A Simulation-based Proof Technique for Dynamic Information Flow

Stephen McCamant Michael D. Ernst
MIT Computer Science and AI Lab
{smcc,mernst}@csail.mit.edu

Abstract

Information-flow analysis can prevent programs from improperly revealing secret information, and a dynamic approach can make such analysis more practical, but there has been relatively little work verifying that such analyses are sound (account for all flows in a given execution). We describe a new technique for proving the soundness of dynamic information-flow analyses for policies such as end-to-end confidentiality. The proof technique simulates the behavior of the analyzed program with a pair of copies of the program: one has access to the secret information, and the other is responsible for output. The two copies are connected by a limited-bandwidth communication channel, and the amount of information passed on the channel bounds the amount of information disclosed, allowing it to be quantified. We illustrate the technique by application to a model of a practical checking tool based on binary instrumentation, which had not previously been shown to be sound.

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

General Terms Languages, Measurement, Security, Theory, Verification

Keywords Information-flow analysis, dynamic analysis, implicit flow

1. Introduction

An information-flow security property provides an end-to-end constraint on a how a program can propagate information: for instance, requiring that it be impossible to deduce any information about the program's secret inputs from its public outputs. A widely held piece of conventional wisdom is that dynamic program analyses, those that examine only a single program execution, are unsuitable for checking information-flow properties. (This belief is based in part on some narrow theoretical impossibility results, discussed in Section 5.1.) One real limitation of past work on dynamic information-flow analysis has been a shortage of formal techniques for reasoning about the soundness of dynamic analyses, especially compared to the well-known techniques for, say, proving the soundness of a static type system.

PLAS'07 June 14, 2007, San Diego, California, USA.

In many practical situations, it is expected that a program will reveal some information. So it is useful to have a *quantitative* information-flow analysis for confidentiality: one whose output is a numeric bound, measured in bits, on the amount of information revealed by a particular execution. In previous work, we developed a tool that measures information flow in a program by tracking, for each bit, whether it might contain secret information (*dynamic tainting*), and that accounts for flows of information via control-flow (*implicit flows*).

Our goal here is to describe a new technique for proving that such a dynamic analysis is sound: that the actual amount of information revealed is no larger than the computed bound. The key idea of the proof is to simulate the analyzed program by a pair of programs connected by a limited-bandwidth channel (or *pipe*): one copy of the program starts with the secret input, and the other copy is responsible for output, so any information flow must occur via the pipe. If the amount of information transmitted over the pipe matches the analysis result, and the pair of programs faithfully simulates the original program, then the analysis's bound is sound. To illustrate this proof technique, we apply it to a formal model of our particular analysis, obtaining the desired soundness result.

We have also built a prototype implementation of our technique, which operates on C, C++, and Objective C programs at the binary level using the Valgrind instrumentation framework [20]. In case studies, this tool checked a variety of security properties on programs of up to half a million lines of code, including a property that was violated by a previously unknown bug. These aspects of the research are described in a technical report [14]; a revised version of that work, including improved enclosure annotations like those described in Section 2.2 and an additional case study, is currently under review [15]. Without the proof technique described here, however, we were previously able to give only informal arguments for our tool's soundness.

The remainder of this paper is organized as follows. Section 2 describes a core language and the application of our analysis to it. Section 3 describes how to simulate an analyzed program by a pair of programs with bounded communication. Section 4 gives a soundness proof using the simulation, in which the key lemma is that the pair of programs faithfully simulates the original program. Finally, Section 5 compares this work with other dynamic and static information-flow analyses and their proofs, and Section 6 concludes.

2. Formalized analysis

In order to prove the soundness result for our technique, we formalize it in the context of a simple imperative core language containing all the features relevant to our analysis. The language includes two features that correspond to annotations in the real system: a *pre-emptive leakage* statement that identifies locations where secret in-

stmt ::= $\mathtt{x}_i$ = $\mathtt{x}_j$
$\mathtt{x}_i$ = not $\mathtt{x}_j$
$\mathtt{x}_i$ = $\mathtt{x}_j$ and $\mathtt{x}_k$
output $\mathtt{x}_i$
if $\mathtt{x}_i$ then goto l
end
leak $\mathtt{x}_i$
enclose(l, n, R)
end_enclose

Figure 1. Syntax for a simple imperative language described in Section 2.1

formation has a compact representation, and *enclosure regions* that isolate independent subcomputations used only for their output.

2.1 An unstructured imperative language

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

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

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

$(pc, S, E) \rightarrow (pc', S', E')$ where:
$pc' = l$ if code(pc) = "if $\mathtt{x}_i$ then goto l" and $S(i) = 1$
$pc' = pc + 1$ if code(pc) = "if $\mathtt{x}_i$ then goto l" and $S(i) = 0$
$pc' = pc$ if $pc = E.l$ and $E.k > 1$
$pc' = E.l$ if $E.k = 1$
$pc' = E.l$ if $E \neq \perp$ and code(pc) = "end"
$pc' = pc + 1$ otherwise
$E' = \perp$ if $E.k = 0$ and $E.l = pc$
$E' = \perp$ if code$(pc) \neq$ "enclose(l, n, R)" and $E = \perp$
$E' = (l, n, R, S)$ if code(pc) = "enclose(l, n, R)" and $E = \perp$
$E' = (E.l, E.k - 1, E.R, E.S)$ otherwise
$S' = S[i \leftarrow S(j)]$ if code(pc) = "$\mathtt{x}_i$ = $\mathtt{x}_j$"
$S' = S[i \leftarrow \neg S(j)]$ if code(pc) = "$\mathtt{x}_i$ = not $\mathtt{x}_j$"
$S' = S[i \leftarrow S(j) \wedge S(k)]$ if code(pc) = "$\mathtt{x}_i$ = $\mathtt{x}_j$ and $\mathtt{x}_k$"
$S' = S[\overline{E.R} \leftarrow E.S(\overline{E.R})]$ if $E.k = 0$ and $E.l = pc$
$S' = S$ otherwise
output $S(i)$ if code(pc) = "output $\mathtt{x}_i$" and $E = \perp$
stop if code(pc) = "end" and $E = \perp$

Figure 2. Regular semantics for the language given in Section 2.1. For brevity, we follow the convention that any condition involving a field of E is false if $E = \perp$.

More formally, we refer to the behavior of a program (without the dynamic information-flow analysis) as the *regular semantics*, given as a state transition relation in Figure 2. The state of a program consists of a program counter pc holding the label of the next instruction to execute, a store S holding the values of variables, and a structure E holding information about the current enclosure region. E is either a distinguished value $\perp$ if execution is not in an enclosure region, or a tuple (l, k, R, S) where l is a label, k is a nonnegative integer, R is a set of variables, and S is a store. l is the label at which the enclosure region will end, k is a counter decremented once per step to control how long the region executes, R is the set of result variables, and S is a saved copy of program variables to restore at the end of the region. The notation $S[i \leftarrow b]$ represents a new store in which i is bound to b, and all other variables have the same bindings as in S; we also abuse notation by using sets as indexes and/or values to indicate that each i is bound to the corresponding or only value from b. Initially, pc points to the first statement, $E = \perp$, and S is populated with the program inputs.

After most statements, execution continues with the following statement, but an if statement causes a branch if its condition is true, and an enclosure region jumps to its end if the counter $E.k$ runs out, or an end is encountered. E is initialized at the beginning of an enclosure region and cleared at the end; when it is present, $E.k$ is decremented on each step. The assignment statements modify the store in the expected way; end_enclose also restores the contents of all variables except the results of the region. The output and end statements have the expected effects, but they are disabled inside enclosure regions.

2.2 Analysis semantics

In the dynamic information-flow analysis, our tool counts the potential propagation of secret information in two ways: a bit may be marked as secret (like "tainted") if it might contain secret information, and a counter keeps track of other leaks outside the set of variables. The secrecy status of bits is propagated conservatively: a

$(pc, E, S, SS, c) \to (pc', E', S', SS', c')$ where:
$SS' = SS[i \leftarrow SS(j)]$ if code(pc) = "$\texttt{x}_i$ = $\texttt{x}_j$"
$SS' = SS[i \leftarrow SS(j)]$ if code(pc) = "$\texttt{x}_i$ = not $\texttt{x}_j$"
$SS' = SS[i \leftarrow (SS(j) \vee SS(k)) \wedge$
$(S(j) \vee SS(j)) \wedge$
$(S(k) \vee SS(k))]$
if code(pc) = "$\texttt{x}_i$ = $\texttt{x}_j$ and $\texttt{x}_k$"
$SS' = SS[i \leftarrow 0]$ if $E = \perp$ and code(pc) = "output $\texttt{x}_i$"
$SS' = SS[i \leftarrow 0]$ if $E = \perp$ and code(pc) = "if $\texttt{x}_i$ then goto l"
$SS' = SS[i \leftarrow 0]$ if $E = \perp$ and code(pc) = "leak $\texttt{x}_i$"
$SS' = E.SS[E.R \leftarrow 1]$ if $E.k = 0$ and $E.l = pc$
$SS' = SS$ otherwise
$c' = c + 1$ if $E = \perp$ and $SS(i)$ and code(pc) = "output $\texttt{x}_i$"
$c' = c + 1$ if $E = \perp$ and $SS(i)$ and code(pc) = "if $\texttt{x}_i$ then goto l"
$c' = c + 1$ if $E = \perp$ and $SS(i)$ and code(pc) = "leak $\texttt{x}_i$"
$c' = c$ otherwise
$E'.SS = SS$ if code(pc) = "enclose(l, n, R)" and $E = \perp$
$E'.SS = E.SS$ otherwise

Figure 3. Instrumented semantics describing secrecy tracking for the core language, as described in Section 2.2. The rules for pc, E, and S are the same as in Figure 2.

copy of a secret bit is secret, and the output of an operation is secret if any of the inputs that contributed to it were. (But the result of an operation on a public value and a secret value can be public if the result does not depend on the secret value: for instance, multiplying a public 0 by a secret number yields a public 0.) Preemptive leakage via a leak statement erases the secrecy of a bit and simultaneously increments the counter to compensate. Preemptively leaking information at a point when it has a compact representation allows the bound computed by the analysis to be more precise. To account for implicit flows, a bit is also leaked if a secret bit is used as a branch condition, and of course secret bits are counted as leaked if they are output. Branches on secret data are not counted as leaks inside an enclosure region, but to compensate, the output of an enclosure region is always marked as secret.

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

3. Simulation environment

Given an instrumented program execution under the semantics described in Section 2, we wish to prove that at any point in execution, the counter c is an upper bound on the number of bits of information about the secret inputs present in the program's output. To do this, we construct two copies of the instrumented program, connected by a unidirectional channel called a *pipe*; we call the two copies the *writer* and the *reader* according to the way they use the pipe. The two copies have the same program text and public inputs, but initially only the writer has the secret input data. The two copies execute in lockstep; on some steps, the writer writes a bit to the pipe, and the reader reads it. The goal is that the two copies of the program should produce the same results; this demonstrates that the information sent via the pipe is the only potentially-secret information needed to produce the program output.

Writer:
$(pc, E, S, SS, c) \to (pc', E', S', SS', c')$ where:
write$(\texttt{x}_i)$ if $E = \perp$ and $SS(i)$ and code(pc) = "output $\texttt{x}_i$"
write$(\texttt{x}_i)$ if $E = \perp$ and $SS(i)$ and code(pc) = "if $\texttt{x}_i$ then goto l"
write$(\texttt{x}_i)$ if $E = \perp$ and $SS(i)$ and code(pc) = "leak $\texttt{x}_i$"

Reader:
$(pc, E, S, SS, c) \to (pc', E', S', SS', c')$ where:
output read() if $SS(i)$ and code(pc) = "output $\texttt{x}_i$" and $E = \perp$ (*)
output $\texttt{x}_i$ if $\neg SS(i)$ and code(pc) = "output $\texttt{x}_i$" and $E = \perp$
$pc' = l$ if code(pc) = "enclose(l, n, R)"
$pc' = l$
if $SS(i)$ and code(pc) = "if $\texttt{x}_i$ then goto l" and read() = 1 (*)
$pc' = l$
if $\neg SS(i)$ and code(pc) = "if $\texttt{x}_i$ then goto l" and $\texttt{x}_i$ = 1
$pc' = pc + 1$
if $SS(i)$ and code(pc) = "if $\texttt{x}_i$ then goto l" and read() = 0 (*)
$pc' = pc + 1$
if $\neg SS(i)$ and code(pc) = "if $\texttt{x}_i$ then goto l" and $\texttt{x}_i$ = 0
$S' = S[i \leftarrow \text{read}()]$ if $SS(i)$ and code(pc) = "output $\texttt{x}_i$"
$S' = S[i \leftarrow \text{read}()]$ if $SS(i)$ and code(pc) = "if $\texttt{x}_i$ then goto l"
$S' = S[i \leftarrow \text{read}()]$ if $SS(i)$ and code(pc) = "leak $\texttt{x}_i$"

Figure 4. Modified semantics for the pipe writer and reader, as described in Section 3. The writer rules are in addition to those given in Figures 2 and 3. The reader rules also extend those, except that the three rules marked (*) replace the corresponding ones from Figure 2; the reader rules add the condition $SS(i)$ and use read() in place of $\texttt{x}_i$. Because the reader skips directly to the end of enclosure regions, the effect is as if each of the new reader rules included the condition $E = \perp$, but we omit it for space.

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

The writer semantics are purely an addition to the instrumented semantics, just as the instrumented semantics added to the regular semantics, so the writer's behavior is the same. By contrast, the reader's semantics are different; proving that the reader's behavior is similar is the main task of Section 4. The modified semantic rules for the writer and reader are given in Figure 4, where the pipe operations are represented as write$(\texttt{x}_i)$ and read(). (When read() appears in the definitions of two post-state variables for a single state transition, the intended meaning is that a single bit is read, and used in multiple places.) As mentioned earlier, the reader does not start with any secret information. It does not matter what its copies of the secret input variables are initialized to, but for concreteness, say they all start as 0.

4. Proof

Section 3 described the construction of a pair of programs intended to give the same results as an instrumented secret-using program, but with the use of secret data by the second (reader) program rationed by a special channel. To use this construction to obtain a soundness result for the analysis, we must first prove that the reader faithfully simulates the instrumented program, and then relate the information disclosed by the reader to the leakage count maintained by the instrumented program.

4.1 Simulation lemma

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

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

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

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

- $\mathtt{x}_i$ `=` $\mathtt{x}_j$: Only the value stored at location i is modified, so we must check that it is either the same or secret in both post-states. If $SS(j) = 0$, then $S_{\mathrm{W}}(j) = S_{\mathrm{R}}(j)$, and so $S'_{\mathrm{W}}(i) = S'_{\mathrm{R}}(i)$. On the other hand if $SS(j) = 1$, then $SS'_{\mathrm{W}}(i) = SS'_{\mathrm{R}}(i) = \mathtt{1}$.
- $\mathtt{x}_i$ `= not` $\mathtt{x}_j$: Similarly, if $SS(j) = 0$, then $S_{\mathrm{W}}(j) = S_{\mathrm{R}}(j)$, and so $S'_{\mathrm{W}}(i) = \neg S_{\mathrm{W}}(j) = \neg S_{\mathrm{R}}(j) = S'_{\mathrm{R}}(i)$. On the other hand if $SS(j) = 1$, then $SS'_{\mathrm{W}}(i) = SS'_{\mathrm{R}}(i) = \mathtt{1}$.
- $\mathtt{x}_i$ `=` $\mathtt{x}_j$ `and` $\mathtt{x}_k$: Here there are three kinds of cases. If both values are public, $SS(j) = SS(k) = \mathtt{0}$, then both arguments are the same by assumption, $S_{\mathrm{W}}(j) = S_{\mathrm{R}}(j)$ and $S_{\mathrm{W}}(k) = S_{\mathrm{R}}(k)$, so the results are also the same: $S'_{\mathrm{W}}(i) = S_{\mathrm{W}}(j) \wedge S_{\mathrm{W}}(k) = S_{\mathrm{R}}(j) \wedge S_{\mathrm{R}}(k) = S'_{\mathrm{R}}(i)$. If either argument is public and zero, say $SS(j) = 0$ and $S_{\mathrm{W}}(j) = S_{\mathrm{R}}(j) = 0$, then both results must be zero, and so equal: $S'_{\mathrm{W}}(i) = \mathtt{0} \wedge S_{\mathrm{W}}(k) = \mathtt{0} = \mathtt{0} \wedge S_{\mathrm{R}}(k) = S'_{\mathrm{R}}(i)$. Otherwise, at least one argument is secret, and neither argument is both public and zero, so all three of the conjuncts in the rule for SS' are true, and $SS'_{\mathrm{W}}(i) = SS'_{\mathrm{R}}(i) = \mathtt{1}$.
- `output` $\mathtt{x}_i$: If $SS(i) = \mathtt{0}$, then the state is unchanged. Otherwise, note that $E = \bot$, so the writer writes a bit b which is read by the reader. $SS'_{\mathrm{W}}(i) = SS'_{\mathrm{R}}(i) = 0$, but $S'_{\mathrm{W}}(i) = S'_{\mathrm{R}}(i) = b$. Also, observe that the writer and reader output the same bit in either case.
- `if` $\mathtt{x}_i$ `then goto` l: As in the `output` case, the branch condition is either the same by assumption if it's public, or if it's secret, the same because it is written by the writer and read by the reader. Thus, either $pc'_{\mathrm{W}} = l = pc'_{\mathrm{R}}$ if the branch is taken, or $pc'_{\mathrm{W}} = pc + 1 = pc'_{\mathrm{R}}$ if not.
- `end`: Note that $E = \bot$, so both programs stop and this case is satisfied vacuously.
- `leak` $\mathtt{x}_i$: Also like the `output` case, if $SS(i) = \mathtt{0}$, then the state is unchanged. Otherwise, $E = \bot$, so the writer writes a bit b which is read by the reader, and $S'_{\mathrm{W}}(i) = S'_{\mathrm{R}}(i) = b$.
- `enclose`(l, n, R): Uniquely in this case, it is not the next states that are related by $\sim$, but the next non-enclosed states, $n + 1$ steps later. Because `end` is disabled in an enclosure region, there are sure to be such subsequent states: when the countdown $E.k$ reaches zero, control will have reached the `end_enclose` at l, so the next states have pc referring to the next statement after that. Using primes for this state, we clearly have $E'_{\mathrm{W}} = E'_{\mathrm{R}} = \bot$. The secrecy store in this state consists of the saved secrecy store SS, with all of the locations in R marked as secret, but R is the same for both programs, so $SS'_{\mathrm{W}} = SS'_{\mathrm{R}}$. For the regular store, locations not in R were saved and restored, so match the values in S_{W} and S_{R}, which are either equal or secret by the induction hypothesis. Values in R are marked as secret, so may be different, but $\sim$ holds.
- `end_enclose`: The usual situation of the end of an enclosure region was described in the previous case. Here, $E = \bot$; if an `end_enclose` is encountered outside an enclosure region, it has no effect.

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

4.2 Final result

To obtain the final soundness result, we argue that the reader produces the output using only the information sent on the pipe as secret input. It is here that our notion of "information" must be defined, though in fact the argument is not sensitive to details of that definition. The key property is that a closed system cannot create information: the amount of information it its outputs is at most the information in its inputs. We use a definition of information leakage based on entropy, specialized to embody conservative assumptions about information not available to the analysis.

We define the amount of information leaked by a program to be the entropy of the program's outputs as a distribution over the possible values of the secret inputs, with the public inputs held constant. This definition is used by Clark et al. [4] and is analogous to other information-theoretic measures used in the literature (e.g., [18]). For a deterministic program like we consider, it is equivalent to the conditional mutual information between the output and the secret inputs, given the public inputs. It is also roughly the logarithm of the effort that knowledge of the output would save an attacker attempting to guess the secret by brute force [13]. Non-interference corresponds to an entropy of 0.

Entropy is defined with respect to a probability distribution of possible secret inputs, but only a single input is available when a program is analyzed. Therefore, we make a conservative assumption that the distribution is chosen to maximize the information revealed. This corresponds to assuming at every step that a bit of data represents a full bit of information (entropy). Note that this does not mean assuming that the input is uniformly distributed: for instance, a negative response from a password checker conveys very little information if the password is randomly distributed over a large space, but it conveys a full bit if the password is `apple` with probability $1/2$ and `banana` with probability $1/2$.

THEOREM 4.1. *Suppose that a program has run for some number of steps under the instrumented semantics, the value of the counter c is k, and the output is a bit string $b_0 b_1 \ldots b_n$. Then the number of bits of information about the secret input in $b_0 b_1 \ldots b_n$ is at most k.*

Proof: Consider an execution of the same program by the writer and reader described above. Since the semantics of the writer are an extension of those of the instrumented program, its output will also be $b_0b_1 \dots b_n$. By the simulation lemma, each of the output states of the reader will be related by $\sim$ to the output states of the reader, so the reader's output will also be $b_0b_1 \dots b_n$. In the writer semantics, a bit is written to the pipe on exactly those steps when the counter is incremented, so the number of bits written to the pipe is also k. Since the reader reads only the bits written by the writer, the number of bits read from the pipe is k as well. Because the data sent on the pipe consisted of k bits, it can contain at most k bits of information. Because the text of the program and the public inputs are public, the only secret data read by the reader is the bits from the pipe. The amount of secret information in the output of the reader cannot be greater than the amount of information in its inputs (by the Data Processing Theorem of information theory [16]). Thus the amount of secret information in the output is at most k bits.

5. Related work

Next we compare our work to other techniques for information-flow tracking, and what is known about their soundness. Dynamic techniques like ours have a long history, but have less often been treated formally, perhaps because of some seeming impossibility results. Static techniques, which make guarantees about any possible execution, have recently been more popular, but are not easily combined with quantitative measurement of leakage.

5.1 Dynamic analyses

Schneider [21], following McLean [17], proves that information-flow is not a property that can be checked by a class of techniques called "execution monitoring" (EM). Two features of our technique put it outside Schneider's class EM. First, it depends not only on the inputs and outputs of a program, but on the program text. Second, like a static information-flow analysis, it is conservative. Because tainting is only an approximation of secrecy, there is no guarantee that when the bound produced by the technique is non-zero, the output will contain secret information.

Some of the earliest proposed systems for enforcing confidentiality policies on programs (including implicit flows) were based on run-time checking: Fenton discovered the difficulties of implicit flows in a tainting-based technique [7], and Gat and Saal propose reverting writes made by secret-using code [8] much as our technique does. Compared to our technique, these approaches do not support permitting acceptable flows or measuring information leakage. Fenton [7] proves a soundness theorem for a counter machine whose counters have static secrecy classes.

Recent dynamic tools to enforce confidentiality policies have a more practical focus, but do not scalably account for all implicit flows. Most similar to our system is Chow et al.'s whole-system simulator TaintBochs [3], 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. The RIFLE project [24] 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 whether it can scale to programs that store secrets in dynamically allocated memory. The RIFLE authors discuss soundness issues similar to those that apply to our tool, but sketch a soundness result only in broad outline.

The TightLip system [26] is a practical information-leak checking tool whose implementation is similar to the duplication we use only as a proof technique. TightLip runs a program in parallel with a copy whose inputs are scrubbed (e.g., have secret bits replaced with zeros): if the two copies' outputs are different, the program has leaked information. TightLip has been used so far only to check policies that exclude any information flow, but it might be extended to allow certain flows by copying that information to the replica, analogously to the treatment of preemptive leakage in our proof.

5.2 Static analyses

Static checking aims to check the information-flow security of programs before executing them [5]. The most common technique is to add information-ownership annotations to a type system; then if a program type checks, a soundness result guarantees that the program satisfies some security property [25]. For instance, non-interference is the property that for any given public inputs to program, the public outputs will be the same no matter what the secret inputs were [9]. Information-flow extensions have been proposed to several general-purpose languages [19, 22, 11], with safety proofs given for some smaller subsets. Of course, the form of a soundness (or safety) proof is different for a static system; in a static system, the analysis produces a single result, which is sound if it bounds any possible execution. By contrast, a dynamic system like ours might give a different result for each program input, but the technique is sound if each result bounds its corresponding execution.

Quantitative measurements based on information theory have often been used in theoretical definitions of information-flow security [10, 6, 12], but quantification of leakage is more difficult in static systems, since programs generally leak different amounts of information when given different inputs. Clark et al.'s system for a simple while language [4] is the most complete static quantitative information flow analysis for a conventional programming language. Recent work on a formula giving precise per-iteration leakage bounds for loops [13] provides a sound basis for estimating information-flow, but the construction of a practical static analysis on top of it remains an open problem.

The idea of using two copies of a program appears in a static information-flow security proof technique known as self-composition [1, 2, 23]. For instance, to prove that P satisfies non-interference, self-composition considers a program $P; P'$ where P' is a copy of P with distinct low inputs and state variables, and asks whether there are any inputs for which the program results are different. Thus self-composition reduces an information-flow property to a safety property more amenable to standard proof techniques, analogously to how our proof technique reduces information-flow to behavioral equivalence. However, self-composition has been used to prove particular programs secure (together with a tool such as a theorem prover), whereas our technique proves the soundness of an analysis algorithm.

6. Conclusion

We have described a novel proof technique for checking that a dynamic information-flow analysis gives sound quantitative flow bounds. The proof technique represents a bound on a program's information usage by giving a simulation of the program by a pair of program replicas, one of which may only access secret data via a channel whose bandwidth is bounded. If this pair of programs can faithfully simulate the original one, then the original program used only as much information in producing its output as passed through the channel. By applying the proof technique to a core language and analysis, we verified for the first time the soundness of a previously-described practical flow-checking tool based on binary instrumentation.

Acknowledgments

We would like to thank the anonymous reviewers for a number of helpful suggestions. 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] Ádám Darvas, R. Hähnle, and D. Sands. A theorem proving approach to analysis of secure information flow. In *Workshop on Issues in the Theory of Security*, Warsaw, Poland, Apr. 5–6, 2003.

[2] G. Barthe, P. R. D'Argenio, and T. Rezk. Secure information flow by self-composition. In *17th IEEE Computer Security Foundations Workshop*, pages 100–114, Pacific Grove, California, USA, June 28-30, 2004.

[3] 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, Aug. 11–13, 2004.

[4] 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, Mar. 27–28 2004.

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

[6] 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.

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

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

[9] 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, Apr. 26–28, 1982.

[10] 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.

[11] 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.

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

[13] 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, Jan. 17–19, 2007.

[14] 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, Nov. 17, 2006.

[15] S. McCamant and M. D. Ernst. Quantitative information-flow tracking for type-unsafe languages. Submitted for conference publication, 2007.

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

[17] J. McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *1994 IEEE Symposium on Research in Security and Privacy*, pages 79–93, Oakland, CA, USA, May 16–18, 1994.

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

[19] 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, Jan. 20–22 1999.

[20] 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.

[21] F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, Feb. 2000.

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

[23] T. Terauchi and A. Aiken. Secure information flow as a safety problem. In *Proceedings of the Twelfth International Symposium on Static Analysis, SAS 2005*, pages 352–367, London, UK, Sept. 7–9, 2005.

[24] 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, Dec. 4–8, 2004.

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

[26] 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, Apr. 11–13, 2007.