Automated Security Analysis

Tool Support for Evaluating C Code

Joe Hurd, Louis Testa and Aaron Tomb

Galois, Inc.
{joe,louis,atomb}@galois.com

HCSS
May 19, 2009

galois

Talk Plan

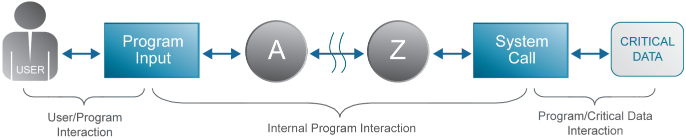
1. Introduction
2. Visualization
3. Information Flow
4. Prototype Analysis Tool
5. Summary

Project Goal

- **Scenario:** An evaluator is handed 100k lines of C code and given two months to perform a security evaluation.
  - Examples: OpenSSH, bftpd, ISC DHCP server.
  - The application seems to fulfill its functional requirements, but how to ensure that there is no malicious behaviour or vulnerabilities?
- **Project Goal:** Develop a diagnostic tool for C code to support security evaluators.

Security Evaluations

- Security evaluations generally focus on:
  - The attack surface (e.g., the interface to the user or network).
  - The critical data (e.g., crypto keys, database queries).
- **Typical Question:** What are the possible effects of changes at the attack surface on the critical data?



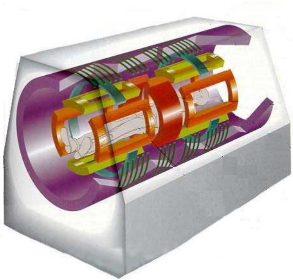
- Answering this requires an understanding of how information flows through the program.

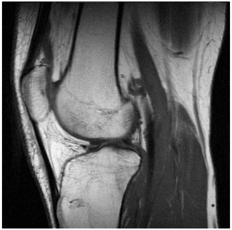
Information Flow and Security Properties

- Many program security properties can be expressed in terms of potential information flow between program variables.
- **Confidentiality:** ✓
  - Example: ensuring Bell-La Padula properties hold for cross domain applications.
  - Look for flows from `secret` to `public`.
- **Integrity:** ✓
  - Example: ensuring that 'tainted' user data does not get stored in fixed-length buffers or appear in SQL queries.
  - Look for flows from `tainted` to `critical`.
- **Availability:** ×
  - No way to specify that a flow *must* happen.

Information Flow Diagnostic Tool

- Tight time constraints mean that evaluators often cannot look at every line of the codebase.
- A diagnostic tool supports faster exploration of information flow properties, allowing the evaluator to look for anomalies.





ASA Project Vision

- **Goal:** Develop a program analysis tool that evaluators interact with to build a mental model of security-relevant information flow.
- **Workflow:**
  1. The evaluator seeds the analysis by annotating some program variables as sensitive data or dangerous user input.
  2. The tool uses the annotations to find candidate insecure information flows.
  3. The evaluator examines the flows, and removes false positives by providing additional annotations so that the tool can make a more precise analysis.

ASA Project Status

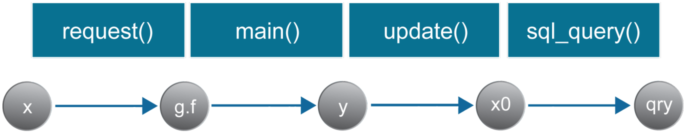
- **Current Status:** We have a research prototype tool implementing the underlying information flow static analysis.
    - Static analysis techniques allow the tool to scale up to large codebases.
    - Tested by applying to open source codebases to discover integrity problems.
- **Next Step:** Visualization of program information flow.

Visualization Requirements

- **Visualization Goal:** Help evaluators build a mental model of how security-relevant information flows through the program.
- **Requirement:** Information flows must be closely tied to the source code of the program, since that is what evaluators are looking at.
- **Requirement:** The visualization of information flows must help to build a consistent model, not present a new view for every information flow.

Visualizing Information Flow

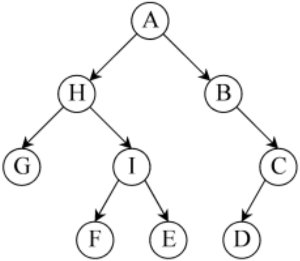
- A program information flow consists of many assignments distributed across the codebase:



- Tracking a long information flow across source code involves much tedious opening, closing and searching of files.
    - *"Evaluating software is like frying 1,000 eggs"*
- A different visualization solution is needed.

Call Trees: A Different Point of View

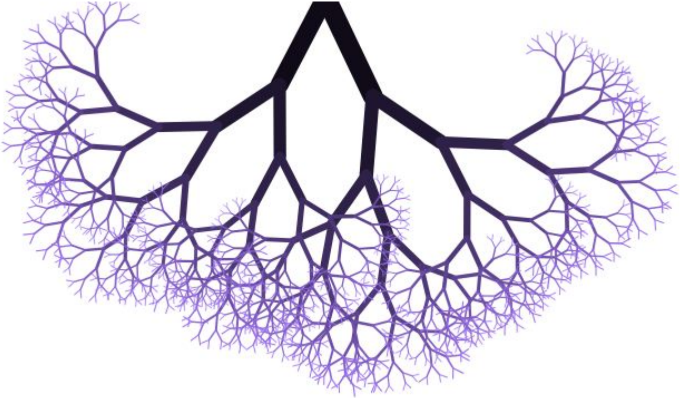
- A call tree is a representation of how the program will execute when run:



- When the program is executed the call tree is traversed in depth first order.
- Information flows forwards through call tree nodes.

Large Codebases Mean Large Call Trees

- For large codebases, the call tree is unmanageably large.
- Need fractal geometry to even display it on a screen:

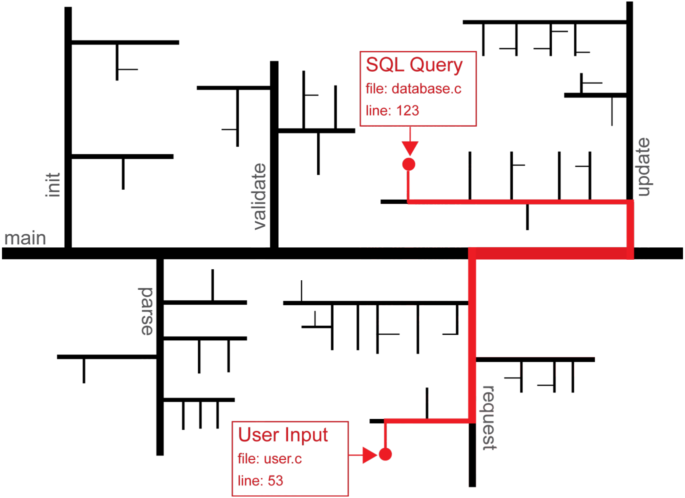


galois

Fractal Call Trees: Why Not?

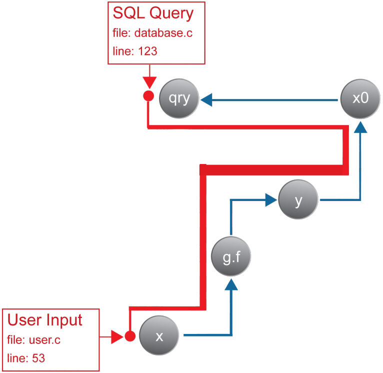
- **But Wait!** The fractal call tree satisfied some of the visualization requirements:
  - It presents a consistent model of information flow through the program.
  - Navigation is intuitive: zoom in to different parts of the call tree 'map'.
- But there are still problems remaining:
  - Fractal call trees don't appear to use space efficiently.
  - How can they represent program information flows?

Right-Angle Fractal Call Trees



Labeling Fractal Call Trees

- Fractal call trees can be labeled with function names, or the variables used in the information flow:



Program Information Flow

- We have developed a prototype analysis tool for computing information flow for C programs.
- The algorithm is based on a theory of information flow between storage locations (local/global variables and heap cells).
- This part of the talk gives an overview of the theory.

Computing Information Flow

- Computing precise information flow for a C program is a challenging problem.
  - Information can flow through complex data structures.
  - Information transferring function pointers must be accurately tracked.
- However, computing an over-approximation of program information flow is feasible.
  - Write $x \longrightarrow y$ to mean information may flow from x to y.
  - It is safe to add extra flows if there is uncertainty.
- **Requirement:** The evaluator is looking for bugs, not verifying the program, so support tools need not be 100% sound or complete to be useful.

Conditional Information Flow

- Consider the following code snippet:

```
if (condition) { x = secret; }
if (!condition) { public = x; }
```

- **Precision Improvement:** Track the `condition` to see that `secret` can never flow into `public`.
- The analysis implements a calculus of conditional information flow:

$$\Gamma \vdash x \longrightarrow y$$

"If the condition Γ holds, the value in the program variable x flows into the program variable y."

Indirect Information Flow

- There are two types of information flow:
  - **Direct:** `y = x;`
  - **Indirect:** `if (x > 0) { y = 0; }`
- Indirect flows use the control path as a channel.
  - They are thus invisible to *dynamic taint analysis* (e.g., Perl).
- How does the type of flow affect security properties?
  - **Confidentiality:** Look for both direct or indirect flows `secret` → `public`.
  - **Integrity:** Look for direct flows `tainted` → `critical`.
- **Warning:** Any direct flow can be artificially coded as an indirect flow, so both kinds of flows should be checked if the program might have malicious integrity bugs.

Computing Indirect Information Flow

- Suppose a direct information flow

$$\Gamma \vdash x \longrightarrow y$$

  with program variable z in the condition Γ.
- Then there is an indirect information flow from z to y.
- Moreover, all indirect information flows must arise in this way from some direct information flow.

Analysis Tool Implementation

- SATURN is a static analysis tool infrastructure developed by Alex Aiken's group at Stanford.
  - SATURN uses CIL to preprocess and simplify the input C code.
- The ASA analysis tool is implemented as three separate SATURN modules:
  - Variable clobbering analysis.
  - Information flow analysis.
  - Sensitivity analysis.
- All SATURN analyses are compositional.
  - Function bodies are analyzed separately and stored in summaries.
  - Summary information is consulted at call-sites.

Toy Example

Code

```
int high, low;
void experiment(int cond) {
  int tmp = 0;
  if (cond) { tmp = high; }
  low = tmp;
}
```

Shell

```
Entering flowprint: cil_sum_body("experiment",s_func)...
experiment.c:experiment: high flows into low (__arg0*)
*** Analysis finished successfully.
```

From Information Flow to Security Bugs

- Application-specific sensitivity analysis.
  - Variables are annotated with their sensitivity levels by the user.
  - Calculate information flow for the `main` function.
  - Check there are no flows that violate the security policy (e.g., high to low).
- Validating input data.
  - Format string bugs, SQL injection attacks.
  - Annotate input data with high, critical function arguments with low.
  - Ensure all flows from high to low go through validation functions.

Benchmarks

- The tool scales up to analyze large open source codebases:

Program	Version	Lines	Func's	Security Analysis
bftpd	1.6	4,229	473	Format string
Neon	0.24.4	13,324	403	Format string
cfengine	1.5.4	36,648	448	Format string
ISC DHCP	3.0.1rc3	75,455	1,237	Format string
OpenSSH	4.7p1	52,399	1,292	Sensitivity

- Known format string bugs found in several open source benchmarks, including FTP and DHCP daemons.

Case Study: The OpenSSH library

- 52,339 lines of C code, in 1,292 functions.
- Set the sensitivity of created keys to high, and parameters of functions that write to disk to low.
- Should discover an information flow violating *confidentiality* in the code for generating a new key.
- Sensitivity analysis completes in 9 hours.
- Except: 30 functions hit the 300 second timeout.

Limitations

- Limitations inherited from the SATURN infrastructure:
  - SATURN sometimes loses track of the effect of function calls on local data.
  - Writing outside array bounds can create hidden information flows.
- The conditions of information flows can become large.
- Some programming language constructs are inherently hard to analyze:
  - Arrays.
  - Heap shape.

Exploding Conditions

- Exact information flow conditions can become large during the analysis.
- Pathological cases where the exact condition is exponentially larger than the program.
- Instead of storing the exact condition E, the analysis tool stores two conditions (A, B), satisfying

$$A \implies E \quad \wedge \quad E \implies B$$

Instead of using E, the analysis uses either A or B (whichever is conservative).

Arrays Considered Harmful

- Consider the following code snippet:

```
a[i] = secret;
public = a[j];
```

- Array indexing makes information flow dependent on integer equations.
- **Approximation:** Treat entire array as a single variable, but switch off clobbering.

Future Plans: Short Term

- Develop a simple annotation language.
  - Must cover common categories of data (e.g., user input, sensitive data, public output, declassifying function).
  - Pre-annotate the interface to the C standard library, allowing more to be done with fewer evaluator annotations.
- Implement a robust information flow analysis tool.
  - Add conditions sparingly, to make the analysis more precise.
- Implement information flow visualization.
  - Make it easy to see whether there are missing annotations, such as declassifying function.

Future Plans: Longer Term

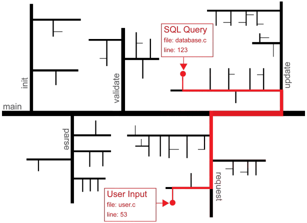
- Information flow specifications.
    - Derive C program specifications from higher-level security policies (Lobster).
- Extend the automatic analysis.
    - Heap shape (SmallFoot).
    - Information flow through C modules.
    - Quantitative information flow.
- Assurance.
    - CEGAR model checker connection (blast).
    - Theorem prover connection (ACL2).

galois

Summary

- A good support tool is like an MRI scanner: the user drives; and the automation keeps track of the details.
- Information flow static analysis tools can scale up to find real security bugs in widely used software.
- The design is not yet set in stone—feedback welcome!

joe@galois.com



galois