Hybrid Information Flow Analysis for Real-World C Code

joint work with Julien Signoles, submitted to POST 2017

Inria, November 14, 2016 | Gergő Barany gergo.barany@cea.fr
Information flow analysis

- pieces of data tagged with labels
  - public/secret
  - provenance (Internet domain, software component, . . .)
- analysis propagates labels to all affected data/computations

Flow policies define how information may flow

Examples:
- personal data may not flow to `send(1)` syscall
- cryptographic keys may not affect branch conditions
- packet routing may only depend on packet header, not payload
Information flow lattice
Labels form finite lattice $\langle S, \sqcup, \sqsubseteq, \bot \rangle$

- example: $S = \{L, H\}$ where $L$ (public) $\sqsubseteq H$ (private)
- example: software components $S = \mathcal{P}(\{C_1, \ldots, C_n\})$

Non-interference property

- ‘secret inputs do not affect public outputs’
- enforced by our analysis (for user-defined labels and policy)
Hybrid analysis with Frama-C

Analysis implemented in Frama-C

- Source-based analysis and transformation framework for C99
- Provides annotation language ACSL

Analysis implemented as hybrid (static/dynamic) analysis

- Instrument code with information flow tracking
- Instrumentation depends on previous static analysis (Frama-C’s Value)

Transformed code:
- can be executed (dynamic analysis)
- can be analyzed statically
Analysis example

- program transformation, annotations to express flow policy
- a label variable for each variable $x$, label updates

extern unsigned int /*@ private */ secret;
extern unsigned int /*@ public */ public;
char secret_status = 1, public_status = 0;
int main(void) {
    int result;
    result = public + secret;
    result_status = public_status | secret_status;
   /*@ assert security_status(result) == public; */
   /*@ assert result_status == 0; */
    return result;
}
Challenges for dynamic analysis for C

Our hybrid analysis supports most of C

- pointers to scalars, structured control flow (earlier work)
- arrays, pointer arithmetic
- struct
- semi-structured control flow: break, continue
- many forms of goto (manual or inserted by front-end)
- interprocedural flow, (some) indirect function calls
- dynamic memory allocation

Unsolved problems: pointer type casts, union types

Formalization of parts of the theory in Isabelle/HOL: in progress
Structured branches
Make control dependences explicit using program counter labels

```c
if_pc = pc | cond;
if (cond) {
    x = a;
    x = a | if_pc;
    y |= if_pc;
} else {
    y = b;
    y = b | if_pc;
    x |= if_pc;
}

while_pc = pc | cond;
while (cond) {
    x = a;
    x = a | while_pc;
}
```

```c
while_pc = pc | cond;
while (cond) {
    x = a;
    x = a | while_pc;
}
```

```
x |= while_pc;
```
**Pointer-based flow**

For pointer $p$, need label variables $p$ and $p_{\text{target}}$

**Invariant:** $p \mapsto v \iff p_{\text{target}} \mapsto v$

* $p = z$; /* assume $p \mapsto \{x, y\}$ */
* $p_{\text{target}} = z$;
* $x \vdash p$; /* propagate $p$ to all possible targets */
* $y \vdash p$;

Possible pointer targets found by static analysis

In general, need $n + 1$ label variables for pointer type $T^{(n)}$: $p$, $p_{\text{target}}_1$, $\ldots$, $p_{\text{target}}_n$
Problem

\[
\begin{align*}
\text{arr[]} &= \{ 0, 0, \ldots, 0 \}; \\
\text{arr[secret]} &= 1; \\
y &= \text{arr[0]};
\end{align*}
\]

Have \( y = 1 \iff \text{secret} = 0 \), so 1 bit leaked from \text{secret} to \( y \)

Solution

Summary label captures all flows into the array monotonically

\[
\begin{align*}
\text{arr[secret]} &= 1; \\
\text{arr\_summary} &= \text{secret}; \quad \text{/* weak update */} \\
y &= \text{arr[0]}; \\
y &= \text{arr\_summary}; \quad \text{/* field-insensitive read */}
\end{align*}
\]
Interaction of arrays and pointers

New invariant for arrays of pointers

If \( p \rightarrow^n \text{arr}[i] \), we need:

- \( \overline{p}_{\text{summary}} \rightarrow^n \text{arr}_{\text{summary}} \)
- \( \overline{p}_n \rightarrow^n \text{arr}[i] \)

Two status pointers per dereference level

For int *b[10]:

```c
char b_status;        /* array summary */
char b_status_d0[10]; /* statuses of array elems */
char *b_status_d1_summary[10]; /* pointers to summaries */
char *b_status_d1[10]; /* ptrs to exact target statuses */
```

Semi-structured control flow

\[ \text{loop}_pc = \text{cond} \mid \text{pc}; \]
while (cond) {
    \[ x = x + 1; \]
    \[ x = x \mid \text{loop}_pc; \]
    \[ \text{if}_pc = \text{secret} \mid \text{loop}_pc; \]
    \[ \text{loop}_pc \mid= \text{if}_pc; \]
    if (secret) break;
    \[ y = y + 1; \]
    \[ y = y \mid \text{loop}_pc; \]
}
\[ x \mid= \text{loop}_pc; \]
\[ y \mid= \text{loop}_pc; \]

Loop is control dependent on the if that controls the break
goto statement considered difficult (1/2)

goto

- may be written by humans:
  - if (error) goto end;
- may be introduced by Frama-C frontend for logical operations, early return, continue in for loops:
  - if (a && b) { c; } else { d; }
  becomes:
  if (a) {
    if (b) { c; }
    else goto _LAND;
  } else { _LAND: d; }

Control-dependent side effects must be treated correctly
x = 1;
if_pc = secret | pc;
if (secret) goto end;
x = 0;
end:

x |= pc;
/* x == 0 <=> secret == 0 */
return x;

Supported cases

- forward jump out of a block (like generalized break)
- jump within a branching statement (for logical ||, &&)
Transform function parameters and return value (and every call)

```c
char add_return;
float add(float x, float y) {
  float add(char local_pc, float x, char x, float y, char y) {
    float sum;
    char sum;
    sum = x + y;
    sum = local_pc \| y \| x;
    add_return = sum;
    return sum;
  }
  return sum;
}

Calls through function pointers allowed if Value can resolve them
Cannot transform external (library) functions

Require annotations in Frama-C’s ACSL annotation language:

/*@ assigns \result \from x, y; */
float add(float x, float y);

Appropriate label updates are generated at the call site:

sum = add(x, y);

*sum = x | y | pc;

Not always possible: Cannot handle pointers in assigns clauses (cannot ensure invariants)
To preserve array-pointer invariants, dynamically allocate labels for dynamically allocated data

```c
p = malloc(...);
p_d1 = calloc(...);
p_d1_summary = &dynalloc_site_1_summary;
if (...) {
    *p = 1;
    *p_d1 = 0 | if_pc;
} else {
    dynalloc_site_1_summary |= if_pc;
}
```

**Problem:** summary labels must have names
Each allocation site has a shared summary, imprecise.
TODO: context-sensitive allocation summaries
Reduce instrumentation code to variables involved in flow policy:

- collect variables $x$ in flow policy annotations like
  ```
 /*@ assert security_status(x) == public; */
  ```
- propagate backward, left-to-right in assignments
  - in assignment $x = \text{exp}$, monitor all vars of $\text{exp}$ if $x$ monitored
- result: overapproximation of variables on which policy annotations depend
  - need not monitor others
Evaluation on **LibTomCrypt** crypto library

**Flow policy:** all branch conditions have public labels

- insert `/*@ assert security_status(condition) == public; */` before each branch
- avoid timing attacks based on control flow based on secret key

**Symmetric cryptosystems (AES etc.):**

- static analysis: flow policy verified
- dynamic analysis: $\sim 2 \times$ slowdown, $+60\%$ memory used

**Elliptic curve cryptography:**

- static analysis: proved known vulnerability
- dynamic analysis: $6.5 \times$ slowdown, $+10\%$ memory used
- even “timing attack resistant” variant is vulnerable
Summary

- hybrid information flow analysis handling almost all of C implementation in Frama-C
- practical evaluation: usable on real-world crypto software

Thank you for your attention!

This work was supported by the French National Research Agency (ANR), project AnaStaSec, ANR-14-CE28-0014.