Proving Information Flow Noninterference by Reusing a Machine-Checked Correctness Proof for Slicing

Daniel Wasserrab and Denis Lohner

```
theorem nonInterferenceSecurity:
  assumes "cf_1 \approx_{L} cf_2" and "(\text{-High-}) \not\in [\text{HRB-slice (CFG-node (\text{-Low-}))}]_{\text{CFG}}" and "valid-edge a"
  and "sourcenode a = (\text{-High-})" and "targetnode a = n" and "kind a = (\lambda s. \text{True})\nu" and "n \triangleq c"
  and "final c'" and "\langle c, [cf_1] \rangle \Rightarrow \langle c', s_1 \rangle" and "\langle c, [cf_2] \rangle \Rightarrow \langle c', s_2 \rangle"
  shows "s_1 \approx_{L} s_2"

proof -
  from High-target-Entry-edge obtain ax where "valid-edge ax" and "sourcenode ax = (\text{-Entry-})"
  and "targetnode ax = (\text{-High-})" and "kind ax = (\lambda s. \text{True})\nu" by blast
  from \'n \triangleq c \', \langle c, [cf_1] \rangle \Rightarrow \langle c', s_2 \rangle\' obtain n_1 as_1 cf_1 where "n \triangleq as_1 \rightarrow \nu" n_1" and "n_1 \triangleq c'" and "preds (kinds as_1) [[cf_1, \text{undefined}]]"
  and "transfers (kinds as_1) [[cf_1, \text{undefined}]] = cf_1" and "map fst cf_1 = s_1" by(fastsimp dest:fundamental-property)
  from \'n \triangleq as_1 \rightarrow \nu" n_1\', \'valid-edge a\' \,'sourcenode a = (\text{-High-})\' \,'targetnode a = n\' \,'kind a = (\lambda s. \text{True})\nu"
  have "(\text{-High-}) \rightarrow a#as_1 \rightarrow \nu" n_1\' by(fastsimp intro:Cons-path simp:wp-def valid-path-def)
  from "final c'\', \'n_1 \triangleq c\' obtain a_1 where "valid-edge a_1" and "sourcenode a_1 = n_1" and "targetnode a_1 = (\text{-Low-})" and "kind a_1 = \phi\id"
  by(fastsimp dest:final-edge-low)
```
Context of this work

[Hammer, Snelting: ISIS '09]:

Security analysis for information flow control

- Bases on Program Slicing
- Handles full Java byte code
- Aim: Precision
  - flow-
  - context- and
  - object-sensitive
Context of this work

[Hammer, Snelting: ISIS '09]:

Security analysis for information flow control

- Bases on Program Slicing
- Handles full Java byte code
- Aim: Precision
  - flow-
  - context- and
  - object-sensitive

But only really trustworthy if verified!
"Quis custodiet"

- Verification of slicing based IFC algorithms
- machine-checked in proof assistant Isabelle/HOL
- new level of trustworthiness
"Quis custodiet"

- Verification of slicing based IFC algorithms
- machine-checked in proof assistant Isabelle/HOL
- new level of trustworthiness

Requires correctness proof for slicing
Roadmap

Backward reasoning:
Roadmap

Backward reasoning:

- Noninterference proof requires correctness proof for context-sensitive interprocedural slicing
Roadmap

Backward reasoning:

- Noninterference proof requires correctness proof for context-sensitive interprocedural slicing
- Slicing correctness proof requires formalization of context-sensitive slicing in Isabelle/HOL
Roadmap

Backward reasoning:

- Noninterference proof requires correctness proof for context-sensitive interprocedural slicing
- Slicing correctness proof requires formalization of context-sensitive slicing in Isabelle/HOL
- And what is this slicing anyway?
Roadmap

Forward talking:

- And what is this slicing anyway?

- Slicing correctness proof requires formalization of context-sensitive slicing in Isabelle/HOL

- Noninterference proof requires correctness proof for context-sensitive interprocedural slicing
Program Slicing

Backward Slicing: (short: Slicing)
Eliminates all statements that cannot influence certain program point

Example:

```c
int main() {
    if (l1 == 42) {
        swap(h1,h2);
    } else {
        swap(l1,l2);
    }
    print l1;
}

void swap(ref int x, ref int y){
    int temp = y;
    y = x;
    x = temp;
}
```

Program Slicing

Backward Slicing: (short: Slicing)
Eliminates all statements that cannot influence certain program point

Example:

```c
int main() {
    if (l1 == 42) {
        swap(l1, l2);
    } else {
        swap(l1, l2);
    }
    print l1;
}
```

```c
void swap(ref int x, ref int y) {
    int temp = y;
    y = x;
    x = temp;
}
```
Program Slicing

Backward Slicing: (short: Slicing)
Eliminates all statements that cannot influence certain program point

Example:

```c
int main() {
    if (l1 == 42) {
        void swap(ref int x, ref int y) {
            int temp = y;
            x = temp;
        }
        swap(l1, l2);
    } else {
        print l1;
    }
}
```
Control and Dependence Graph

\begin{verbatim}
E int main() {
    if (l1 == 42) {
        swap(h1,h2);
    } else {
        swap(l1,l2);
    }
    print l1;
}

void swap(ref int x, ref int y){
    int temp = y;
    y = x;
    x = temp;
}
\end{verbatim}
Control and Dependence Graph

![Diagram of a control and dependence graph with nodes labeled E, e, 1, i, ii, iii, 2, h1, h2, l1, l2, 3, 4, X, x, y, i, ii, iii. The diagram shows directed edges between the nodes.]
Control and Dependence Graph
Intraprocedural Approach: Reachability
Intraprocedural Approach: Reachability
Intraprocedural Approach: Reachability

Intraprocedural slicing proved correct in Isabelle/HOL
[Wasserrab et al.: PLAS '09]
Intraprocedural Approach: Reachability

Intraprocedural slicing proved correct in Isabelle/HOL
[Wasserrab et al.: PLAS '09]

But: not context-sensitive!
Slicing Algorithm of Horwitz, Reps, Binkley

- context-sensitive interprocedural algorithm
- 2 Phases:
  - intraprocedural edges + edges descending to callers
  - intraprocedural edges + edges ascending to callees
- requires intraprocedural summary edges
  "shortcuts" dependence path in procedure call
Slicing Algorithm of Horwitz, Reps, Binkley

- context-sensitive interprocedural algorithm
- 2 Phases:
  - intraprocedural edges + edges descending to callers
  - intraprocedural edges + edges ascending to callees
- requires intraprocedural summary edges
  "shortcuts" dependence path in procedure call

![Diagram of slicing algorithm](image)
Slicing Algorithm of Horwitz, Reps, Binkley

- context-sensitive interprocedural algorithm
- 2 Phases:
  - intraprocedural edges + edges descending to callers
  - intraprocedural edges + edges ascending to callees
- requires intraprocedural summary edges
  "shortcuts" dependence path in procedure call
Slicing Algorithm of Horwitz, Reps, Binkley

- context-sensitive interprocedural algorithm
- 2 Phases:
  - intraprocedural edges + edges descending to callers
  - intraprocedural edges + edges ascending to callees
- requires intraprocedural summary edges
  "shortcuts" dependence path in procedure call
Slicing Algorithm of Horwitz, Reps, Binkley

- context-sensitive interprocedural algorithm
- 2 Phases:
  - intraprocedural edges + edges descending to callers
  - intraprocedural edges + edges ascending to callees
- requires intraprocedural summary edges
  "shortcuts" dependence path in procedure call

![Diagram of slicing algorithm](image)
Slicing Algorithm of Horwitz, Reps, Binkley
Slicing Algorithm of Horwitz, Reps, Binkley
Slicing Algorithm of Horwitz, Reps, Binkley

Result: context-sensitive slice of program points
Slicing Algorithm of Horwitz, Reps, Binkley

Result: context-sensitive slice of program points

Although standard algorithm, never formally verified
The Framework for Slicing

Where do we start?

```c
int main() {
    if (l1 == 42) {
        swap(h1, h2);
    } else {
        swap(l1, l2);
    }
    printf("%d", l1);
}

void swap(ref int x, ref int y) {
    int temp = y;
    y = x;
    x = temp;
}
```
The Framework for Slicing

- graph-based slicing is language-independent
- hence: framework bases on abstract CFG
- split call-sites in call and return nodes
- edges carry semantic effect
  - state update \( \uparrow \)
  - predicate check \( \checkmark \)
  - call (argument passing) \( \leftarrow \)
  - return (restores state with return variables) \( \rightarrow \)
- Def and Use sets for every node
  - axiomatize abstract CFG
  - structural properties
  - well-formedness properties
The Framework for Slicing

- graph-based slicing is language-independent
- hence: framework bases on abstract CFG
- split call-sites in call and return nodes
- edges carry semantic effect
  - state update $\uparrow$
  - predicate check $\sqrt{\ }$
  - call (argument passing) $\leftarrow$
  - return (restores state with return variables) $\leftarrow$
- Def and Use sets for every node
- axiomatize abstract CFG
  - structural properties
  - well-formedness properties
The Framework for Slicing

- graph-based slicing is language-independent
- hence: framework bases on abstract CFG
- split call-sites in call and return nodes
- edges carry semantic effect
  - state update \( \uparrow \)
  - predicate check \( \checkmark \)
  - call (argument passing) \( \leftarrow \)
  - return (restores state with return variables) \( \rightarrow \)

- Def and Use sets for every node
- axiomatize abstract CFG
  - structural properties
  - well-formedness properties
The Framework for Slicing

- structural properties
- well-formedness properties
The Framework for Slicing

- structural properties
  - unique Entry and Exit node,
  - no multi-edges,
  - one return for each call edge,
  - calls to the same procedure reach same node,
  - ...

- well-formedness properties
The Framework for Slicing

- structural properties
  - unique Entry and Exit node,
  - no multi-edges,
  - one return for each call edge,
  - calls to the same procedure reach same node,
  - ...

- well-formedness properties
  - Entry and Exit empty Def and Use sets,
  - lengths of formal and actual parameters agree,
  - update changes only Def variables,
  - same values in Use variables \( \implies \)
    same semantic effect on Def variables,
  - ...

The Framework for Slicing

- structural properties
  - unique Entry and Exit node,
  - no multi-edges,
  - one return for each call edge,
  - calls to the same procedure reach same node,
  - ...

- well-formedness properties
  - Entry and Exit empty Def and Use sets,
  - lengths of formal and actual parameters agree,
  - update changes only Def variables,
  - same values in Use variables \( \implies \)
    same semantic effect on Def variables,
  - ...

Combine edges to **paths**

**Valid paths:**

- context-sensitive
- when leaving procedure, always return to "correct" call site
- only valid paths agree to program executions
The Framework for Slicing

System dependence graph

- Nodes
  - CFG nodes +
  - additional parameter nodes

- Edges
  - control and data dependence
  - call and return
  - param-in and -out
  - **summary edges** replace context-sensitive dependence path
The Framework for Slicing

System dependence graph

- Nodes
  - CFG nodes +
  - additional parameter nodes

- Edges
  - control and data dependence
  - call and return
  - param-in and -out
  - **summary edges** replace context-sensitive dependence path

HRB slice:

- reflexive-transitive closure of corresponding SDG edges for each phase
- sets combined according to algorithm
Correctness of Slicing

"If there is no dependence path, there is no influence!"

- Execution of sliced program yields **same observable effect** at slicing node as execution of original program
  - values in used variables equal
  - execution visits same control structures in same order
- In framework: program execution = traversing valid path in CFG
- Sliced Program: **Sliced CFG**
  - replace semantic effect of nodes not in the slice with no-ops
- Corollary from a more general weak simulation property
Correctness of Slicing

Weak simulation property:

- Regard original and sliced CFG as labelled transition systems
- Weak simulation between (node stack, state) tuples
  - node stack: current node + call history
- If move possible in original CFG, has to be possible in sliced CFG
- Silent moves: originate in nodes not in slice
Correctness of Slicing

Weak simulation property:
- Regard original and sliced CFG as labelled transition systems
- Weak simulation between (node stack, state) tuples
  - node stack: current node + call history
- If move possible in original CFG, has to be possible in sliced CFG
- Silent moves: originate in nodes not in slice

Weak simulation formalization in Isabelle:

\[
\forall n \in \text{set } ns. \text{valid-node } n \quad \forall n' \in \text{set } ns'. \text{valid-node } n' \\
\forall n \in \text{set } (tl \ ns). \text{return-node } n \quad |ns| = |s| \quad |ns'| = |s'| \\
s \neq [] \quad s' \neq [] \quad ns = nsx \cdot \text{tl } ns' \quad \text{get-proc } nx = \text{get-proc } (hd \ ns') \\
\forall n \in (tl \ ns'). \exists n'. \text{call-of-return-node } n \ n' \wedge n' \in [\text{HRB-slice } n_c]_{\text{CFG}} \\
nsx \neq [] \rightarrow (\exists nx'. \text{call-of-return-node } nx \ nx' \wedge nx' \notin [\text{HRB-slice } n_c]_{\text{CFG}}) \\
\text{obs } ns [\text{HRB-slice } n_c]_{\text{CFG}} = \text{obs } ns' [\text{HRB-slice } n_c]_{\text{CFG}} \\
\forall i < |ns'|. \text{snd } s[|nsx| + i] = \text{snd } s'[i] \\
\forall i < |ns'|. \forall V \in rv \ n_c (\text{CFG-node } (\text{nsx} \cdot \text{tl } ns')[|i|]). (\text{fst } s[|nsx| + i]) V = (\text{fst } s'[i]) V \\
\left( (ns,s),(ns',s') \right) \in WS \ n_c
\]
Correctness of Slicing

The diagram illustrates the correctness of slicing in information flow noninterference. Nodes represent statements or expressions, and arrows indicate the flow of information or the dependency between them. The values True and False are used to denote the flow of information, with True indicating a flow and False indicating no flow. The slicing process involves removing irrelevant parts of the program to ensure that sensitive information is not leaked to the wrong places.
Correctness of Slicing
Correctness of Slicing

![Diagram of Correctness of Slicing](image-url)
Correctness of Slicing

The diagram illustrates the flow of information and the correctness of slicing in a program. The nodes represent states or points in the program, and the edges show the flow of information. The labels on the edges indicate the flow direction and the conditions under which the information is propagated.

Key points:
- Nodes labeled with numbers represent states or points in the program.
- The labels on the edges indicate the flow direction and the conditions for information flow.
- The correctness of slicing is verified through the paths and the truth values at each state.

The diagram uses symbols such as arrows to denote the flow of information and logical operators to represent the conditions under which the information is propagated.
Correctness of Slicing
Correctness of Slicing

\[
\begin{align*}
E \quad &\quad \text{True} \checkmark \\
1 \quad &\quad \text{False} \checkmark \\
2 \quad &\quad \text{True} \checkmark \\
3 \quad &\quad \text{False} \checkmark \\
4 \quad &\quad \text{True} \checkmark \\
X \quad &\quad \text{False} \checkmark
\end{align*}
\]
Correctness of Slicing
Information Flow Control

"Can secret information flow to public outputs?"
Program noninterferent $\iff$ no such flows possible

Classical noninterference:

- Assigns security labels to variables (here: $H$ secret, $L$ public)
- Regards two complete program runs
- In initial states, values equal in all $L$ variables: low equality $\approx_L$
- If final states again low equal, program noninterferent
IFC with Slicing

```c
int main() {
    if (l1 == 42) {
        swap(h1, h2);
    } else {
        swap(l1, l2);
    }
    print l1;
}

global void swap(ref int x, ref int y) {
    int temp = y;
    y = x;
    x = temp;
}
```
int main() {
    if ($l_1$ == 42) {
        swap($h_1$, $h_2$);
    } else {
        swap($l_1$, $l_2$);
    }
    print $l_1$;
}

void swap(ref int $x$, ref int $y$) {
    int temp = $y$;
    $y$ = $x$;
    $x$ = temp;
}
IFC with Slicing

Def \( H = H, \text{ Use } L = L \)

```c
int main() {
    if (l1 == 42) {
        swap(h1, h2);
    } else {
        swap(l1, l2);
    }
    print l1;
}

void swap(ref int x, ref int y){
    int temp = y;
    y = x;
    x = temp;
}
```
IFC with Slicing
Correctness of slicing: Influence only possible, if node in slice
Correctness of slicing: Influence only possible, if node in slice
Correctness of slicing: Influence only possible, if node in slice

\( H \) not in slice of \( L \) \( \implies \) program noninterferent
Correctness of slicing: Influence only possible, if node in slice $H$ not in slice of $L \implies$ program noninterferent

Main theorem:

\[
\begin{align*}
  s_1 \approx_L s_2 & \quad \text{High} \notin \text{HRB-slice Low} & \quad \langle c, s_1 \rangle \Rightarrow \langle c', s_1 \rangle & \quad \langle c, s_2 \rangle \Rightarrow \langle c', s_2 \rangle \\
  s_1' \approx_L s_2'
\end{align*}
\]

Proof:

Mainly by slicing correctness theorem
Overview

Results

- Proved correctness of context-sensitive interprocedural slicing
- Proved that this slicing can guarantee classical noninterference
- Instantiated the framework with two different languages
  - simple While language with procedures
  - OO byte code language Jinja [Klein,Nipkow: TOPLAS '06]
- All proofs machine-checked in proof assistant Isabelle/HOL

Formalization size

- Framework about 19kLoC
- Instantiations: While 6.7kLoC, Jinja 3.4kLoC
- Noninterference proof: 1.4kLoC, Lifting: 2kLoC
Related Work

- **Horwitz-Reps-Binkley Algorithm:**
  
  [Horwitz et al.: TOPLAS '90], [Reps et al.: SIGSOFT '94]

- **Correctness of Slicing:**
  
  [Reps, Yang: TAPSOFT '89], [Ranganath et al.: TOPLAS '07], [Amtoft: IPL '08]
  
- All purely intraprocedural
- Restricted to simple imperative language
- Not machine-checked

- **Correctness of Flow Type Systems in Theorem Provers:**
  
  - Isabelle/HOL: [Barthe, Nieto: J. Comp. Sec. '07], [Beringer,Hofmann: CSF '07]
  - Coq: [Barthe et al.: ESOP '07], [Kammüller: Form. Asp. Comp. '07]

Future Work

- Verification of slicing-based IFC algorithm by Hammer et al.
- Requires new noninterference definition