Visualizing Information Flow through C Programs

Joe Hurd, Aaron Tomb and David Burke

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

Systems Software Verification Workshop
7 October 2010

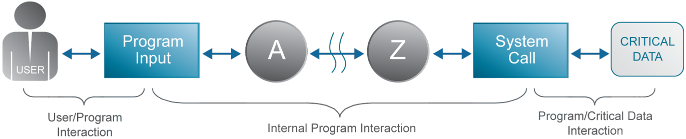
galois

Talk Plan

galois

Software Security Evaluation

- Evaluating the security of a program generally focuses 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?

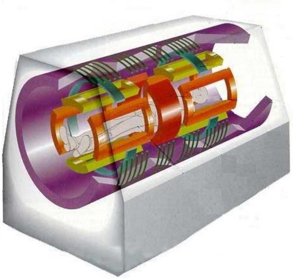


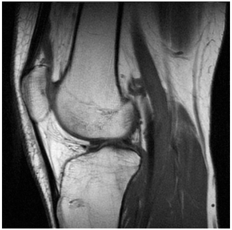
- Answering this requires an understanding of how information flows through the program.

galois

Information Flow Diagnostic Tool

- Tight time constraints mean that evaluators often cannot look at every line of the codebase.
- **Project Goal:** Develop an interactive diagnostic tool that allows an evaluator to scan for anomalies in the program information flows.





galois

Information Flow and Security Properties

- Many program security properties can be expressed in terms of potential information flow between program variables.
- **Confidentiality:** $\surd$
  - There are no information flows from secret variables to public variables.
- **Integrity:** $\surd$
  - There are no information flows from tainted variables to critical variables.
- **Availability:** $\times$
  - No way to specify that a flow must happen.

Information Flow and Confidentiality

- **Use Case:** Ensure Bell-La Padula properties hold for a cross domain application.
- **Annotate:** Program variables with their sensitivity level.
- **Check:** There are no flows from higher sensitivity to lower sensitivity variables.
- **Example:**

Code (Confidentiality Bug)

```
void f() {
  int k = get_secret_key();

  publish_to_internet(k);
}
```

galois

Information Flow and Integrity

- **Use Case:** Defend against SQL injection attacks.
- **Annotate:** Tainted data variables; critical data variables; and validation functions.
- **Check:** All flows from tainted variables to critical variables go through validation functions.
- **Example:**

Code (Integrity Check Succeeds)

```
void f() {
  int data = get_user_input();

  data = validate_input(data);

  query_sql_database(data);
}
```

User Centered Design

- *"All tools are user interfaces"* – Clark Dodsworth
- **Evaluator/Tool 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.
- **Tool Requirements:**
  1. Scalable analysis of program information flow.
  2. Intuitive visualization of information flow in terms of source code.

galois

Analysis Evidence

- **Evidence of Security Bugs:** Insecure information flows are presented as a sequence of assignments on the control flow.
  - **False Positives:** The evaluator uses the tool to browse the insecure information flows, and adds annotations to eliminate false positives.
- **Evidence of Assurance:** The analysis computes an conservative over-approximation of information flow on a subset of the programming language.
  - **False Negatives:** The tool will emit a warning message when the analysis detects that the program is outside of the conservative subset, allowing the evaluator to assess the residual risk.

galois

Static Analysis

- Static analysis is a program verification technique that is complementary to testing.
  - Testing works by executing the program and checking its run-time behavior.
  - Static analysis works by examining the text of the program.
- Driven by new techniques, static analysis tools have recently made great improvements in scope.
  - **Example 1:** Modern type systems can check data integrity properties of programs at compile time.
  - **Example 2:** Abstract intepretation techniques can find memory problems such as buffer overflows or dangling pointers.
  - **Example 3:** The TERMINATOR tool developed by Microsoft Research can find infinite loops in Windows device drivers that would cause the OS to hang.

Information Flow Static Analysis: Requirements

- **Evidence:** Generating evidence of assurance relies on the information flow static analysis being sound:
  1. Define a sound static analysis on a simple flow language.
  2. Implement a conservative translator from the target programming language to the simple flow language.
- **Scalability:** To help the static analysis scale up to realistically sized codebases, we design it to be compositional.
  - Preserve function calls in the flow language.
- **Program Understanding:** The analysis result must help an evaluator understand how information flows through the program source code.
  - Link each step in the analysis to the program source code.

galois

Information Flow Static Analysis of C Code

- The following front end processing is performed to translate C code to the flow language:
  1. **Preprocessing:** The C preprocessor.
  2. **Parsing:** The Haskell `Language.C` package.
  3. **Simplification:** Normalizing expressions (like CIL).
  4. **Variable Classification:** Special handling for address-taken locals and dynamically allocated memory.
  5. **Pointer Analysis:** Anderson's algorithm replaces each indirect reference with a set of direct references.
- **Key Property:** The front end processing is conservative.
  - Every information flow in the C code is translated to an information flow in the flow language.
  - **Assumption:** the C code is memory safe.

galois

The Flow Language

- **Variables**
  - Global variables.
  - Local variables of a function.
- **Statements**
  - Simple variable assignment $v_1 \leftarrow v_2$.
  - Function call $v \leftarrow f(v_1, \ldots, v_n)$.
- **Functions**
  - Special local variables representing input arguments `$arg1,...,$argN` and return value `$ret`.
  - A function contains a set of statements (flow insensitive).
- **Programs**
  - A set of functions, including a distinguished main function where execution begins.

galois

The Flow Language as a C Subset

Code (Example Program)

```
/* High global variables */
int high_in; int high_out;

/* Low global variables */
int low_in; int low_out;

int f(int x) { return x + 1; }

int main() {
  high_in = 42;
  low_in = 35;

  high_out = f(high_in);
  low_out = f(low_in);

  return 0;
}
```

Step 1/4: Compute Function Transformers

- For a function f, the transformer T_f is the subset of global variables and argument variables that can flow into the return value.
- Transformers can be efficiently computed by a bottom-up traversal of the call graph (using Bourdoncle's algorithm).

Analysis (Example Function Transformers)

$$\begin{aligned} T_{\texttt{f}} &= \{\texttt{\$arg1}\} \\ T_{\texttt{main}} &= \emptyset \end{aligned}$$

galois

Step 2/4: Compute Function Contexts

- For a function f, the context C_f is a mapping from each argument of f to the subset of global variables that can flow into the argument.
- Contexts can be efficiently computed by a top-down traversal of the call graph, starting with main (using Bourdoncle's algorithm and the transformers).

Analysis (Example Function Contexts)

$$\begin{aligned} C_{\mathtt{f}} &= \mathtt{\$arg1} \mapsto \{\mathtt{low\_in},\ \mathtt{high\_in}\} \\ C_{\mathtt{main}} &= \emptyset \end{aligned}$$

galois

Step 3/4: Compute Function Information Flow Graphs

- For a function f, the information flow graph G_f is a directed graph between global variables, where an edge $\texttt{x} \to \texttt{y}$ indicates that f enables a possible information flow from x to y.
- The function information flow graphs can be efficiently computed from the transformers and contexts.
- **Key Property:** The information flow analysis is context sensitive.

Analysis (Example Function Information Flow Graphs)

$$
\begin{aligned}
G_{\texttt{f}} &= \emptyset \\
G_{\texttt{main}} &= \{\texttt{low\_in} \to \texttt{low\_out}, \\
&\quad\ \texttt{high\_in} \to \texttt{high\_out}\}
\end{aligned}
$$

galois

Step 4/4: Compute Program Information Flow Graph

- The program information flow graph G is a directed graph between global variables, where an edge $\mathtt{x} \rightarrow \mathtt{y}$ indicates that the program enables a possible information flow from x to y.
- The program information flow graph is the union of all the function information flow graphs G_f where f is reachable from the main function.
- **Key Property:** The information flow analysis is sound.

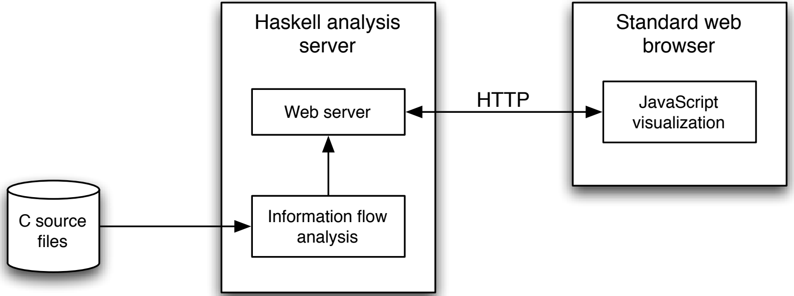
Analysis (Example Program Information Flow Graph)

$$
\begin{aligned}
G \quad = \quad & \{\mathtt{low\_in} \rightarrow \mathtt{low\_out}, \\
& \ \mathtt{high\_in} \rightarrow \mathtt{high\_out}\}
\end{aligned}
$$

galois

Cift Architecture

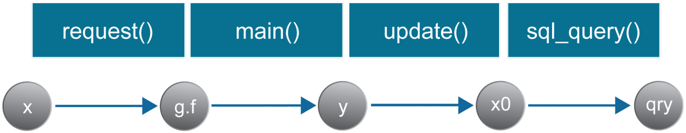
The C Information Flow Tool (Cift) allows evaluators to examine information flows in C code using a standard web browser.



The architecture is designed to support multiple simultaneous users browsing code and sharing annotations.

Visualizing Information Flow

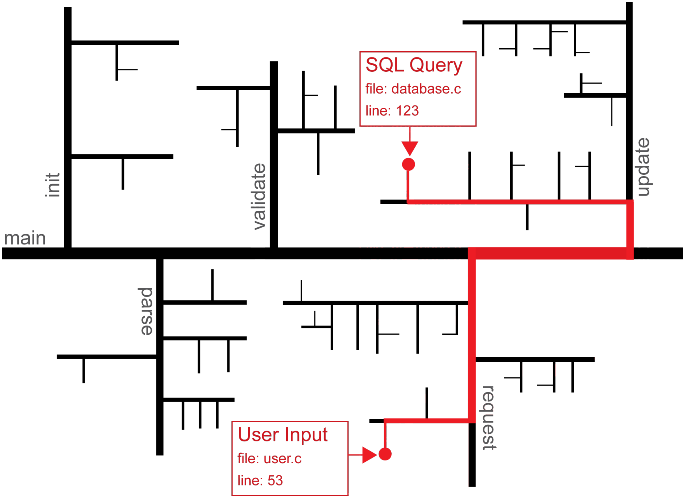
- A program information flow consists of many assignments distributed across the codebase:



- 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.

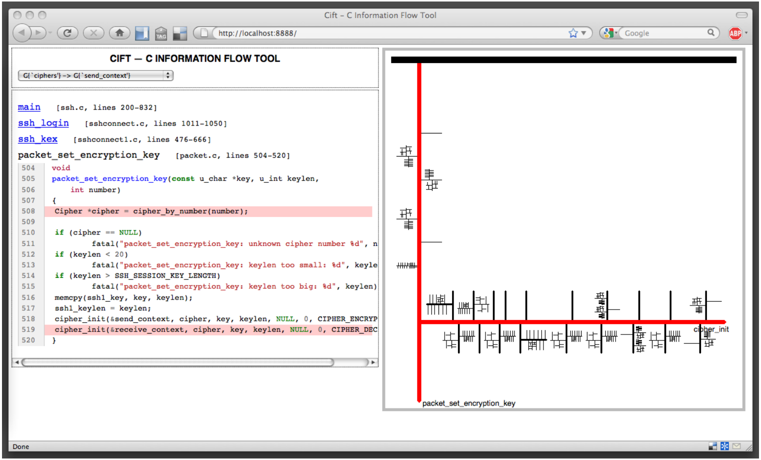
galois

Right-Angle Fractal Call Trees



galois

Demo



Cift – C Information Flow Tool
http://localhost:8888/
Google
CIFT — C INFORMATION FLOW TOOL
G(`ciphers') -> G(`send_context')
main [ssh.c, lines 200-832]
ssh_login [sshconnect.c, lines 1011-1050]
ssh_kex [sshconnect1.c, lines 476-666]
packet_set_encryption_key [packet.c, lines 504-520]
void
packet_set_encryption_key(const u_char *key, u_int keylen,
int number)
{
Cipher *cipher = cipher_by_number(number);
if (cipher == NULL)
fatal("packet_set_encryption_key: unknown cipher number %d", n
if (keylen < 20)
fatal("packet_set_encryption_key: keylen too small: %d", keyle
if (keylen > SSH_SESSION_KEY_LENGTH)
fatal("packet_set_encryption_key: keylen too big: %d", keylen)
memcpy(ssh1_key, key, keylen);
ssh1_keylen = keylen;
cipher_init(&send_context, cipher, key, keylen, NULL, 0, CIPHER_ENCRYP
cipher_init(&receive_context, cipher, key, keylen, NULL, 0, CIPHER_DEC
}
cipher_init
packet_set_encryption_key
Done

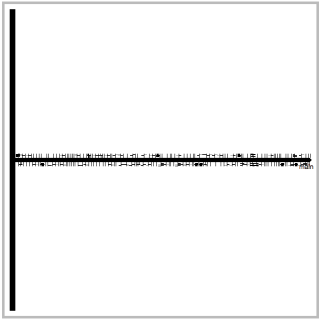
Restricting to Variables of Interest

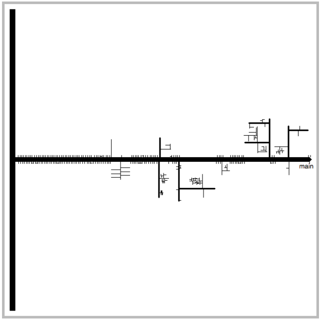
- **Problem:** A typical large program contains many variables V in the program information flow graph: information overload.
- **Solution:** Allow the user to specify a subset $X \subseteq V$ of interesting variables.
- Remove the uninteresting global variables $V - X$ from the program information flow graph one by one.
  - When removing a variable v, an extra edge $x \to y$ must be added between every pair of variables x, y satisfying $x \to v$ and $v \to y$.
  - This amounts to computing the transitive closure of the program information flow graph on demand.
- A first step towards information flow annotations.

galois

Emphasizing Call Tree Paths of Interest

Problem: Functions with too many function calls result in an uninformative hairy spike (left graphic).





Solution: Emphasize function calls contributing to information flows between variables of interest (right graphic).

galois

Open Source Benchmarks

- All experiments were carried out on a MacBook Pro 2.2Ghz Core 2 Duo with 4Gb of RAM, using GHC 6.12.1.
- Analyzing the 67 KLoC C implementation of OpenSSH takes 1:53s of CPU time and consumes 1.6Gb of RAM.
- Analyzing the 94 KLoC C implementation of the SpiderMonkey JavaScript interpreter takes 6:49s of CPU time and consumes 1.3Gb of RAM.

galois

Cift Development Plan

Milestones:

- $\surd$ Develop an automatic information flow analysis that scales up to realistic C codebases.
- $\surd$ Develop a visualization technique for program information flow that is grounded in the source code.
- $\surd$ Implement a research prototype tool to examine information flows in C programs.
- $\rightarrow$ Develop an annotation language for information flow properties of C functions and variables.
- $\rightarrow$ Allow users to edit annotations through the browser interface and see the resulting effects on the analysis.

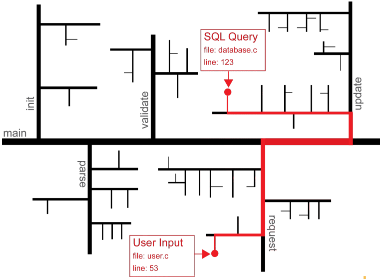
Future Plans

- Extend the scope of the information flow analysis.
  - Supporting array sensitivity to distinguish the elements of an array or cells in a memory block.
  - Adding flow sensitivity and a clobber analysis to detect failures to sanitize confidential data after use.
  - Target LLVM to extend the analysis to C++/Ada/etc.
- Support higher-level information flow specifications.
  - Derive program specifications from higher-level security policies.
  - Track information flow across module and language barriers.

Summary

- **This Talk:** We have presented a research prototype static analysis tool that an evaluator can use to visualize how information flows through C programs.
- **Feedback Welcome:** Please let us know what features you'd like to see in a program understanding tool.

joe@galois.com



galois