EXERCISE #13

INFORMATION FLOW REVIEW

Write your name and answer the following on a piece of paper

- *Give an example of a (pseudocode) program with an information flow that may be considered to violate integrity. Explain why the program violates integrity.*

ADMINISTRIVIA AND ANNOUNCEMENTS

Let's read a paper!

Operating Systems

R.S. Gaines
Editor

Certification of Programs for Secure Information Flow

Dorothy E. Denning and Peter J. Denning
Purdue University

This paper presents a certification mechanism for verifying the secure flow of information through a program. Because it exploits the properties of a lattice structure among security classes, the procedure is sufficiently simple that it can easily be included in the analysis phase of most existing compilers. Appropriate semantics are presented and proved correct. An important application is the confinement problem: The mechanism can prove that a program cannot cause supposedly nonconfidential results to depend on confidential input data.

Key Words and Phrases: protection, security, information flow, program certification, lattice, confinement, security classes

CR Categories: 4.3, 4.35, 5.24

Work reported herein was supported in part by the National Science Foundation under grants GJ-43176 and GJ-41289 and by IBM under a fellowship. Authors' present address: Computer Science Department, Purdue University, West Lafayette, IN 47907.

1. Introduction

Computer system security relies in part on *information flow control*, that is, on methods of regulating the dissemination of information among objects throughout the system. An information flow policy specifies a set of *security classes* for information, a *flow relation* defining permissible flows among these classes, and a method of *binding* each storage object to some class. An operation, or series of operations, that uses the value of some object, say x, to derive a value for another, say y, causes a *flow* from x to y. This flow is admissible in the given flow policy only if the security class of x flows into the security class of y.

Prior work on the enforcement of flow policies has concentrated on run-time mechanisms. One type of mechanism enforces a given flow policy by controlling processes' read and write access rights to objects: no process may acquire read access for an input object, or write access for an output object, unless the security class of every input flows into the security class of every output – even if some outputs depend on only a subset of the inputs. ADEPT-50 [30], the Case system [29], the MITRE system [3, 23], and the Privacy Restriction Processor [26] are of this type. These mechanisms are generally easy to implement because they make no attempt to examine the structure of a program. A second type of (more complex) mechanism accounts for program structures in order to determine flows between specific input and output objects. Fenton's data mark machine [10], the mechanism of Gat and Saal [13], and the surveillance mechanism of Jones and Lipton [19] are of this type. The surveillance mechanism employs a program transformation to insure that all flows are properly accounted for at run time. A detailed discussion of all these mechanisms can be found in [7].

This paper presents a compile-time mechanism that certifies a program only if it specifies no flows in violation of the flow policy. Besides the aesthetic attraction of establishing a program's security before it executes, a certification mechanism has important advantages. It can be specified directly in terms of language structures, which facilitates its comprehension and its proof of correctness. It greatly reduces the need for run-time checking. It does not impair a program's execution speed. (See also [23]).

Prior certification does not completely eliminate the need for run-time checking. Run-time support is needed to raise the tolerance against hardware malfunctions and other threats to the integrity of certified

4 part
1-2 page
submission
1) Summary
2) Strengths
3) Weaknesses
4) Future work

CLASS PROGRESS

DETECTING INFORMATION LEAKS **BEFORE** THE PROGRAM RUNS

Good fit for static analysis!

LAST TIME: INFORMATION FLOW

REVIEW: LAST LECTURE

AN APPLICATION OF STATIC DATAFLOW TRACKING

Formulation of confidentiality and integrity properties as dataflow properties

Source: Originator of tagged data

Sink: Consumer of tagged data

MANIFESTATIONS

Confidentiality
- Sources: functions that read “secret” resources
- Sinks: functions that write to “untrusted” places

Integrity
- Sources: functions that read “untrusted” places
- Sinks: functions that write to “sensitive” resources

PRACTICAL INFORMATION FLOW

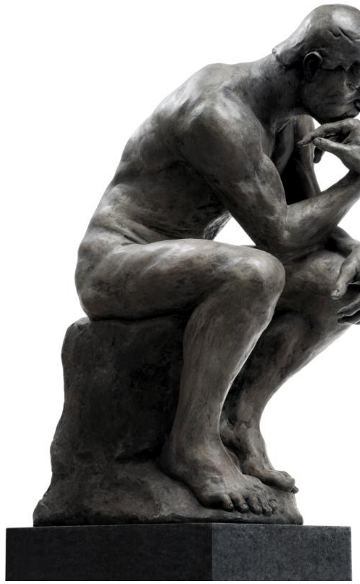
EECS 677: Software Security Evaluation

Drew Davidson

OVERVIEW

LET'S SAY WE WANT TO IMPLEMENT THE DATAFLOW IDEA

How would you actually do it?



LECTURE OUTLINE

- Source/Sink Identification
- Sneaky flows
- Sanitization

ANALYSIS DEPLOYMENT

PRACTICAL CONSIDERATIONS

This class is concerned with two incarnations of secure software evaluation:

Proactive SSE – Keep code that you are writing from misbehaving

Reactive SSE – Keep code that you've received from misbehaving

Good news:

Pretty straightforward case for the proactive incarnation – deploy analysis as part of compilation (or CI/CD) workflow

Plausible case for the reactive incarnation – raise a binary program to IR

FURTHER CONSIDERATIONS

PRACTICAL CONSIDERATIONS

Let's consider some of the practical aspects of getting the analysis to do some good

Source / Sink Identification – Where might flows start and end?

Sneaky behavior – How do we deal with code that wants to sneak past analysis?

SOURCE/SINK IDENTIFICATION

PRACTICAL CONSIDERATIONS

How do we know what should be a source and a sink?

Mind that semantic gap!

Idea #1 – Programmer annotations

Idea #2 – Build annotations into the system

Idea #3 – something something inferencing handwave

PROGRAMMER ANNOTATIONS

PRACTICAL CONSIDERATIONS – SOURCE/SINK IDENTIFICATION

Basic Idea

Ask the programmer to say what's a source and sink

- Auxiliary file of information
- Inline annotations within the program

```
; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @target() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32*, align 8
  %4 = alloca i32*, align 8
  %res = call i32 @function1 (i8* %strptr)
  store i32* %1, i32** %3, align 8
  %5 = load i32, i32* %1, align 4
  %6 = add nsw i32 %5, 1
  %7 = sext i32 %6 to i64
  %8 = inttoptr i64 %7 to i32*
  %res = call i32 @function2 (i32 %res)
  store i32* %8, i32** %4, align 8
  ret i32 0
}
```

```
; Function Attrs: info_sink
define i32 @function2(i8* %arg) #1 {
  ...
}
```

```
; Function Attrs: info_source
define i32 @function1() #3 {
  ...
}
```

PROGRAMMER ANNOTATIONS

PRACTICAL CONSIDERATIONS – SOURCE/SINK IDENTIFICATION

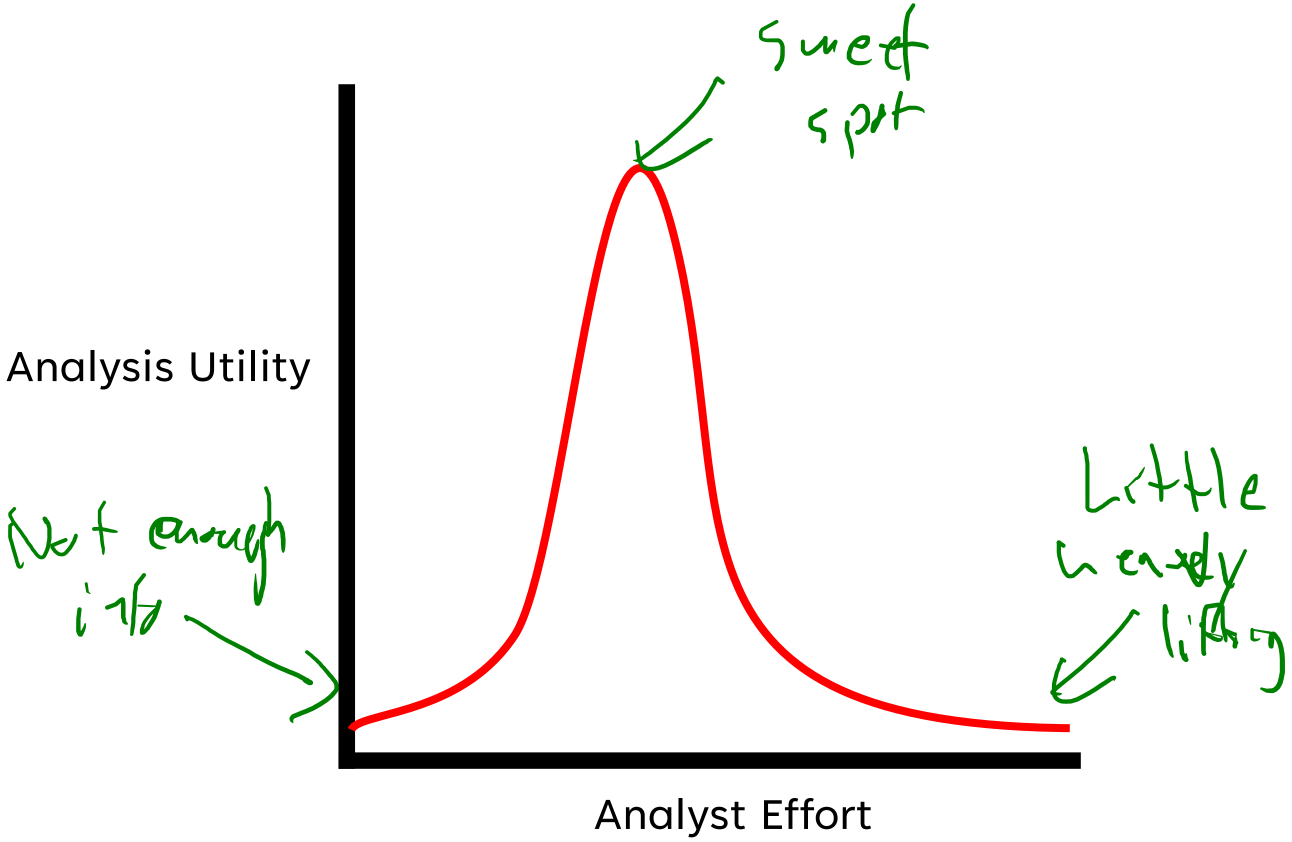
THE UTILITY OF PROGRAMMER EFFORT

A frequent struggle in analysis

ISSUES OF HUMAN INTERVENTION

Ultimately, we're trying to solve a limitation of human behavior

- Incorrect annotations
- Laziness
- Reactive SSE goes out the window



A totally-made-up conceptual graph

BUILT-IN "ANNOTATIONS"

PRACTICAL CONSIDERATIONS – SOURCE/SINK IDENTIFICATION

Enrich the system with notions of behavior

Platform developer bakes capabilities into the system

Analysis developer retrofits annotations into the analysis engine

Issues of Semantic Gap again

Can be quite hard to predict what becomes security-relevant

Analysis engine needs to be kept in lockstep with the system

INFERENCING

PRACTICAL CONSIDERATIONS – SOURCE/SINK IDENTIFICATION

You could try to automatically discover "Sourcelike" and "Sinklike" functions

Maybe we can detect UI asking for credit card?

Maybe we can write an analysis that looks for even more fundamental core behavior?

Machine learning???!?!?!?!?!?!?!?!?!?!?!?!?!?!?!?!?!?!?!

CASE STUDY: ANDROID PERMISSIONS

PRACTICAL CONSIDERATIONS – SOURCE/SINK IDENTIFICATION

MOBILE PHONES SURE COLLECT A LOT OF PRIVATE INFORMATION!

Maybe that information rises to the level of confidentiality?

Maybe this is a good application of an information flow analysis?

CASE STUDY: ANDROID PERMISSIONS

PRACTICAL CONSIDERATIONS – SOURCE/SINK IDENTIFICATION

HYBRID CASE OF BUILT-IN ANNOTATIONS

System has a built-in capability model

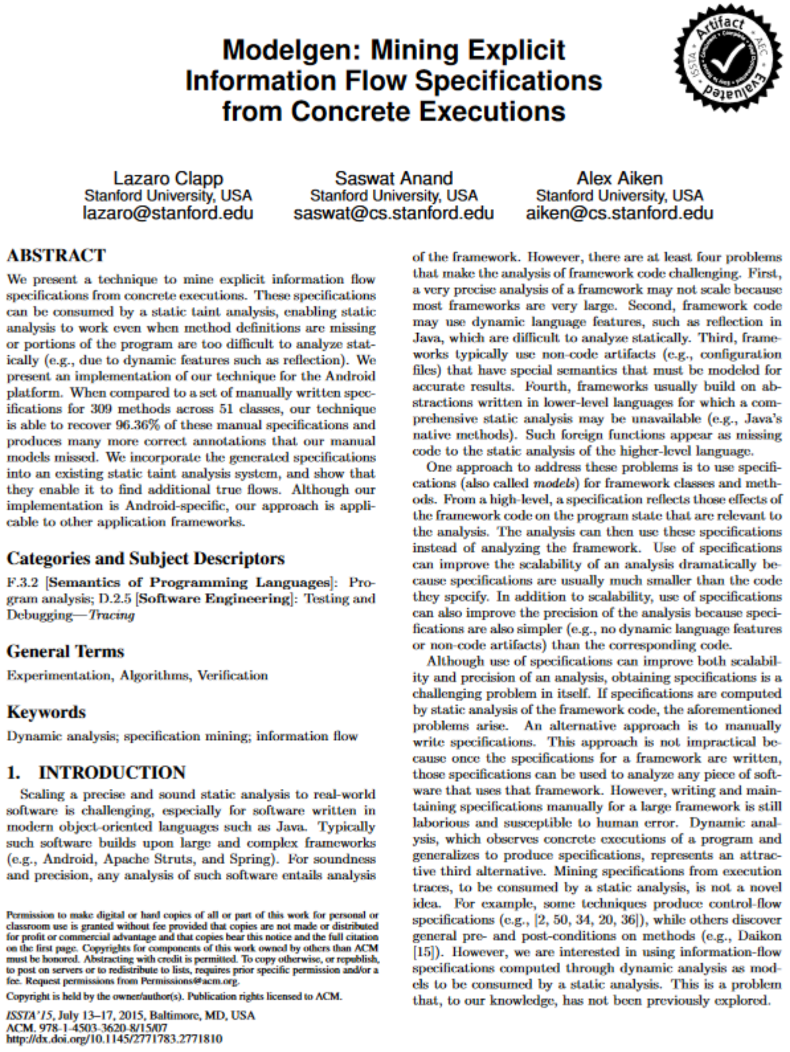
Surprisingly hard to map those capabilities to system functions

MODELGEN

- Manually annotate capabilities as sources or sinks

- Do a dynamic analysis of the Android system to discover capabilities uses

- Do a static dataflow analysis of the Android system to discover capabilities uses



Modelgen: Mining Explicit Information Flow Specifications from Concrete Executions

Lazaro Clapp
Stanford University, USA
lazaro@stanford.edu

Saswat Anand
Stanford University, USA
saswat@cs.stanford.edu

Alex Aiken
Stanford University, USA
aiken@cs.stanford.edu

ABSTRACT

We present a technique to mine explicit information flow specifications from concrete executions. These specifications can be consumed by a static taint analysis, enabling static analysis to work even when method definitions are missing or portions of the program are too difficult to analyze statically (e.g., due to dynamic features such as reflection). We present an implementation of our technique for the Android platform. When compared to a set of manually written specifications for 309 methods across 51 classes, our technique is able to recover 96.36% of these manual specifications and produces many more correct annotations that our manual models missed. We incorporate the generated specifications into an existing static taint analysis system, and show that they enable it to find additional true flows. Although our implementation is Android-specific, our approach is applicable to other application frameworks.

Categories and Subject Descriptors

F.3.2 [**Semantics of Programming Languages**]: Program analysis; D.2.5 [**Software Engineering**]: Testing and Debugging—*Tracing*

General Terms

Experimentation, Algorithms, Verification

Keywords

Dynamic analysis; specification mining; information flow

1. INTRODUCTION

Scaling a precise and sound static analysis to real-world software is challenging, especially for software written in modern object-oriented languages such as Java. Typically such software builds upon large and complex frameworks (e.g., Android, Apache Struts, and Spring). For soundness and precision, any analysis of such software entails analysis of the framework. However, there are at least four problems that make the analysis of framework code challenging. First, a very precise analysis of a framework may not scale because most frameworks are very large. Second, framework code may use dynamic language features, such as reflection in Java, which are difficult to analyze statically. Third, frameworks typically use non-code artifacts (e.g., configuration files) that have special semantics that must be modeled for accurate results. Fourth, frameworks usually build on abstractions written in lower-level languages for which a comprehensive static analysis may be unavailable (e.g., Java's native methods). Such foreign functions appear as missing code to the static analysis of the higher-level language.

One approach to address these problems is to use specifications (also called *models*) for framework classes and methods. From a high-level, a specification reflects those effects of the framework code on the program state that are relevant to the analysis. The analysis can then use these specifications instead of analyzing the framework. Use of specifications can improve the scalability of an analysis dramatically because specifications are usually much smaller than the code they specify. In addition to scalability, use of specifications can also improve the precision of the analysis because specifications are also simpler (e.g., no dynamic language features or non-code artifacts) than the corresponding code.

Although use of specifications can improve both scalability and precision of an analysis, obtaining specifications is a challenging problem in itself. If specifications are computed by static analysis of the framework code, the aforementioned problems arise. An alternative approach is to manually write specifications. This approach is not impractical because once the specifications for a framework are written, those specifications can be used to analyze any piece of software that uses that framework. However, writing and maintaining specifications manually for a large framework is still laborious and susceptible to human error. Dynamic analysis, which observes concrete executions of a program and generalizes to produce specifications, represents an attractive third alternative. Mining specifications from execution traces, to be consumed by a static analysis, is not a novel idea. For example, some techniques produce control-flow specifications (e.g., [2, 50, 34, 20, 36]), while others discover general pre- and post-conditions on methods (e.g., Daikon [15]). However, we are interested in using information-flow specifications computed through dynamic analysis as models to be consumed by a static analysis. This is a problem that, to our knowledge, has not been previously explored.

ISSTA'15, July 13–17, 2015, Baltimore, MD, USA
ACM. 978-1-4503-3620-8/15/07
http://dx.doi.org/10.1145/2771783.2771810

LECTURE OUTLINE

- Source/Sink Identification
- Sneaky flows
- Sanitization

SNEAKY FLOWS

PRACTICAL CONSIDERATIONS – SNEAKY FLOWS

Might an adversary attempt to avoid detection?

The proliferation of tools for Android analysis gives them an obvious incentive

SNEAKY FLOWS

PRACTICAL CONSIDERATIONS – SNEAKY FLOWS

IMPLICIT FLOWS

CONTROL DEPENDENCIES

PRACTICAL CONSIDERATIONS – SNEAKY FLOWS

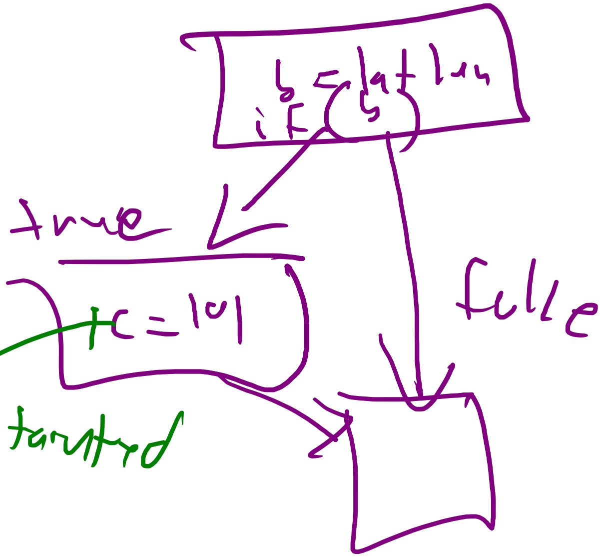
```
bool b;
bool c;
b = isActuallyAnEvilSpy()
c = b;
sendToNetwork(c);
```

```
bool b = isActuallyAnEvilSpy()
bool c;
if (b == true){
  c = true;
} else {
  c = false;
}
sendToNetwork(c);
```

isSpy ← source

sendToNet ← sink

c is tainted



b = LatLon

if (b > 100 && b < 102){

c = 101;

}

sendToNetwork(c);

LECTURE OUTLINE

- Source/Sink Identification
- Sneaky flows
- Precision / Sanitization

GRANULARITY OF ANALYSIS

PRACTICAL CONSIDERATIONS – PRECISION/SANITIZATION

DATA IS COMPLEX!

What happens when a field of a struct is tainted?

What happens when an index of an array is tainted?

SANITIZATION

PRACTICAL CONSIDERATIONS – PRECISION/SANITIZATION

WE ALSO WANT TO PROVIDE SOME EXCEPTIONS TO THE FLOW RULES

i.e. tainted data is encrypted

res ← encrypt(arg)

WRAP-UP

NEXT TIME

HOW DO WE FIX OUR LEAKY PROGRAMS?