DE LA RECHERCHE À L'INDUSTRIE

Hybrid Information Flow Analysis for Programs with Arrays

VPT 2016, April 2, 2016 | Gergö Barany gergo.barany@cea.fr

Information flow analysis 1/2

Information flow analysis

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

Information flow analysis 1/2

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 analysis 2/2

Information flow lattice

Labels form finite lattice $\langle S, \sqcup, \sqsubseteq, \perp \rangle$

- example: $S = \{L, H\}$ where L (public) $\sqsubset H$ (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)

Contributions

Contributions of this work

- extended hybrid (static/dynamic) analysis for C to handle arrays and pointer arithmetic
- machine-checked proof of non-interference property for underlying semantics (Isabelle/HOL)

Hybrid analysis: basics (earlier work)

Dynamic analysis (program transformation): introduce label variable $\underline{\mathtt{x}}$ for each variable `x`, assignment to $\underline{\mathtt{x}}$ for assignment to `x`

Direct information flow

$$\begin{array}{l} \quad \mathtt{z = x + y;} \\ \rightarrow \underline{\mathtt{z}} \mathtt{\ =\ } \underline{\mathtt{x}} \mathtt{\ |\ } \underline{\mathtt{y}}\mathtt{;} \end{array}$$

/* combination operator | (bitwise or) */

Hybrid analysis: basics (earlier work)

Dynamic analysis (program transformation): introduce label variable $\underline{\mathtt{x}}$ for each variable $\mathtt{x}$, assignment to $\underline{\mathtt{x}}$ for assignment to $\mathtt{x}$

Direct information flow

$\quad \mathtt{z = x + y;}$

$\rightarrow \underline{\mathtt{z}} = \underline{\mathtt{x}} \mid \underline{\mathtt{y}};$ /* combination operator | (bitwise or) */

Pointer-based flow

$\quad \mathtt{*p = z;}$ /* assume $\mathtt{p} \mapsto \{\mathtt{x}, \mathtt{y}\}$ */

$\rightarrow \mathtt{*}\underline{\mathtt{p\_d1}} = \underline{\mathtt{z}};$ /* maintain invariant $\mathtt{p} \mapsto v \Leftrightarrow \underline{\mathtt{p\_d1}} \mapsto \underline{v}$ */

Hybrid analysis: basics (earlier work)

Dynamic analysis (program transformation): introduce label variable $\underline{x}$ for each variable x, assignment to $\underline{x}$ for assignment to x

Direct information flow

$\quad$ `z = x + y;`

$\rightarrow \underline{\texttt{z}} = \underline{\texttt{x}} \mid \underline{\texttt{y}};$ $\quad$ /* combination operator | (bitwise or) */

Pointer-based flow

$\quad$ `*p = z;` $\quad$ /* assume $p \mapsto \{x, y\}$ */

$\rightarrow *\underline{\texttt{p\_d1}} = \underline{\texttt{z}};$ $\quad$ /* maintain invariant $p \mapsto v \Leftrightarrow \underline{p\_d1} \mapsto \underline{v}$ */

$\rightarrow \underline{\texttt{x}} = \underline{\texttt{x}} \mid \underline{\texttt{p}};$ $\quad$ /* propagate $\underline{p}$ to all possible targets */

$\rightarrow \underline{\texttt{y}} = \underline{\texttt{y}} \mid \underline{\texttt{p}};$

Possible pointer targets found by static analysis

Information flow analysis for arrays 1/2

Naïve approach

Array elements independent of each other

```
  arr[1] = x;
→ arr[1] = x;
  y = arr[0];
→ y = arr[0];
```

Information flow analysis for arrays 1/2

Naïve approach

Array elements independent of each other

```
  arr[1] = x;
→ arr[1] = x;
  y = arr[0];
→ y = arr[0];
```

Problem

Array elements not independent of index

```
arr[] = { 0, 0, ..., 0 };
arr[secret] = 1;
y = arr[0];
```

Have $y = 1 \Leftrightarrow secret = 0$, so 1 bit leaked from secret to y

Information flow analysis for arrays 2/2

Problem

```
arr[secret] = 1;
y = arr[0];
```

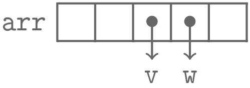
Solution

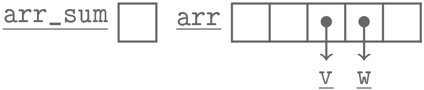
Use extra summary label for arrays

```
  arr[secret] = 1;
→ arr_summary |= secret;    /* weak update */
  y = arr[0];
→ y = arr_summary;          /* field-insensitive read */
```

Summary captures all flows into the array, increases monotonically

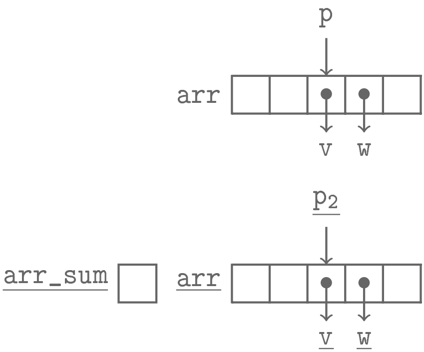
Interaction of arrays and pointers 1/2





Invariants

- $\mathtt{p} \mapsto^{n} \mathtt{x} \Leftrightarrow \underline{\mathtt{p}_{\mathtt{n}}} \mapsto^{n} \underline{\mathtt{x}}$



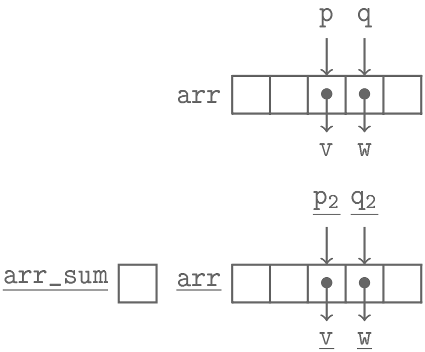
$\mathtt{p = \&arr[2];}$

$\rightarrow$ $\underline{\mathtt{p}_2}\ \mathtt{=\ \&}\underline{\mathtt{arr}}\mathtt{[2];}$

Interaction of arrays and pointers 1/2

Invariants

- $p \mapsto^{n} x \Leftrightarrow \underline{p_n} \mapsto^{n} \underline{x}$
- pointer arithmetic on p is reflected on $\underline{p}$



$p = \&arr[2];$

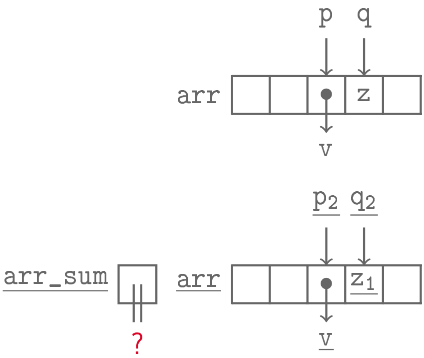
$\underline{p_2} = \&\underline{arr}[2];$

$q = p + 1;$

$\rightarrow \underline{q_2} = \underline{p_2} + 1;$

Invariants

- $\mathtt{p} \mapsto^{n} \mathtt{x} \Leftrightarrow \underline{\mathtt{p_n}} \mapsto^{n} \underline{\mathtt{x}}$
- pointer arithmetic on $\mathtt{p}$ is reflected on $\underline{\mathtt{p}}$



$\mathtt{p = \&arr[2];}$

$\underline{\mathtt{p_2}}\ \mathtt{=}\ \mathtt{\&}\underline{\mathtt{arr}}\mathtt{[2];}$

$\mathtt{q = p + 1;}$

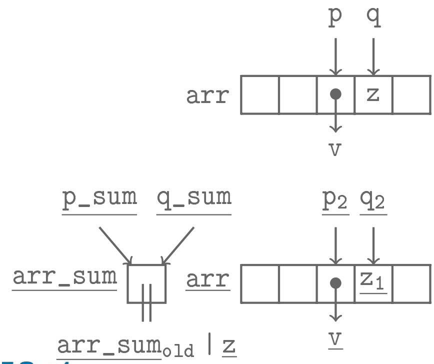
$\underline{\mathtt{q_2}}\ \mathtt{=}\ \underline{\mathtt{p_2}}\ \mathtt{+ 1;}$

$\mathtt{*q = z;}$

$\mathtt{*}\underline{\mathtt{q_2}}\ \mathtt{=}\ \underline{\mathtt{z_1}}\mathtt{;}$

Invariants

- $p \mapsto^{n} x \Leftrightarrow \underline{p_n} \mapsto^{n} \underline{x}$
- pointer arithmetic on p is reflected on $\underline{p}$
- need both exact and summary pointers



p = &arr[2];
$\underline{p_2}$ = &$\underline{arr}$[2];
→ $\underline{p\_sum}$ = &$\underline{arr\_sum}$;
q = p + 1;
$\underline{q_2}$ = $\underline{p_2}$ + 1;
→ $\underline{q\_sum}$ = $\underline{p\_sum}$;
*q = z;
*$\underline{q_2}$ = $\underline{z_1}$;
→ *$\underline{q\_sum}$ |= $\underline{z}$;

Interaction of arrays and pointers 2/2

Main new invariant

if p $\mapsto^n$ arr[i], we need:

- $\underline{\texttt{p\_summary}_\texttt{n}} \mapsto^n \underline{\texttt{arr\_summary}}$
- $\underline{\texttt{p}_\texttt{n}} \mapsto^n \underline{\texttt{arr}}\texttt{[i]}$

Two status pointers per dereference level

for `int *b[10]`:

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

Soundness

Monitor semantics

- extend semantic judgements: $E \vdash \mathit{prog}, M \Rightarrow M'$
  with label memory: $E, S_P, pc \vdash \mathit{prog}, M, \Gamma \Rightarrow M', \Gamma'$
- $M(b)$: value of memory block b, $\Gamma(b)$: label of b
- semantic rules extended to update Γ using alias analysis S_P

Soundness

Monitor semantics

- extend semantic judgements: $E \vdash prog, M \Rightarrow M'$
  with label memory: $E, S_P, pc \vdash prog, M, \Gamma \Rightarrow M', \Gamma'$
- $M(b)$: value of memory block b, $\Gamma(b)$: label of b
- semantic rules extended to update Γ using alias analysis S_P

Soundness proof

- showed that our rules for Γ have non-interference property
  - change b with $\Gamma(b) \not\sqsubseteq s \Rightarrow \Gamma'(c) \not\sqsubseteq s$ for changed outputs c
- full development: 1900 lines of Isabelle/HOL

Soundness

Monitor semantics

- extend semantic judgements: $E \vdash prog, M \Rightarrow M'$
  with label memory: $E, S_P, pc \vdash prog, M, \Gamma \Rightarrow M', \Gamma'$
- $M(b)$: value of memory block b, $\Gamma(b)$: label of b
- semantic rules extended to update Γ using alias analysis S_P

Soundness proof

- showed that our rules for Γ have non-interference property
  - change b with $\Gamma(b) \not\sqsubseteq s \Rightarrow \Gamma'(c) \not\sqsubseteq s$ for changed outputs c
- full development: 1900 lines of Isabelle/HOL

Future work

show that program transformation correctly computes Γ

Implementation 1/2

Prototype implementation in Frama-C

- program transformation, annotations to express flow policy

```
extern unsigned int /*@ private */ secret;
extern unsigned int /*@ public */ public;

int main(void) {
    int result;
    result = public + secret;

    /*@ assert security_status(result) == private; */

    return result;
}
```

Implementation 1/2

Prototype implementation in Frama-C

- program transformation, annotations to express flow policy

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

Implementation 2/2

Status

- uses Frama-C's points-to analysis (Value)
- arrays, pointers, structures, control flow, function calls
  - TODO: semi-structured control flow (`continue`, early `return`)
- annotations checked dynamically or statically (Value, WP)
- real-world case studies: coming soon

Summary

- hybrid information flow analysis handling pointers, arrays, pointer arithmetic
- monitor semantics proved correct, proof of transformation WIP
- prototype implementation in Frama-C

Summary

- hybrid information flow analysis handling pointers, arrays, pointer arithmetic
- monitor semantics proved correct, proof of transformation WIP
- prototype implementation in Frama-C

Thank you for your attention!

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