TAJ: Effective Taint Analysis of Web Applications

Omer Tripp Marco Pistoia Stephen Fink Manu Sridharan Omri Weisman

IBM Software Group
omert@il.ibm.com

IBM T. J. Watson Research Center
{pistoia,sjfink,msridhar}@us.ibm.com

IBM Software Group
weisman@il.ibm.com

Abstract

Taint analysis, a form of information-flow analysis, establishes whether values from untrusted methods and parameters may flow into security-sensitive operations. Taint analysis can detect many common vulnerabilities in Web applications, and so has attracted much attention from both the research community and industry. However, most static taint-analysis tools do not address critical requirements for an industrial-strength tool. Specifically, an industrial-strength tool must scale to large industrial Web applications, model essential Web-application code artifacts, and generate consumable reports for a wide range of attack vectors.

We have designed and implemented a static Taint Analysis for Java (TAJ) that meets the requirements of industry-level applications. TAJ can analyze applications of virtually any size, as it employs a set of techniques designed to produce useful answers given limited time and space. TAJ addresses a wide variety of attack vectors, with techniques to handle reflective calls, flow through containers, nested taint, and issues in generating useful reports. This paper provides a description of the algorithms comprising TAJ, evaluates TAJ against production-level benchmarks, and compares it with alternative solutions.

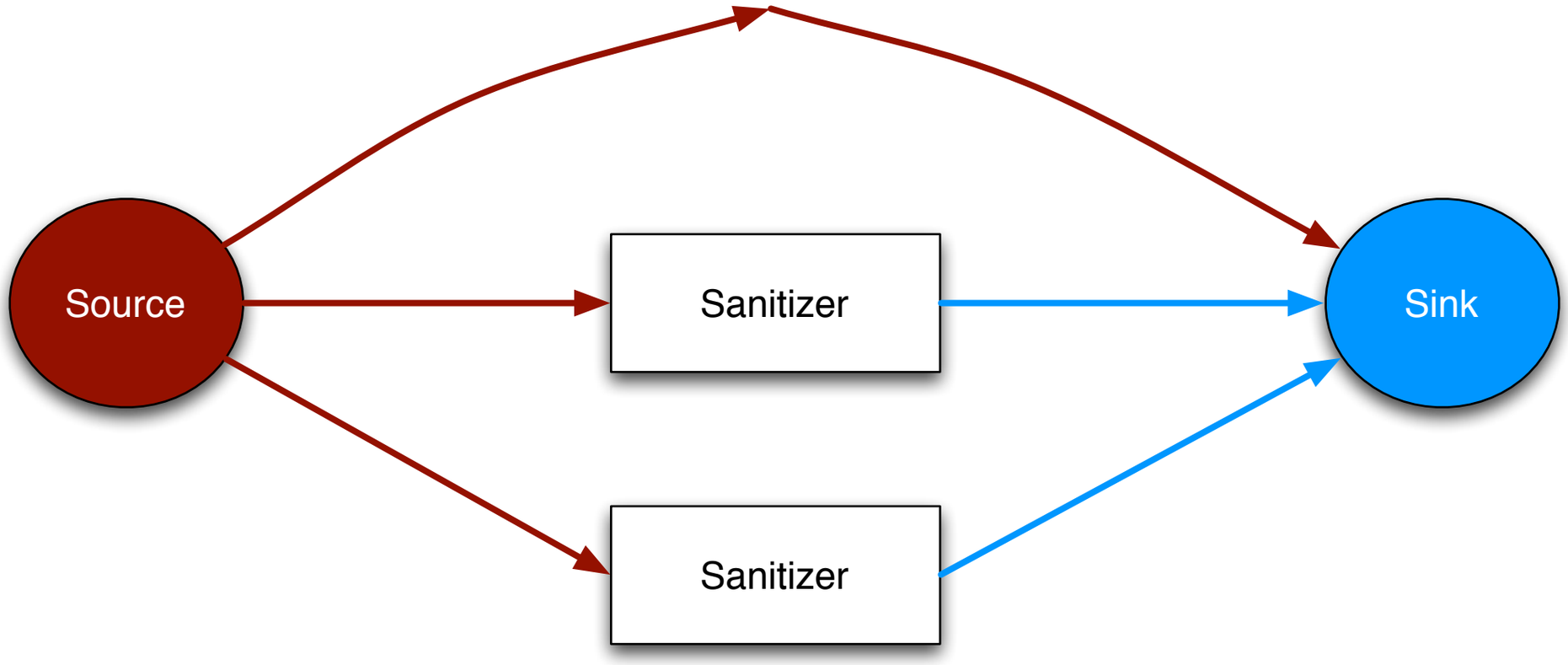
Categories and Subject Descriptors D.2.4 [*Software Engineer-*

- *Cross-site scripting (XSS) attacks* (the most common vulnerability) may occur when a Web application accepts data originating from a user and sends it to another user's browser without first validating or encoding it. For example, suppose an attacker embeds malicious JavaScript code into his or her profile on a social Web site. If the site fails to validate such input, that code may execute in the browser of any other user who visits that profile.
- *Injection flaws* (the second most frequent vulnerability) arise when a Web application accepts input from a user and sends it to an interpreter as part of a command or query, without first validating it. *Via* this vulnerability, an attacker can trick the interpreter into executing unintended commands or changing data. The most common attack of this type is Structured Query Language injection (SQLi).
- *Malicious file executions* (the third most common vulnerability) happen when a Web application improperly trusts input files or uses unverified user data in stream functions, thereby allowing hostile content to be executed on the server.
- *Information leakage and improper error-handling attacks* (the sixth most common vulnerability) take place when a Web application leaks information about its own configuration, mech-

Reality Check

- Anyone read this paper?
- Taint Analysis
- Thin Slicing
- Flow-sensitive Analysis
- Context-sensitive Pointer Analysis

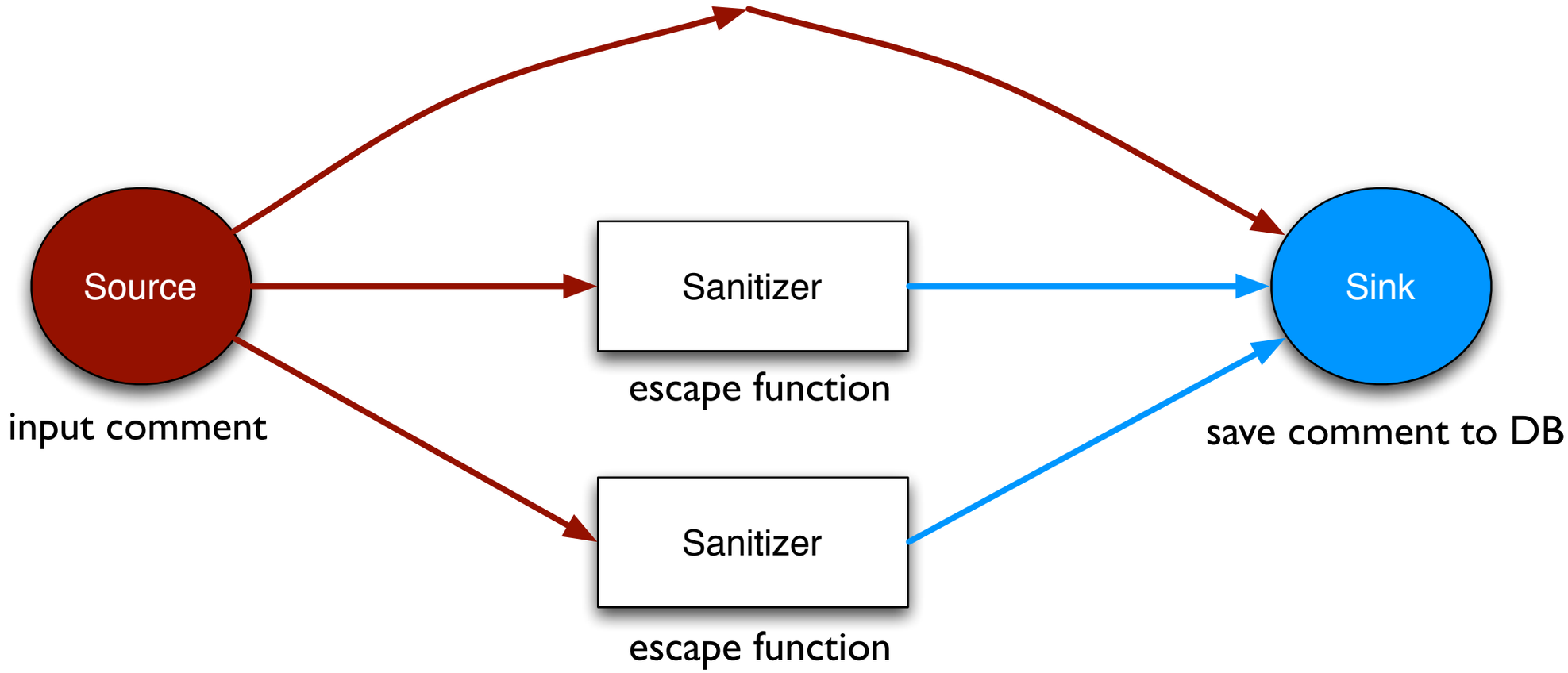
Taint Analysis Problem



All paths from a source to a sink should pass “Sanitizer” function

Taint Analysis Problem

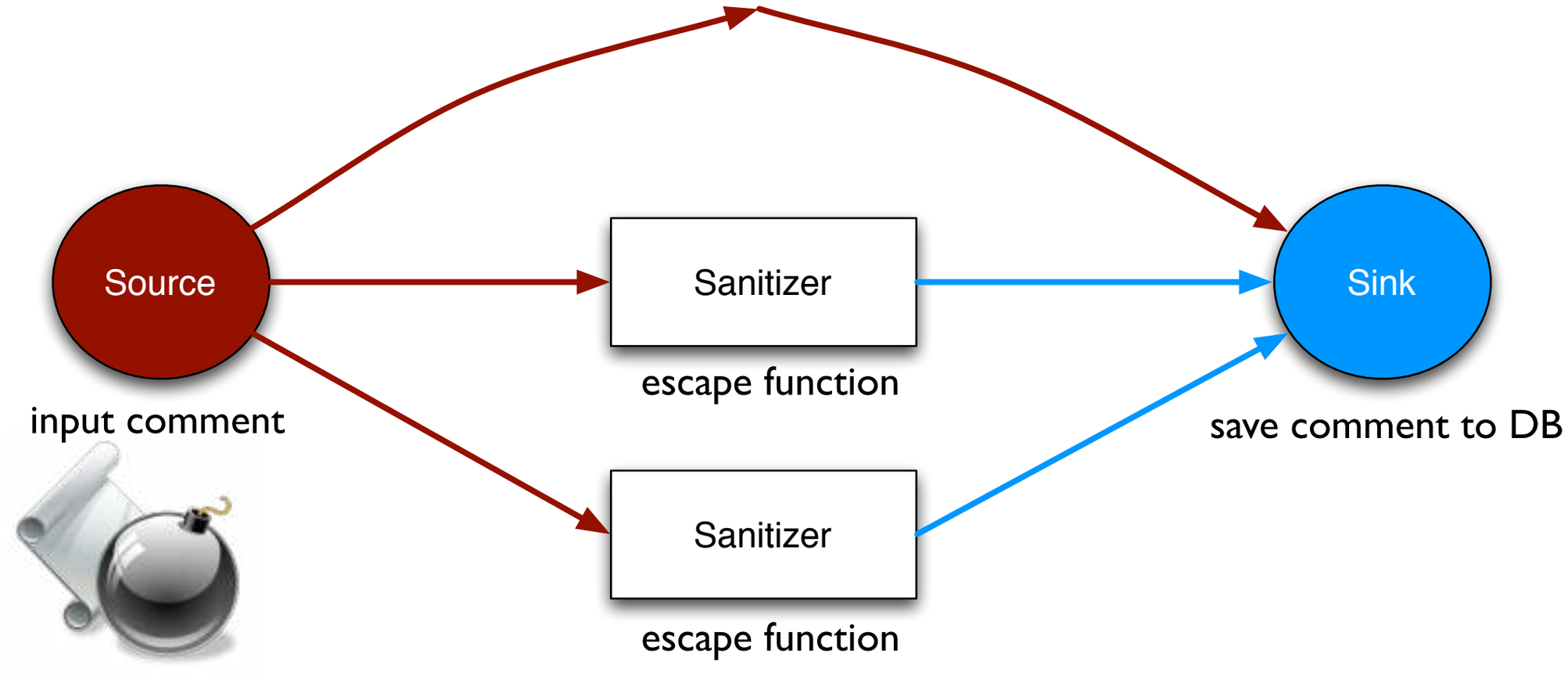
Example



All paths from a source to a sink should pass “Sanitizer” function

Taint Analysis Problem

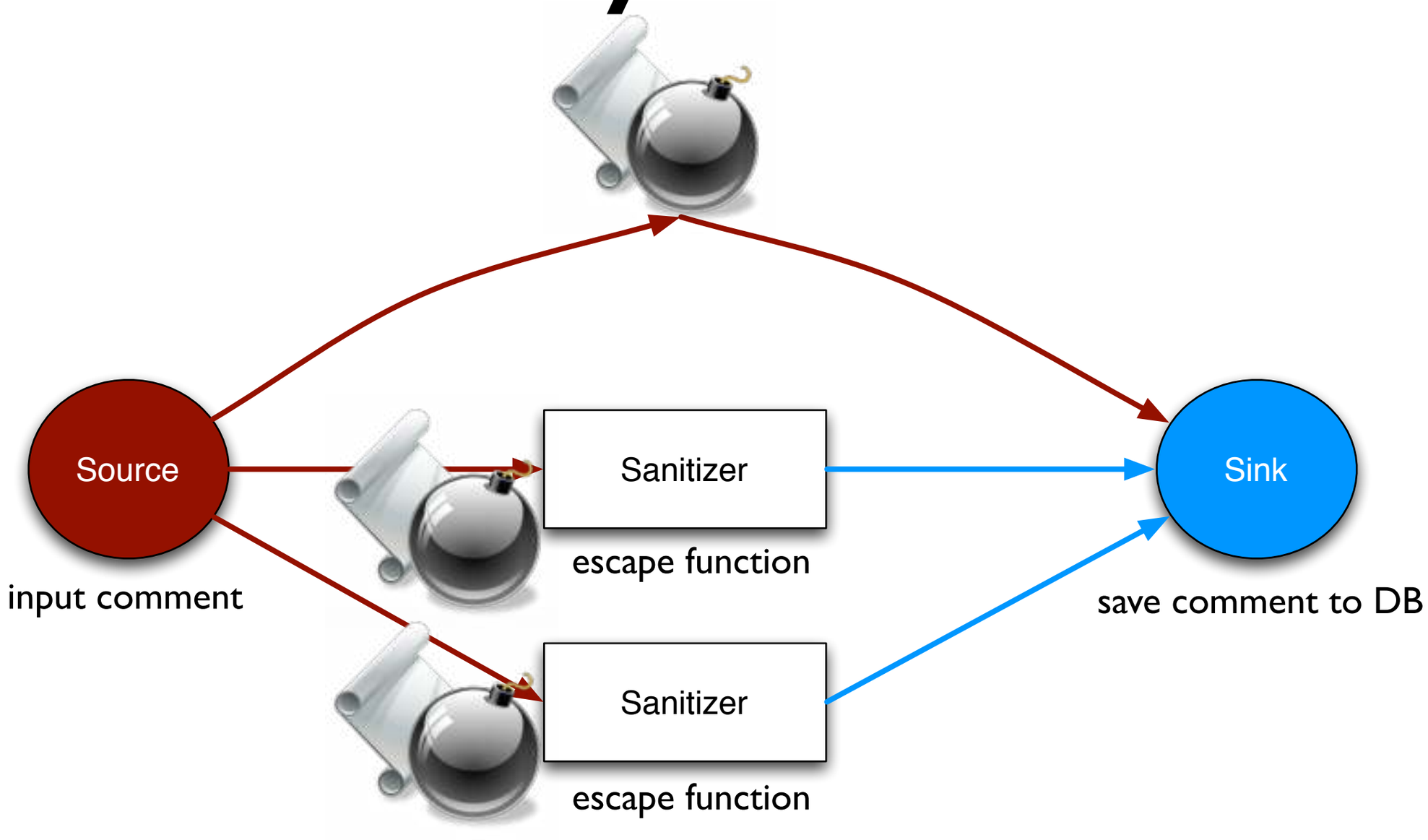
Example



All paths from a source to a sink should pass “Sanitizer” function

Taint Analysis Problem

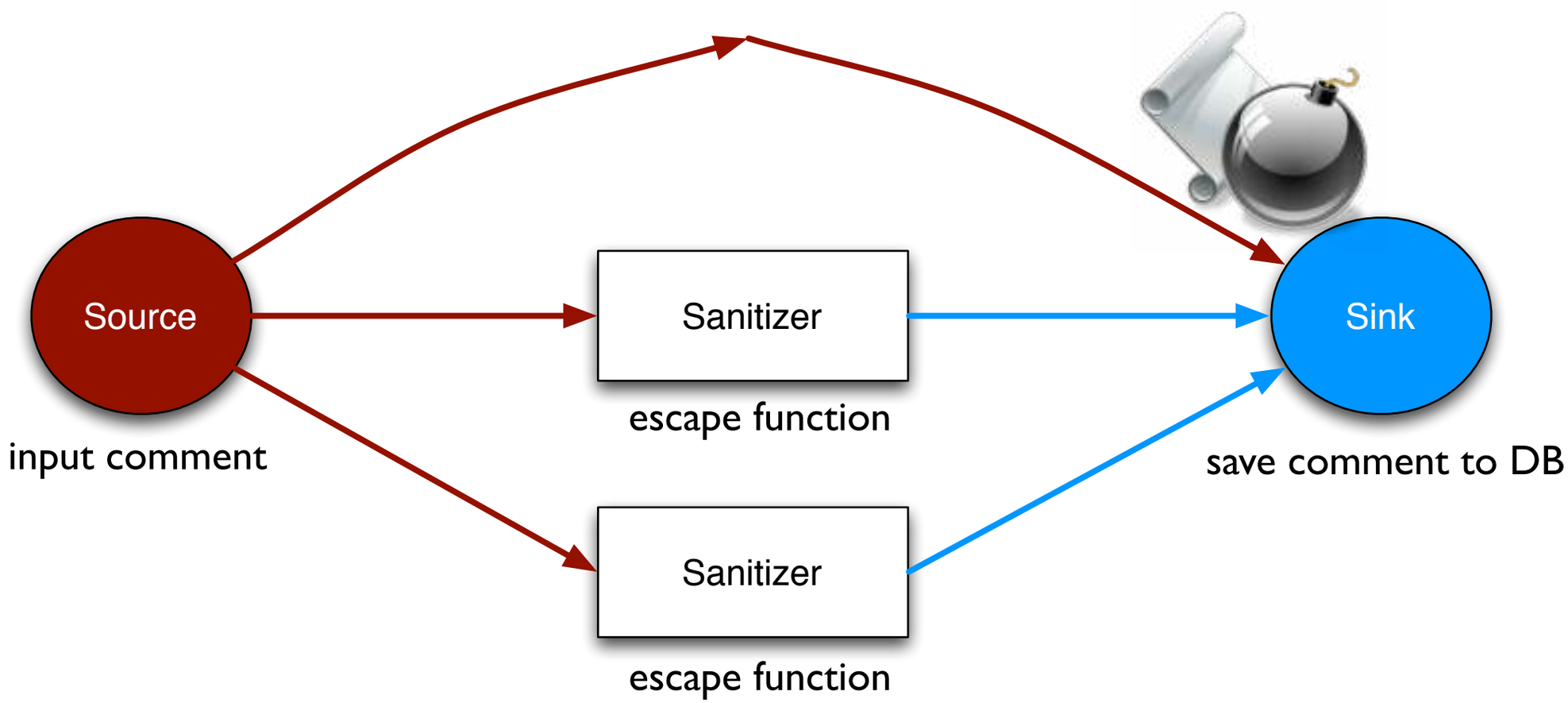
Example



All paths from a source to a sink should pass “Sanitizer” function

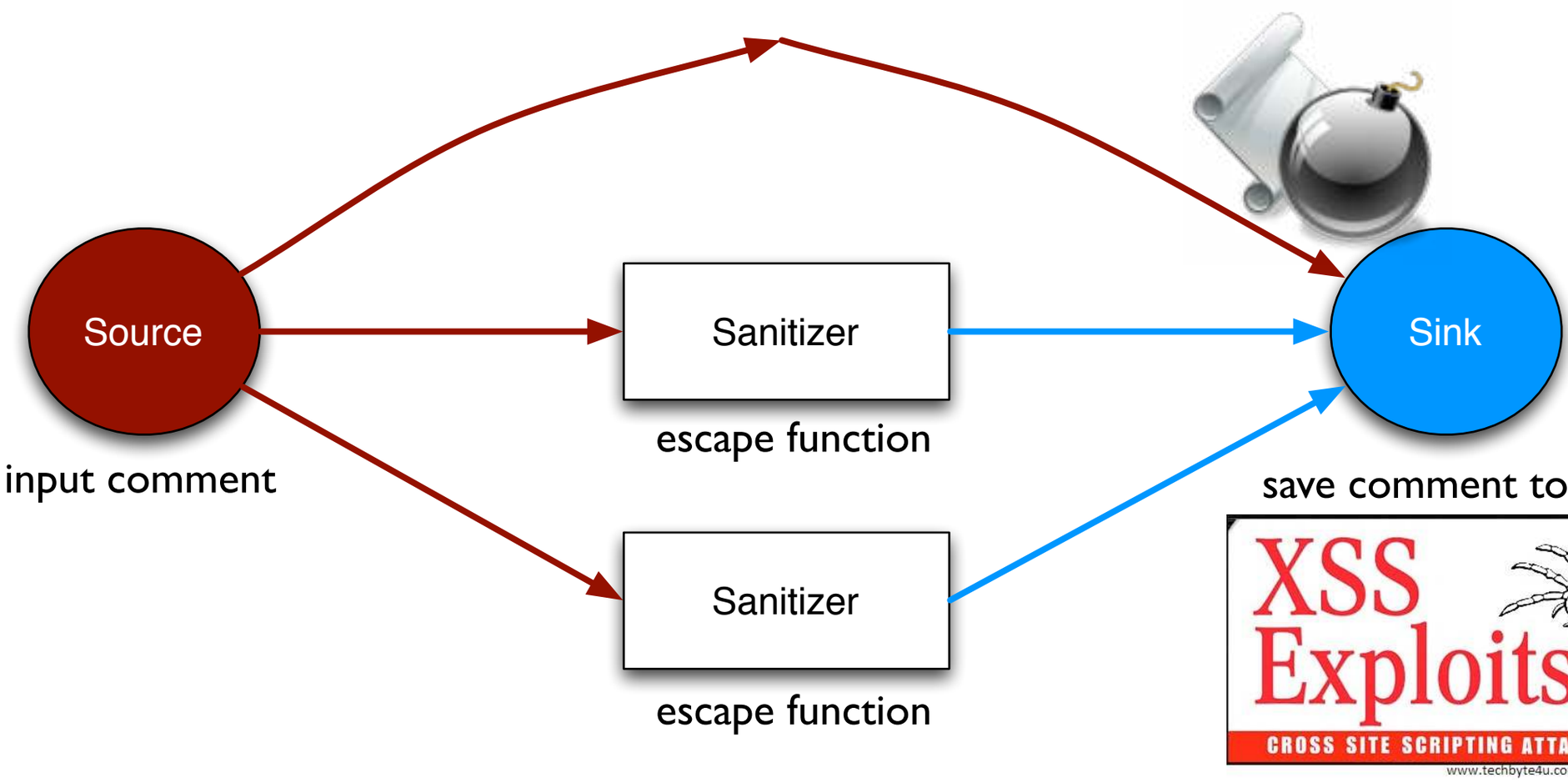
Taint Analysis Problem

Example



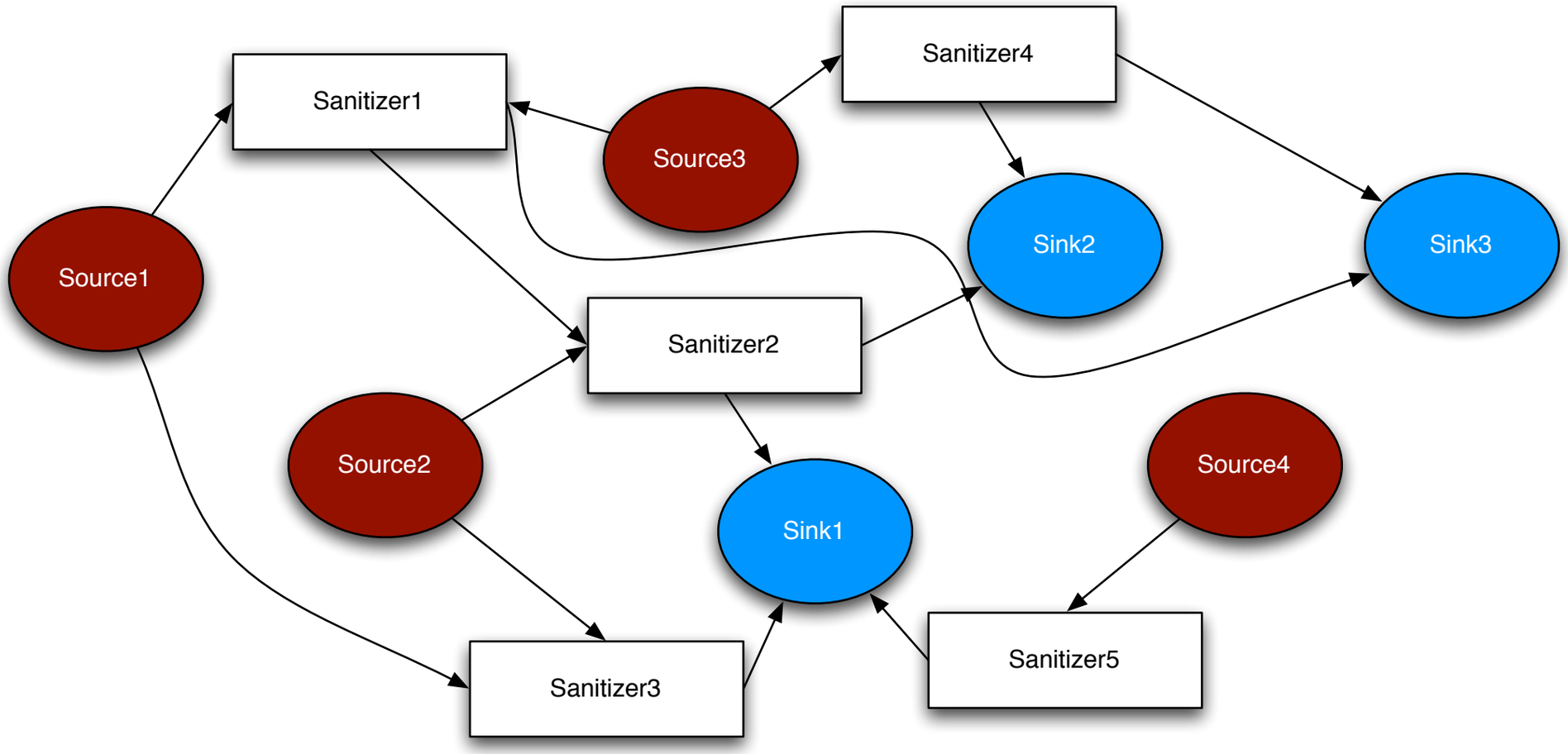
All paths from a source to a sink should pass “Sanitizer” function

Taint Analysis Problem



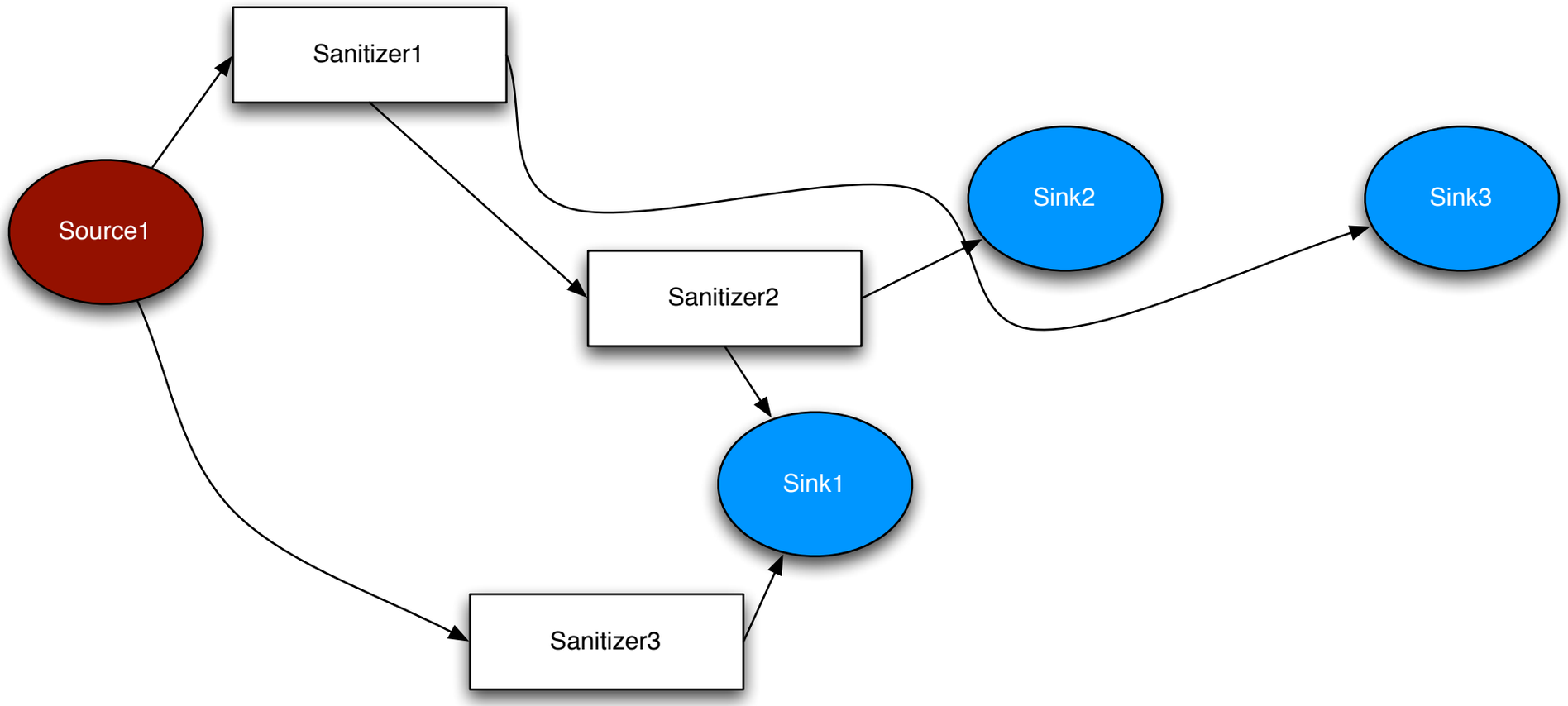
XSS is #1 Vulnerability
(even more than Buffer Overflow)

Key Idea : Thin Slicing



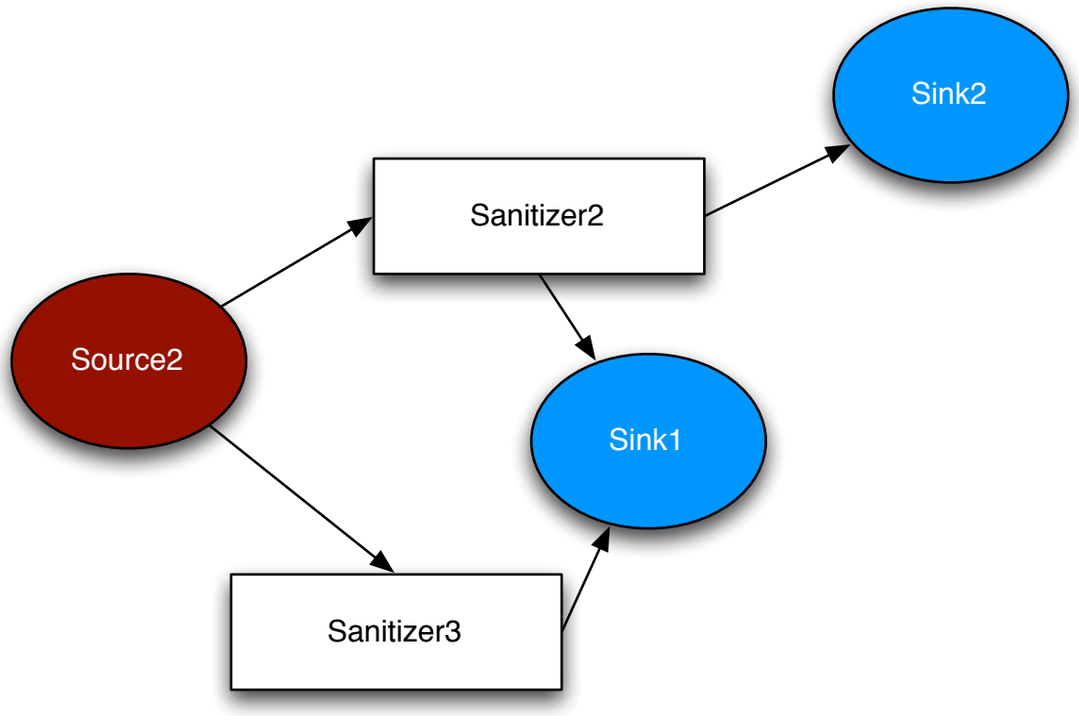
Thin Slicing Program
with Each Source

Key Idea : Thin Slicing



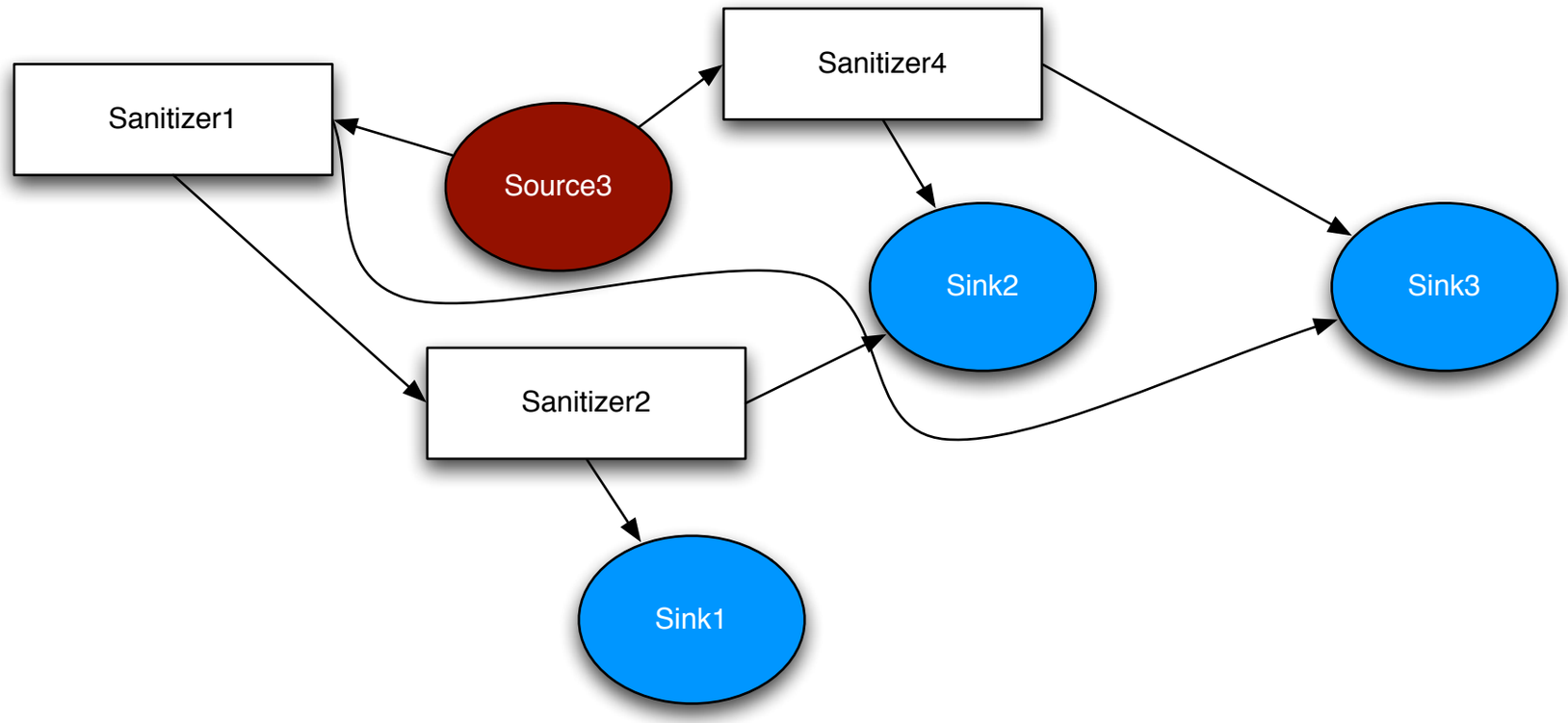
Thin Slicing Program
with Each Source

Key Idea : Thin Slicing



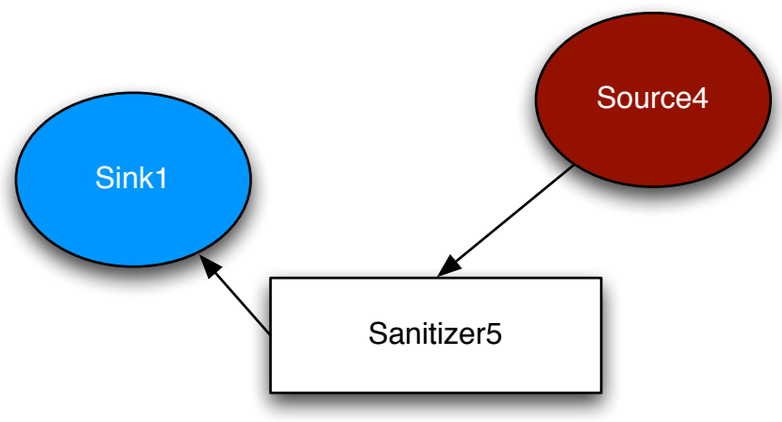
Thin Slicing Program
with Each Source

Key Idea : Thin Slicing



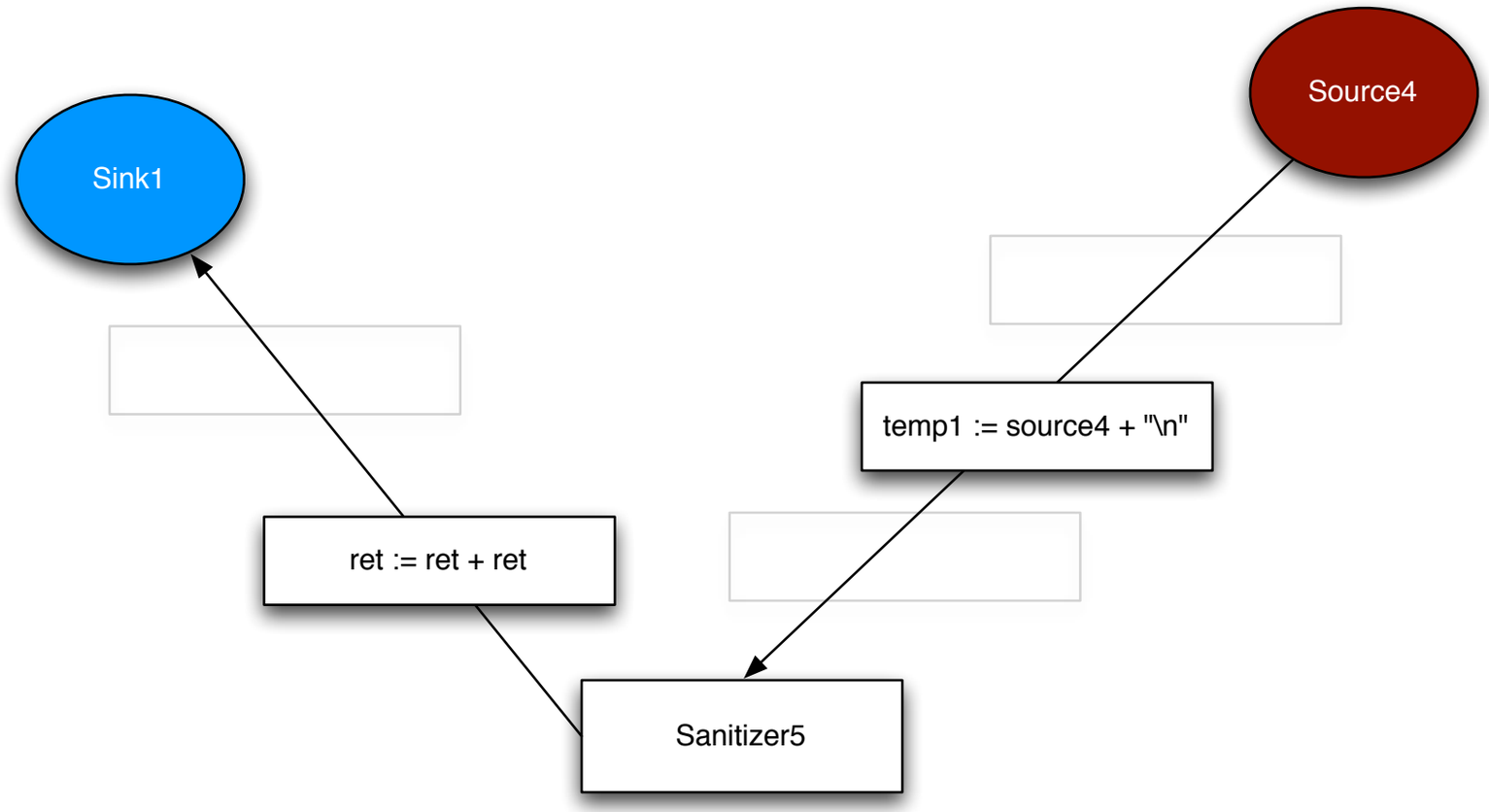
Thin Slicing Program
with Each Source

Key Idea : Thin Slicing



Thin Slicing Program
with Each Source

Key Idea : Thin Slicing

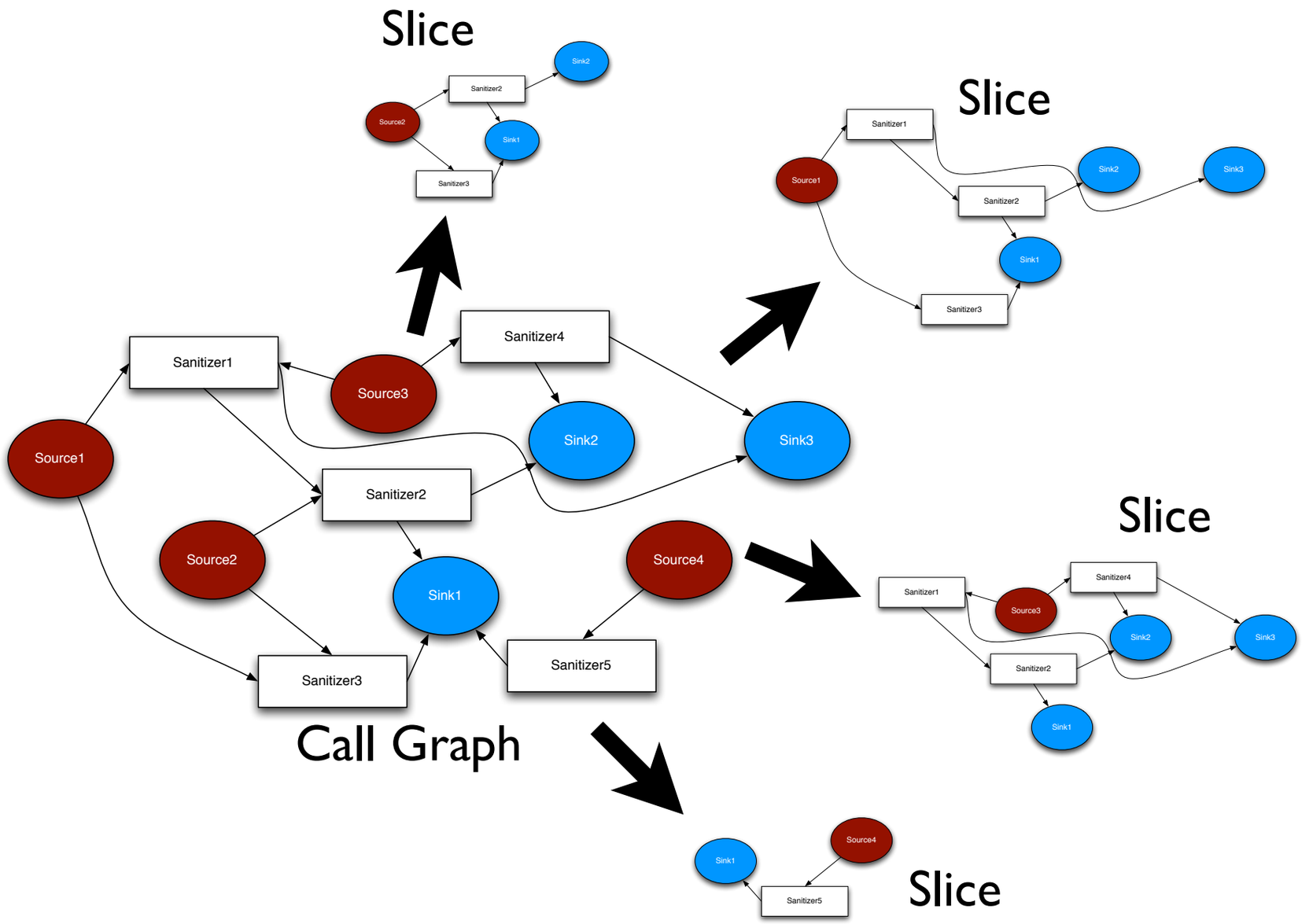


Thin Slice from a source

= Statements data-dependent on the source

=> Much smaller and more understandable

Key Idea : Hybrid Thin Slicing

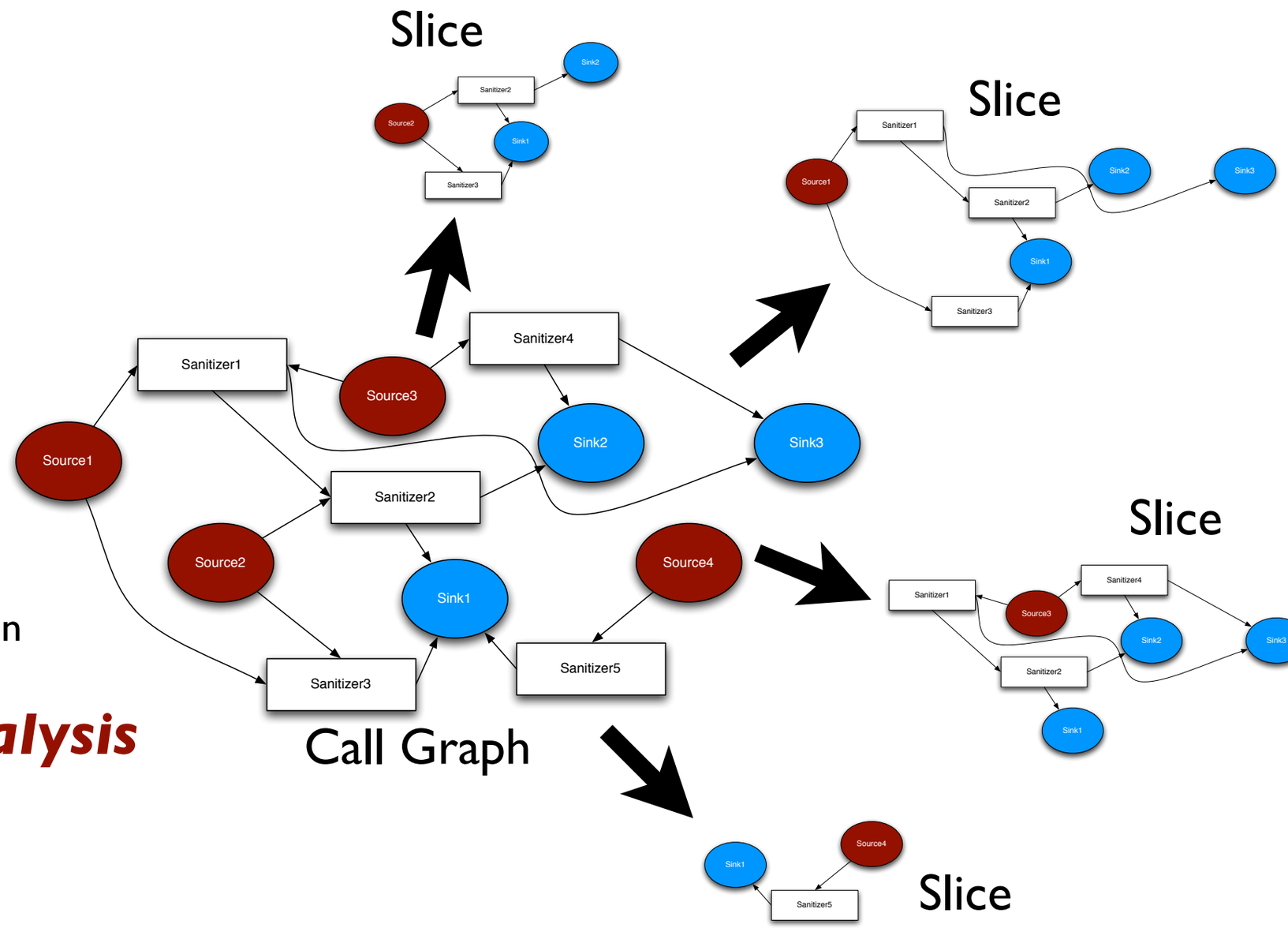


Key Idea : Hybrid Thin Slicing

Program

Graph Construction via

Pointer Analysis



Call Graph

Slice

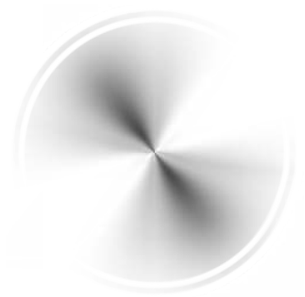
Slice

Slice

Slice

Key Idea : Hybrid Thin Slicing

Scalability



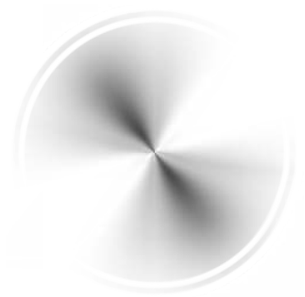
CI(Context-insensitive)
Thin Slicing

CS(Context-sensitive)
Thin Slicing

Precision

Key Idea : Hybrid Thin Slicing

Scalability



CI(Context-insensitive)
Thin Slicing

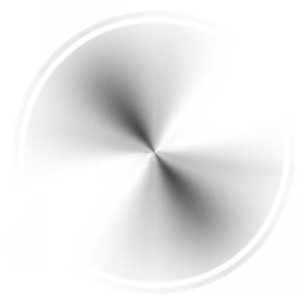
Hybrid Thin Slicing
flow-insensitive about heap
flow- and context-sensitive about local variable

CS(Context-sensitive)
Thin Slicing

Precision

Key Idea : Hybrid Thin Slicing

Scalability



CI(Context-insensitive)
Thin Slicing

better
precision

Hybrid Thin Slicing
flow-insensitive about heap
flow- and context-sensitive about local variable

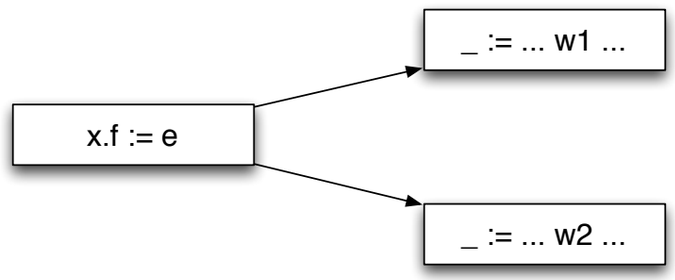
better
scalability

CS(Context-sensitive)
Thin Slicing

Precision

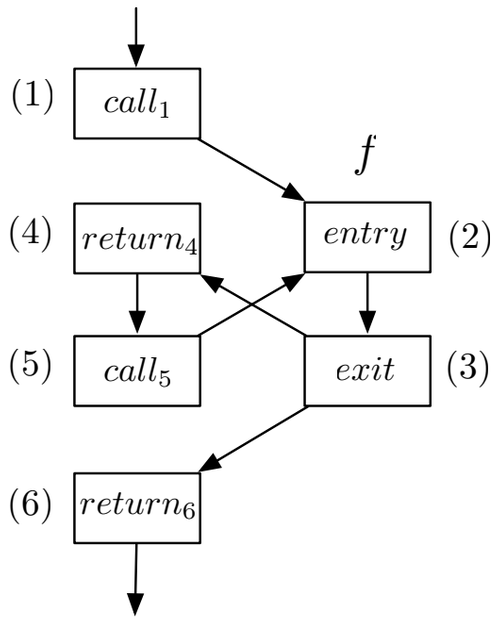
Key Idea : Hybrid Thin Slicing

* Flow-insensitive about heap
  - For a statement `x.f := e`, we add an edge to each statement with an expression `w.f` on its right-hand side, such that the pre-computed points-to analysis indicates x *may-alias* `w`.

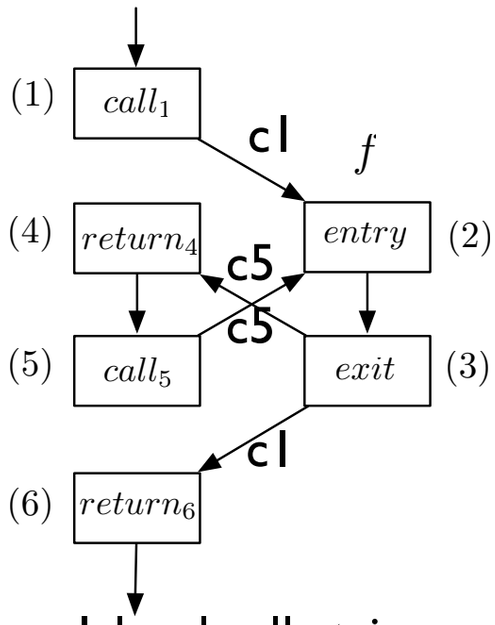


x may alias w1 and w2

* Flow- and context-sensitive about local variable



context-insensitive



1-level call-string context-sensitive

Other Contributions

- Effective Model for Static Analysis of Web Applications

  Support reflection, tainted flow through containers, detection of taint in the internal state of objects, JSP, EJB, Struts and Spring frameworks.

- Bounded Analysis Techniques

  Support a prioritization policy that focuses the analysis on portions of the Web application that are likely to participate in taint propagation.

Thank You