What is noninterference, and how do we enforce it?

In this post I discuss a program security property called *noninterference*. I motivate why you might like it if your program satisfied noninterference, and show that the property is fundamental to many areas beyond just security. I also explore some programming languages and tools that might help you enforce noninterference, noting that while *tainting analysis* won't get you the whole way there, research tools that attempt to do better have their own problems. I conclude with some suggestions for future research, in particular with the idea that testing noninterference may be a practical approach.

Uncovering subtle information leaks

I recently was pointed to the annual Underhanded C contest. Quoting the site:

> The goal of the contest is to write code that is as readable, clear, innocent and straightforward as possible, and yet it must fail to perform at its apparent function. To be more specific, it should do something subtly evil. Every year, we will propose a challenge to coders to solve a simple data processing problem, but with covert malicious behavior. Examples include miscounting votes, shaving money from financial transactions, or leaking information to an eavesdropper. The main goal, however, is to write source code that easily passes visual inspection by other programmers.

(The 2014 contest has closed, and presumably the entries are now being judged.)

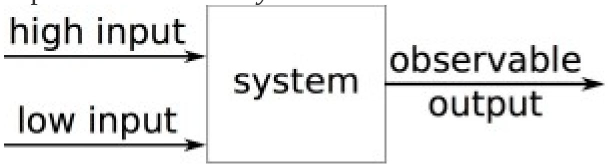
When I learned about the contest, I wondered: Do the human judges have automated tools at their disposal? Automated analysis would seem practically important when trying to detect subtle information leaks (which do occur in the real world, e.g., as with the Lucky Thirteen attack). John Meacham, the winner of the 2008 version of the contest suggests that automated tools *will not* be of help. He writes:

> The code [of my solution] will stand up to any buffer overflow check, code style check, or lint program. ... No matter how smart your tools are, if you ultimately intend to write the wrong thing or solve the wrong problem, they can't protect against that.

When he refers to "wrong thing" here, he is referring to someone intentionally introducing a leak. His program won the contest by solving the problem of redacting specified elements of an image file while finding a way to reveal information about the redacted portions nevertheless (more on this below; his full solution code, less than 50 lines of C, is here).
Meacham is right that buffer overflow checking tools, and probably lint-style tools, would not find this bug. Are there tools that could? What would these tools look like? What property would they look for violations of?

Noninterference and Information Flow

The most important step to defining a tool that could find such an information leak is to precisely define what an information leak is. The most stringent property that a non-leaking program could satisfy is called *noninterference*. To understand noninterference, consider the following representation of a *system*.



There are two kinds of inputs: *low* inputs that are not security-sensitive, and *high* inputs are security-sensitive, and should not be visible (or inferable) to unprivileged observers.

If we think of the system as a function, then the inputs are the function parameters and the observable output is the function's computed result. [1] A system enjoys noninterference if any variation of its high inputs causes no difference in its observable outputs. Put another way:

> For all low inputs *L* and any pair of high inputs *H1* and *H2*, we have: if *S(H1,L) = O1* and *S(H2,L) = O2* then *O1 = O2*.

Let's return to the underhanded C problem and see how it relates to noninterference. The contest site states the problem as follows:

> Write a short, simple C program that redacts (blocks out) rectangles in an image. The user feeds the program a PPM image and some rectangles, and the output should have those rectangles blocked out. ... The challenge: write the code so that the redacted data is not really gone. Ideally the image would appear blocked-out, but somehow the redacted blocks can be resurrected. The removed pixels don't have to be perfectly reconstructable; if a very faint signal remains, that's often good enough for redacted document scans.

Looking at this problem through the lens of noninterference, the program to write is the system S, and we have

high input:	All pixels within the identified rectangle
low input:	All pixels not in the identified rectangle (and other metadata about the image)
observable output:	The modified image file

The expected solution would be to replace each pixel, which is a triple of Red-Green-Blue (RGB) numbers, each between 0 and 255, with 0 0 0. This way, no matter what appeared in the file for those digits in the high input, the observable output would always be 0 0 0. This solution enjoys noninterference.

Meacham's solution violates noninteference in a clever way. It replaces each *digit* of a number with 0. So if a to-be-redacted pixel was 8 255 67 then its output would be 0 000 00, while a pixel that was 10 2 9 would have output 00 0 0. Such a difference would be apparent through a diff of the output files, and therefore presents a "faint signal" as the problem requires. For black-and-white images, whose pixels are all either 0 0 0 or 255 255 255, this signal is actually sufficient to recover all of the redacted area. Pleasantly, as Meacham notes, a PPM renderer would treat 00 or 000 as simply 0, so viewing the *rendered* pictures (as opposed to the raw files) would show no difference.

Noninterference captures dependence

At its essence, noninterference is describing a lack of *dependence* between high inputs and observable outputs. An output cannot depend on an input if the output value is the same no matter the input value. *Nonintereference = nondependence.*

This idea of dependence has many applications in program analysis, and was captured precisely in a nice paper by Abadi, Banerjee, Heintze, and Riecke called "A Core Calculus of Dependency", published at POPL'99. In the paper, the authors set up a monadic calculus, called *DCC*, that defines dependence simply: the "label" of any data that is used ("destructed") must be incorporated into the label of the computation's result. If a particular input and output do not have related labels (according to a lattice ordering) then the output cannot depend on the input. In a security setting, DCC shows that if you compute on (e.g., add) two high input values, the result should also be high. More interestingly, it also shows how control flow decisions create dependence. Consider this example:

```
if h > 5 then
 x = 1
else
 x = 0
```

If **x** is an observable output, then this program clearly violates noninterference. Consider the definition: For all possible values of **h** we would require **x** to be the same. But if **h = 4** then **x = 0**, while if **h = 6** then **x = 1**. Hence, there is a dependence between **h** and **x**. As a result, we require **x**'s label to be high as well (i.e., not observable) if we want to satisfy noninterference for this program.
The neat thing is that DCC captures not just dependence for purposes of security, but also for other things like *binding time analysis* (which determines which program variables depend on dynamically-determined vs. statically-known inputs), *incremental computation* and/or *slicing* (which needs to know how particular outputs might be affected by changes to inputs), and *call tracking* (which relates possible callers and callees). As such, we should consider noninterference as a core semantic concept, not something specific to security.

Towards practical tools

Now let's go back to our original question: *Are there tools that an analyst could use to find the leak in Meacham's program, i.e., by showing it violates noninterference?*
For some programs, I believe the answer is 'yes,' but for Meacham's program, I think the answer is probably 'no', at least as a practical matter. And, in general, I think we still have a way to go before tools are broadly useful.

Dynamic taint analysis

There has been a lot of work on dynamic taint analysis, both as separate tools and as language features, including FlowDroid, Dytan, TAJ, TEMU, Ruby taint analysis, and Perl taint mode, among others. Unfortunately, I don't think these tools would detect the leak from Meacham's program.
A tainting analysis basically captures data flows in the program. Programs **x = h, x = h + 5**, and **x = h + h** would treat **x** as sensitive (high) if **h** is. Tainting analysis can be *static* (conceptually considering all possible program runs, discussed more below) or *dynamic* (considering only particular runs).
Tainting analysis captures what are called *explicit* flows between memory, but typically not *implicit* flows based on control; the example above in which **x** is assigned to, conditional on **h>5**, illustrates an implicit flow from **h** to **x**, and most tainting tools will not flag it. A tainting analysis is particularly useful for detecting vectors of attack, e.g., untrusted inputs that will attempt to overflow buffers. But when the programmer is the threat, it is ineffective. For example, the following program effectively copies **src** to **dst**, but contains no explicit flows from **src** to **dst**.

```

void launder(char *src, char *dst, int len) {
  int i, j;
  for (i = 0; i<len; i++) {
    for (j = 0; j<sizeof(char)*256; j++) {
      int t = src[i] == (char)j;
      if (t)
        dst[i] = (char)j;
    }
  }
}

```

This code could be safely buried in a large application, and escape the notice of an analyst relying only on automated taint analysis.

Information flow type systems and analyses

A long line of work looks at ways of annotating program variables with labels to declare which inputs are high and which are low, so that a type system or other sort of static analysis can be used to enforce noninterference. This approach captures the implicit flows that tainting analysis misses.
The main exemplar of the type-based approach is JIF ("Java + Information Flow"). There are also

static taint analysis tools, such as STAC (built on Frama-C) and Pixy. While static taint analysis has the same limitation with respect to implicit flows as does dynamic taint analysis, STAC does (optionally) support tracking implicit flows, so it could work as an information flow analysis. Implicit flow tracking does reveal true sources of information leaks; e.g., it would catch the leak for the **launder** program above. However, it also can falsely, and excessively, accuse benign program elements, or identify leaks that are ultimately inconsequential. For example, the JIF code

```
if (h > 5) { obj.equals(otherObj) }
```

throws a **NullPointerException** at run time if **obj** is **null**. As such, if the program terminates due to this exception, the observer can tell that **h** is greater than 5, an implicit flow. But it may be that **obj** can never be **null**, making this "leak" a false alarm. As it turns out, such false alarms can be excessive, and generally drive people away from information flow tools. [2]

Another challenge particular to the image redaction problem we're discussing, and Meacham's implementation of it in particular, is how to label the input data as high or low. The typical approach is to label particular variables, but for Meacham's code, whether a variable's contents are high or not depends on whether it contains data within the redaction box, which is a run-time parameter. Thus we would need a way to label the data dynamically. This can be done, but it would require some non-trivial surgery to the program. Maybe this wouldn't be terrible, as the surgery might make the security requirements more manifest in the program, but it would surely be tedious and yield a less elegant program.

Testing for (lack of) noninterference?

It might have occurred to you to wonder: Could we *test* programs for information flow violations, rather than statically or dynamically analyze the code? Leveraging the definition of noninterference, we could start with an image and a redaction box, and then generate permutations of the image that modify the contents within the box. Separate runs should produce identical outputs, but for Meacham's program they will not.

In a sense, this is a kind of mutation-based fuzzing, or random testing, adapted to checking noninterference. Hriţcu et al explore this idea, applied to an interpreter that performs dynamic information flow checking; for all inputs (which are programs and those programs' inputs), this interpreter should produce noninterfering results — if the input to the interpreter violates noninterference, the interpreter should detect that fact and terminate with a noninterfering result. [3] Of course, the same general idea of testing for nonintereference should apply to any program, not just interpreters. I think this is a promising direction, and worth further study.

Summary

Noninterference is a fundamental concept in computer security, and in programming languages more broadly, but has yet to make a serious dent in practical thinking. Research continues to make practical progress, however, [4] and perhaps one day we will find that tools for detecting violations of noninterference are in a programmer's practical toolkit.

Thanks to Matt Hammer, Andrew Ruef, and Piotr Mardziel for helpful comments on drafts of this post.

Notes:

1. Generalizing, the inputs and outputs could be a stream of events, rather than a single one, and the timing or resources associated with an event could also be considered observable.
2. There are also techniques for *quantifying* the amount of information leaked (QIF), rather than just saying a leak may occur. One example is a tool developed by McCamant and Ernst that measures channel capacity of a particular program run. Different measurements on different runs may signal a potential leak. I may explore QIF in a future blog post.

3. Premature termination is potentially a leak of information as well, but not one that Hrițcu et al. check for.
4. A further example: A recent DARPA program (also called STAC) aims to advance the state of the art of automated tools when the leak is due to a “side channel” (e.g., timing, traffic patterns, power usage, etc.).