is written in. Since our tool targets C and related languages, whose memory usage can be rather unstructured, doing a full job is complex.

We have implemented the inference of enclosure regions as a simple C source code analysis using the CIL framework [36], which

scans the program function by function keeping track of which lvalues are assigned to within each branch or loop. Figure 2 summarizes the results of our simple enclosure region inference tool on four programs (the remaining case studies from Section 7 are written in C++ or Objective C, so our static tool cannot parse them). Each output mentioned in an enclosure region is counted separately. Overall, 72% of the needed variables could be found even by this very simple analysis. The columns labelled "need expansion" and "need inter-procedural" represent the two further features the tool would need to find the remaining variables. *Expansion* refers to 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, either because the index expression was not constant throughout the region, or because the index value might itself be secret. An *inter-procedural* analysis would be needed to find some enclosure regions where the side effects to be enclosed are in a called function, when it would not be enough to place the enclosure region in the callee, either because there are implicit flow operations in the caller or not all uses should be enclosed. As an orthogonal classification, the final column "need length" counts the number of outputs where the area being written to was a dynamically allocated array, and the enclosure region required a static 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.

2.4 Bit-capacity analysis

Previous subsections 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 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. Every location in a memory or a register is shadowed by a location of the same size containing a bit vector representing which data bits might be secret. Each basic program operation 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 then bounded by the number of its bits that are marked secret. This analysis is very similar to the analysis that the Valgrind Memcheck tool [42] uses to track undefined values, so we were able to reuse much of its implementation.

3. Input spaces and consistency

In practice, the technique of the previous section can often give a good bound based on only a single program execution, but to be confident that the results will generalize, it is best to analyze a set of representative executions. This section first explains why it makes a difference what the possible inputs to a program are, and then discusses how to get consistent results by analyzing several executions together.

3.1 Why distributions matter

In information theory, there is no definition of the "amount of information" in a string of bits considered on their own. The amount of information (*entropy*) in a piece of data depends on the distribution from which it is drawn. This can be an obstacle for a program analysis; for instance, it would be natural to take the distribution of a program's inputs as an input to the analysis, but such a distribution is usually not available. A static analysis has only the program itself, and a dynamic analysis can observe only some small subset of all the inputs that might be supplied. And even worse than being unknown, the input distribution may chosen by an adversary. A password checker, or any other deterministic program with more than one distinguishable output, can reveal at least a bit on every input if the space of possibilities is small. For instance, an administrator might create users whose password is one of two choices prearranged with an unprivileged user.

An alternate perspective on the flow measurement technique described in Section 2 is that it determines from the execution a compressed form of the secret input that is enough (along with the public inputs) to recreate the program's public output. For instance, a sequence of bits from branch conditions is a compact coding of the predicate on the input that those branches check. This choice of perspectives is completely general: for a set of bit strings, there is a natural isomorphism between a probability distribution on those strings, a compression function on those strings that is optimal for the distribution, and a measurement of the information in each string that is the size of the result of the compression.

As another example, suppose that a program that can recognize repeated characters in its secret input is presented with the string of ASCII characters `aa`. (This example is slightly simplified from a behavior we observed in the first pass of the `bzip2` compression tool, discussed further in Section 5.4.) How many bits of information are in the string `aa`? One simple answer would be to count each character as 8 bits, for a total of 16; this corresponds to measuring the flow at the program input. Alternatively, if the program checks that the two characters of its input are the same and then doesn't otherwise use the second `a`, another measurement would be 9 bits, corresponding to measuring the 1 bit result of the comparison instead of the value of the second character. Both of these measurements are equally correct; they both extend naturally to sound measurements of the information in any string. (For a string of $n > 0$ characters of which $r < n$ are identical to the preceding character, they would measure $8n$ or $9n - 8r - 1$ bits respectively.) For this example, our tool would measure 9 bits, but the choice of which measurement is smallest depends on the input. (There other even smaller measurements of this input that our tool happens not to consider: for instance, one in which `aa` conveys 1 bit and any other string conveys $8n + 1$ bits.) However, it could give misleadingly small results to measure each execution by a different standard; the measurements of different executions of a program should be consistent.

3.2 Combining runs

A dynamic analysis is limited to examining only a finite number of program executions, but given that constraint, we would still like the tool's results to generalize as much as possible to unobserved executions. The best predictor of generalizing to future executions is generalizing over existing executions, so we would the tool to a produce single consistent set of measurements for a number of different runs. It does so by combining the graphs from multiple executions, adding the edge capacities, and computing a total flow on the combined graph.

Combining flow graphs from multiple runs of a program requires identifying which parts of each graph correspond to the "same" program locations. Our tool does this by labelling each edge with a value that includes both a static location (i.e., instruction address), and a 64-bit hash of the calling context (stack backtrace), computed in a similar way to Bond and McKinley's probabilistic calling context [4]. Any number of so labelled graphs can be combined by identifying edges with the same label (adding their capacities), and unifying all of the nodes the various copies of an edge are incident upon, which can be done almost-linear time with a union-find structure.

It is easy to see that 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. However, the converse does not hold: the bound computed for the combined graph can be larger than the sum of the individual bounds. Intuitively speaking, the flow in the combined graph is limited only by bottlenecks that appear consistently in each original graph.

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.

4.1 Dynamic instruction rewriting

Our tool instruments a program by dynamically rewriting its instruction stream, using the Valgrind framework [38]. Valgrind translates each basic block of instructions into a simple compiler-like intermediate representation; our tool adds instrumentation operations and calls 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: 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], are handled automatically. Valgrind's automatic register allocation also makes it easier to insert instrumentation operations.

One architectural complexity of that 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 Value tagging

To build the flow graph described in Section 2, the tool associates a positive integer, which we call a *tag*, with each value during execution 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. 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's needed is the combined graph, 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 the need to minimize memory usage for long-running programs. For instance, Redux [37] 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 Large-region operations

Because enclosure regions can mention entire arrays or other large data structures, the tool often wishes to represent the fact that a piece of information might flow to any byte in a large memory region. However, it would be too slow to do this by operating on 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 on each iteration would lead to quadratic runtime cost. Instead, the tool performs operations on large memory regions lazily, by maintaining a limited-size set of region descriptors, each of which describes a large range of contiguous memory locations, perhaps with another limited-size list of addresses excepted. Operations such as a flow to an entire region can be recorded just by modifying the descriptor, and operations on single addresses can be marked as exceptions. However, if a region accumulates too many exceptions, it is either shrunk to exclude them (if they are all near the beginning), or eliminated.

4.4 Other issues

Because the analysis operates at the binary level, all of the libraries that program uses are included automatically. Inputs and outputs are recognized based on system calls, such as `read` and `write` respectively; memory-mapped I/O is not supported, though doing so would not be difficult because every memory operation is already instrumented. 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. 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 7 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 this 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 a fairly well-structured, so specialized optimizations are both necessary and possible.

5.1 Asymptotic performance

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. Therefore a more specialized flow algorithm is called for. The flow graphs produced by our technique have a number of special features that could guide the choice of an algorithm. Because vertices and edges are added at the same time, the graphs are sparse; i.e., $E = O(V)$. The capacities of edges are all small integers; e.g., no more than 32 if the program only uses word-sized operations. Intuitively, the graph from a long program execution will be deep but not very wide: the length of a path from the source to the sink is unbounded, but the

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

Figure 3. Experimental evaluation of SPQR trees for representing flow graphs. An SPQR tree is an efficient representation for maximum flow computation if the size of its largest R node (last column, measured in vertices) is small.

number of vertices that are in use at any moment is bounded by the size of the original program's memory.

In theory, this last "narrowness" property is be sufficient to give an algorithm that is linear in the execution length, since for any flow graph with at most k outputs, there is a bounded-size graph that allows the same flows. However, the best upper bound we have found on the size of such a mimicking graph is 2^{2^k} vertices [22, 6], clearly impractical if k is the size of memory. A related approach is to bound the treewidth of a flow graph: graphs with bounded treewidth can be hierarchically decomposed in a way that again gives a linear-time maximum flow algorithm [22]. We suspect that the flow graphs produced by our tool have small treewidth, though we have not been able to experimentally verify this; the well-known algorithms that are efficient for fixed treewidth k all apparently have exponential dependencies on k that make them impractical for treewidths as small as 4 (e.g, [3]). However, the next subsection gives a further specialization of this idea that is within the realm of experiment, using the class of series-parallel graphs, whose treewidth is at most 2.

5.2 SPQR trees

A series-parallel graph is one that can be formed using only the operations of series and parallel composition familiar from electrical circuits. The maximum flow in a series-parallel graph can be computed easily, since series and parallel composition correspond to the operations of minimum and addition on the maximum flows of the subgraphs. Our flow graphs are not generally series-parallel, but they often contain large series-parallel portions, which suggests the use of a data structure called an SPQR tree. An SPQR tree is a tree that represents a hierarchical decomposition of a directed acyclic graph with exactly one source and sink (an *s-t DAG*). The nodes in the tree are of four kinds labelled S, P, Q, or R: S nodes represent a series composition of their children, P nodes represent a parallel composition, Q nodes are leaves that represent single edges, and R nodes represent any composition that is not series-parallel. An SPQR tree can be constructed efficiently, and maintained incrementally as vertices and edges are added [2]. Depending on the structure of the graph, the tree can range between having no R nodes (for a series-parallel graph), and representing the entire graph by a single R node with all Q nodes attached directly.

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

To test the efficacy of SPQR tree decomposition on our flow graphs, we computed SQPR trees for them using the batch algorithm from the AGD library [21]. (OGDF [7], the successor library to AGD, also includes incremental SPQR tree construction and is open-source, but was not available when we started this project.) The results are shown in Figure 3 (for OpenSSH, the flow graph was too large for the SPQR tool to process; for ImageMagick we used a smaller image size). Series-parallel structure occurs across all the programs, as shown by the large number of S and P nodes. However, most of the trees also had a large R node at the root, indicating that the high-level structure of the flow is not series-parallel. Comparing the two runs of `bzip2`, notice that the R node at the root grows as a constant fraction of the graph size; if this is the general pattern, it means that an SPQR tree will not provide an asymptotic performance advantage for this program. Therefore, while SPQR trees capture some useful regularities, they do not appear sufficient to allow the technique to scale to very large graphs.

5.3 Graph collapsing by code location

Since we have not found an efficient way to compute the maximum flow in large graphs exactly, the next best approach is to simplify the graph in a way that is sound and does not greatly increase the flow. We considered some general-purpose graph operations, but 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. In fact, our tool is able to do 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 only on their (context-insensitive) code location. With either variant, the size of the collapsed graph grows not with the runtime of the original execution, but with its code coverage; since the latter tends to plateau, much longer executions can be analyzed. A disadvantage of this collapsing technique is that it undoes some of the properties that make computations on the original graph easy: the summed capacities on edges can be unbounded, and collapsing can introduce cycles. For instance, collapsed graphs cannot be represented with SPQR trees.

5.4 Empirical results

We measured the scalability of our tool by testing it on `bzip2`, a general-purpose compression tool based on block sorting. `bzip2`, with its entire input marked secret, is not a very realistic example for security analysis, since no detailed measurements are needed to determine that its output contains the same information as its inputs. 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. 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 4 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 latter is somewhat uncertain, because part of the binary output format consists of fixed headers, and the commentary printed to the terminal is only partially input-dependent; so we give lower and upper bounds. The results match our expectations: very small inputs cannot be compressed by `bzip2`, but for inputs that

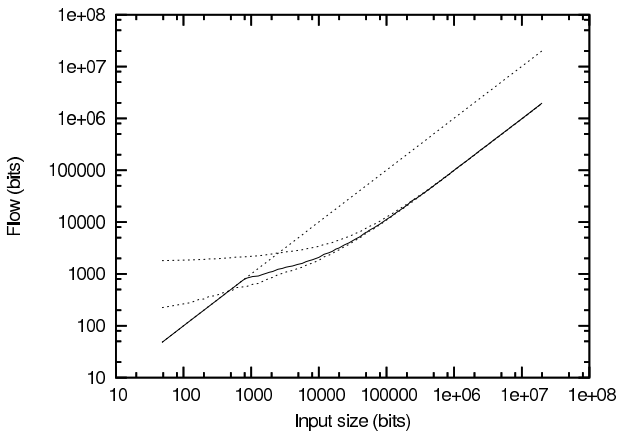

Figure 4. The amount of information revealed in compressing files with `bzip2`, as measured by our tool (note log-log scale). The solid line shows the flows measured by our tool, in bits. The dotted lines represent other functions that would be expected to bound the flow: The straight line through the origin represents the input size. The two curved lines (which are close to linear but do not pass through the origin) represent 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.

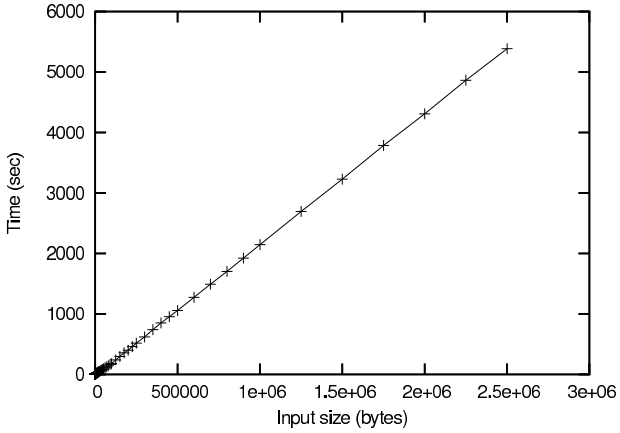

Figure 5. The running time of our analysis, with graph collapsing, on `bzip2` running on a range of input sizes. For a 2.5 megabyte file, the tool took about 1.5 hours. For small inputs, the performance is also linear, but dominated by a constant startup time of about 4 seconds.

`bzip2` can compress, the size of the compressed output is the best estimate of the amount of information the program processes.

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, as shown in Figure 5. 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. When tracing code that is not operating on secrets, no graph is constructed, so the tool's 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 technique discussed in this paper has been used to determine the amount of information a program reveals under testing, one would also like to enforce that the same bound always holds as the program is used in deployment. Checking a policy is a simpler problem than discovering it, so a more optimized implementation technique can be used. After describing how to compute a cut of the flow graph from a maximum flow, we will give two checking techniques that use such a cut.

Intuitively, a cut is a way of dividing a flow graph into two pieces, one containing the source and the other the sink (a more precise term is an *s-t cut*). Formally, it is convenient to define a cut as the set of nodes that lie in the half containing the source, but we are often interested in the set of edges that cross from that set to its complement; their removal disconnects the source from the sink. The capacity of a cut is the sum of the capacities of these edges. There is 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]. It is often helpful to think about a maximum flow computation as instead finding a minimum cut: for instance, the graph combining technique of Section 3.2 can be thought of as restricting the choice of cuts to those that are consistent across the combined graphs.

6.1 Computing a minimum cut

Once a maximum flow has been discovered, a corresponding minimum cut can be computed by finding the set of nodes reachable from the source along a path in which each edge has excess capacity. This set of nodes can be computed with a depth-first search; then the edges of the cut are those that connect reached and unreached nodes.

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 likely be a good estimate 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; enforcement is reduced to a tainting-style reachability check. Our tool reports the source code line numbers to which the cut edges correspond; since cuts tend to have only a few edges, and most lines contain only one assignment, it is easy to use these results to write static annotations. If the static annotation language is context-insensitive, the best results are obtained by collapsing the flow graph according to static edge locations before computing the cut.

6.2 Tainting-based checking

When a cut is supplied, checking that no secret information reaches the output other than across the cut is a simple reachability problem, so one obvious approach is to use dynamic tainting. We have implemented this as an alternate mode of our tool, reusing the bit-level tainting analysis described in Section 2.4. 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: enclosure regions are still required. The runtime overhead of this approach is

Program	KLOC	# of libraries	secret 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 6. Summary of the programs examined in the case studies of Section 7. The program sizes, measured in thousands of lines of code (KLOC), include blank lines and comments, but do not include binary libraries (3rd column, measured with `ldd`) that were included in the analysis but not directly involved with the security policy.

comparable to that of Memcheck: between 10 and 100 times the uninstrumented execution time.

6.3 Output-comparison checking

If no support for debugging policy violations is required, an even more efficient checking technique can be based on running two copies of a program. The basic idea is to run two copies of a program in parallel, 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 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 2x 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 [52]. The extension to quantitative policies by sending information at a cut was suggested in a previous paper [33], but in a theoretical context to convert an information-flow property into a safety property that could be more easily proved by induction.

7. 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 6. In each program the sensitive information we would like to protect 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 regions beyond those that could be inferred by the automatic tool of Section 2.3; for the programs written in Objective C or C++, we supplied all the enclosure regions by hand. Because of limitations in our current syntax for specifying such regions, this sometimes required local code refactorings, such introducing a temporary value to hold a return value. We spent about as much time writing such annotations as compiling and configuring the programs to run on our system and developing test cases for the relevant policies.

7.1 KBattleship

In the children's game Battleship, successful play requires keeping secrets from one's opponent. Each player secretly chooses locations for four rectangular ships on a grid representing the ocean, and then they take turns firing shots at locations on the other player's board. The player is notified whether each shot is a hit or a miss, and 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" or "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 [34]) 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 would have been detected with our tool, we discovered it by inspection while considering whether to use the program as a case study (before the tool was implemented).

However, 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 bits of information; a non-fatal hit reveals two bits, one indicating the shot is a hit and a second indicating it is non-fatal. These flows can be observed in real time by running our tool in a mode that recomputes the flow on every program output, or each second, whichever is less frequent. Information about the ship locations is also revealed via the program's graphical interface, but we excluded that code from the analysis by explicitly declassifying some data passed to drawing routines; thus this analysis could miss leaks that occurred through the GUI libraries.

7.2 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 [44]). 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

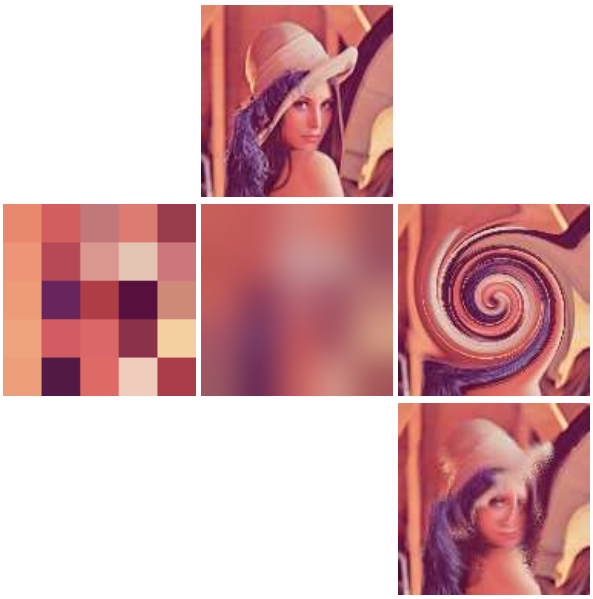

Figure 7. Image transformations vary in how much information they preserve. Our tool verifies that pixelating (left) or blurring (middle) the original image (top, 375120 bits), reveals only 1461 or 1717 bits respectively. By contrast, the bound our tool finds for the information revealed by a twisting transformation (right) is 357120 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).

key (a number of arbitrary-precision integers) as secret as it is read from a file.

Our tool finds that 128 bits of information about the secret key are revealed. The cut location reveals that this is the MD5 checksum of a response that includes a value decrypted with the public key, as expected under the protocol. Of course, our tool is not able to verify that MD5 is a secure one-way function, though that belief is part of why revealing those particular 128 bits is acceptable.

7.3 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 attempting 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 7 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 1461 bits revealed for pixelation and 1717 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 probably possible).

7.4 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 as 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 10:00am to noon, our tool bounds the amount of information revealed as 12 bits. In previous research using the tainting version of our tool, we had discovered that a loop that computes appointment intersections unnecessarily considered times every minute, and fixed it to use the same half-hour interval as the final display; the 12-bit measurement corresponds to a cut at checks made in this loop.

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

7.5 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 whether clients could access the server; more recently, the protocol has been extended with mechanisms that can enforce information-flow policies, by dividing clients into trusted and untrusted classes and re-

stricting what untrusted clients can do [50]. 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, our tainting-based checker can use 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 [25] 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. 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.

8.1 Static information-flow

Static checking aims to check the information-flow security of programs before executing them [12]. 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 [17, 35], barriers remain to the adoption of information-flow type checking [48] extensions to general purpose languages [34, 43, 28]. 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 [34] (closest are the poker game of [1] and the email client of [26]) or Flow Caml [43] dialects. Techniques based on type safety are inapplicable to languages that do not guarantee type safety (such as C and 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 program, the public outputs will be the same no matter what the secret inputs were [19, 48]. 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 [20, 14, 29]. 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 [30] may be difficult to automate. Our technique's results reflect the number of iterations that occur on a particular execution.

8.2 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 [16], and Gat and Saal propose reverting writes made by secret-using code [18] much as our technique does. 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.

Recent dynamic tools to enforce confidentiality policies do not scalably account for all implicit flows. Chow et al.'s whole-system simulator TaintBochs [8], which 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. Haldar, Chandra, and Franz [23] track information flow at the object level for the Java virtual machine, but since their technique does not allow an object to write public data after reading secret data, it prohibits most useful computations involving secrets. The RIFLE project [47] 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 sound and precise 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. Masri et al. [31] 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 did not evaluate how close their tool came to a sound tainting. For instance, they marked the input to `gzip` as tainted, much as we did with `bzip2`, but do not report whether the output was tainted. None of these tools enforce a quantitative security policy.

In an earlier unpublished 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 [33] for a simple formalized system that can be seen as modelling our tainting based implementation, with enclosure regions modified to be sound in the same way those in the present paper are. The simulation proof technique use there could be extended to the present system by treating the minimum cut corresponding to a maximum flow as a preemptive leakage annotation. Our discussion there of the policy guaranteed by the tool omits the issues of consistency we describe in Section 3.

Restrictions on information flow can also be enforced by an operating system. Traditional mandatory access control (MAC) techniques [13] at 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 [15], 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.

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 [39] is based on the same Valgrind framework as our tool, while other researchers have suggested using more optimized dynamic translation [27, 41], source-level translation [51], or novel hardware support [45] 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" [49]) or SQL statements [40].

9. Future directions

9.1 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 2.4 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, but would still be piecewise linear in the execution counts.

9.2 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 [49, 40]. 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 [46]. 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.

10. 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.

References

[1] 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.

[2] G. D. Battista and R. Tamassia. Incremental planrity testing. In *30th Annual Symposium on Foundations of Computer Science*, pages 436–441, Research Triangle Park, NC, USA, October 30–November 1, 1989.

[3] H. L. Bodlaender. A linear time algorithm for finding tree-decompositions of small treewidth. In *Proceedings of the Twenty-Fifth*

Annual ACM Symposium on Theory of Computing, pages 226–234, San Diego, CA, USA, May 15–18, 1993.

[4] M. D. Bond and K. S. McKinley. Probabilistic calling context. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2007)*, 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] S. Chaudhuri, K. V. Subrahmanyam, F. Wagner, and C. D. Zaroliagis. Computing mimicking networks. *Algorithmica*, 26(1):31–49, 2000.

[7] M. Chimani, C. Gutwenger, M. Jünger, K. Klein, P. Mutzel, and M. Schulz. The Open Graph Drawing Framework. In *15th International Symposium on Graph Drawing*, Sydney, Australia, September 23–26, 2007.

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

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

[14] 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.

[15] 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.

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

[17] 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.

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

[19] 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.

[20] 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.

[21] C. Gutwenger, M. Jünger, G. W. Klau, S. Leipert, P. Mutzel, and R. Weiskircher. AGD - a library of algorithms for graph drawing. In *9th International Symposium on Graph Drawing*, pages 473–474, Vienna, Austria, September 23–26, 2001.

[22] T. Hagerup, J. Katajainen, N. Nishimura, and P. Ragde. Characterizing multiterminal flow networks and computing flows in networks of small treewidth. *Journal of Computer and System Sciences*, 57(3):366–375, 1998.

[23] V. Haldar, D. Chandra, and M. Franz. Practical, dynamic information flow for virtual machines. In *2nd International Workshop on Programming Language Interference and Dependence*, London, UK, September 6, 2005.

[24] R. Hastings and B. Joyce. Purify: A tool for detecting memory leaks and access errors in C and C++ programs. In *Proceedings of the Winter 1992 USENIX Conference*, pages 125–138, San Francisco, CA, USA, January 20–24, 1992.

[25] 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`.

[26] 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.

[27] 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.

[28] 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.

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

[30] 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.

[31] W. Masri, A. Podgurski, and D. Leon. Detcting and debugging insecure information flows. In *Fifteenth International Symposium on Software Reliability Engineering*, pages 198–209, Saint-Malo, France, November 3–5, 2004.

[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. A simulation-based proof technique for dynamic information flow. In *PLAS 2007: ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, San Diego, California, USA, June 14, 2007.

[34] 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.

[35] 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.

[36] 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*, Grenoble, France, April 8–12, 2002.

[37] 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.

[38] 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*, San Diego, CA, USA, June 11–13, 2007.

[39] 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.

[40] 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.

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

[42] 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.

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

[44] S. Smith and M. Thober. Refactoring programs to secure information flows. In *PLAS 2006: ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, Ottawa, Canada, June 10, 2006.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

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

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

[51] 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.

[52] 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.