Static Information Flow Analysis for Mobile Code in Dynamic Security Environments

Robert Grabowski

Dissertation an der Fakultät für Mathematik, Informatik und Statistik der Ludwig-Maximilians-Universität München

Disputationsvortrag, 23.03.2012
The iPhone security model is broken ... can it be fixed?

Motivation

The Huffington Post: Ramona Espareda
First Posten: 02/8/2012 8:34 am | Updated: 02/8/2012 4:10 pm

UPDATE: A Path spokesperson told The Huffington Post via email that Path version 2.0.5 for iOS has hit the iTunes App Store. The update will let users choose whether or not they want the app to pull all the contacts from their devices. As before, the current version of the Android app continues to allow users to opt in or out.

Path co-founder and CEO Dave Morin also posted an apology on the company’s official Tumblr blog and notified
Motivation

Smartphones:
- contain personal, sensitive data
- run downloadable software
- important application scenario for information security

Goal:
- prevent smartphone software from leaking sensitive data
- support developer with writing such software
- automatic & reliable verification method for end user and app store maintainer
- contents of this thesis: prototypical framework to reach this goal
Access Control [Pell & LaPadula, 1973]
- well-known security paradigm
- too coarse-grained: app may have legitimate access to both address book and Internet

Information Flow Control [Denning & Denning, 1977]
- allow access, but control where information flows
- exchange between accessed data resources
  - local files, memory, cloud storage, etc.
IF Control in a Nutshell

**social network application**

```
contacts ───> server
```

---

赋子安全域，规定流策

$$\rightarrow$$

**contacts**: med

**server**: low

high

---

IF control mechanism

应用为不干涉环境：

数据域A不直接影响数据域B，除非A $$\rightarrow$$ B

---

$$\frac{5}{32}$$
IF Control in a Nutshell

security environment assigns security domains, specifies flow policy →

social network application

contacts → server

contacts: med
server: low

high
med
low

application is noninterferent with respect to environment: data of domain A does not influence data of domain B unless A → B
IF Control in a Nutshell

application is **noninterferent** with respect to environment:
data of domain A does not influence data of domain B unless A $\leadsto B$.
Challenges in smartphone scenario

Security environments varies among smartphones

- user-defined security policies and domains
- availability of data objects not known before execution

Software should respect security environments everywhere

- provide facility to adapt behaviour to environment
- developer should not make assumptions about environment

Reliable security guarantee

- for programs deployed in bytecode form
- verifiable by user’s smartphone or app store maintainer
Related work

Static program analysis
- information flow type systems
  - for Java-like languages [Volpano et al., 1996], [Banerjee & Naumann, 2003]
  - for JVM-like bytecode [Barthe et al., 2005]
- formal verification for given fixed security environment
- proof of correctness: typable programs are noninterferent

Dynamic security environments
- only for other languages and security notions
  - functional language: $\lambda_{\text{DSec}}$ [Zheng & Myers 2007]
  - tracking updates to previously-known environments: Paralocks [Broberg & Sands 2010], RTI [Bandhakavi et al. 2008]
  - without correctness proof: Jif [Myers, 1999]
- not for bytecode programs
Contributions

Verification framework for software in dynamic security environments

- definition of *universal noninterference* for programs that respect arbitrary security environments

- high-level and bytecode languages for creating *privacy-aware software*

- *information flow type systems* with correctness proof to certify software

- designed for practical integration into typical app deployment framework
Approach: Verification Framework

Developer

- privacy-aware high-level program
  - verify
  - type-based source code verification
    - certify
    - universally non-interferent high-level program
      - compile
      - privacy-aware bytecode program

Abstract environment

- contacts: ?
- server: ?
Approach: Verification Framework

developer

- privacy-aware high-level program
  - verify
  - type-based source code verification
    - certify
    - universally non-interferent high-level program
      - compile
      - privacy-aware bytecode program
      - submit
      - type-based bytecode verification
      - certify
      - universally noninterferent bytecode program

app store

abstract environment

contacts: ?
server: ?
Approach: Verification Framework

- **Developer**
  - Privacy-aware high-level program
  - Type-based source code verification
  - Universally non-interferent high-level program
  - Compile
  - Privacy-aware bytecode program
  - Submit
  - Type-based bytecode verification
  - Universally noninterferent bytecode program
  - Verify
  - Certify

- **Abstract Environment**
  - Contacts: ?
  - Server: ?

- **App Store**
  - Type-based bytecode verification
  - Certify
  - Universally noninterferent bytecode program

- **Users**
  - Sue’s smartphone
    - Contacts: med
    - Server: low
  - Dave’s smartphone
    - Contacts: def
    - Server: def

...
Outline

1 Introduction

2 Technical Aspects
   - Privacy-Aware Software
   - Universal Noninterference
   - Type Systems
   - Implementation

3 Summary and Outlook
Privacy-aware software

New language “DSD”: Dynamic Security Domains

- Java-like, object-oriented imperative language
  with security domains as first-class values

\[
\begin{align*}
\textit{Val} & \ni \nu ::= n | a | k \\
\textit{Exp} & \ni e ::= n | x | e.f | e + e | e < e | \top | \bot | e \sqcup e | e \sqsubseteq e \\
\textit{Stmt} & \ni S ::= \text{skip} | S ; S | \text{if } e \text{ then } S \text{ else } S | \text{while } e \text{ do } S | \\
& \quad x ::= e | e.f ::= e | x ::= \text{new } C(\overline{e}) | x ::= e.m(\overline{e})
\end{align*}
\]

- security policy = lattice of domains,
  can only be referred to abstractly in the syntax

\[
\begin{align*}
\text{(highest domain)} & \quad \top \\
\text{(lowest domain)} & \quad \bot \\
\text{order } \rightsquigarrow & \quad \sqsubseteq \\
\text{least upper bound } \lor & \quad \sqcup
\end{align*}
\]
objects with runtime security domains are modelled via special $f_\delta$ field:

```haskell
class Contacts {
    f_\delta: dom_{⊥};
    list : Data_{f_\delta};
}

instance: contacts

class ServerBuffer {
    f_\delta: dom_{⊥};
    buf : Data_{f_\delta};
}

instance: server
```

- type of `contacts.list` is value of `contacts.f_\delta`
- type of `server.buf` is value of `server.f_\delta`
- restricted form of dependent type
Flow guards in privacy-aware software

```
class Contacts {
    fδ: dom⊥;
    list : Datafδ;
}
class ServerBuffer {
    fδ: dom⊥;
    buf : Datafδ;
}
```

actions that induce flows can be guarded:

```
server.buf := contacts.list;
```

- assignment causes flow from `contacts.list` to `server.buf`, i.e. from domain `contacts.fδ` to domain `server.fδ`
Flow guards in privacy-aware software

```java
class Contacts {
    fδ: dom⊥;
    list : Datafδ;
}

class ServerBuffer {
    fδ: dom⊥;
    buf : Datafδ;
}
```

actions that induce flows can be guarded:

```java
if contacts.fδ ⊑ server.fδ then
    server.buf := contacts.list;
else
    // show error message...
```

- assignment causes flow from `contacts.list` to `server.buf`, i.e. from domain `contacts.fδ` to domain `server.fδ`
- is only performed if this flow is permitted
Flow guards in privacy-aware software

```java
class Contacts {
    f_δ: dom_⊥;
    list : Data_{f_δ};
}
class ServerBuffer {
    f_δ: dom_⊥;
    buf : Data_{f_δ};
}
```

actions that induce flows can be guarded:

```java
if contacts.f_δ ⊑ server.f_δ then
    server.buf := contacts.list;
else
    // show error message...
```

- assignment causes flow from `contacts.list` to `server.buf`, i.e. from domain `contacts.f_δ` to domain `server.f_δ`
- is only performed if this flow is permitted
- field assignment is universally noninterferent
Visible fields in objects

Contacts

\[ f_δ = M : \text{dom} \]
\[ \text{tok} = 0xc19a0013 : \text{AuthToken} \]
\[ \text{list} = "my data" : \text{Data} \]
Visible fields in objects

- Type environments assign security types \(\{\top, \bot, f_\delta\}\) to fields

<table>
<thead>
<tr>
<th>Contacts</th>
</tr>
</thead>
<tbody>
<tr>
<td>(f_\delta = M : \text{dom}_\bot)</td>
</tr>
<tr>
<td>tok = 0xc19a0013 : AuthToken_\top</td>
</tr>
<tr>
<td>list = “my data” : Data_{f_\delta}</td>
</tr>
</tbody>
</table>
Visible fields in objects

- type environments assign security types \( \{ T, \bot, f_\delta \} \) to fields
- interpreted at runtime as a domain

<table>
<thead>
<tr>
<th>Contacts</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f_\delta = M : \text{dom}_\bot )</td>
</tr>
<tr>
<td>( \text{tok} = 0xc19a0013 : \text{AuthToken}_T )</td>
</tr>
<tr>
<td>( \text{list} = \text{&quot;my data&quot; : Data}<em>{f</em>\delta} )</td>
</tr>
</tbody>
</table>

example:

\[
\begin{array}{c}
\text{H} \\
\text{M} \\
\text{L}
\end{array}
\]

\( \rightsquigarrow = \)

\[
\begin{array}{c}
\text{T} \\
\text{M} \\
\text{L} \\
\bot
\end{array}
\]
Visible fields in objects

- type environments assign security types \( \{ T, \bot, f_\delta \} \) to fields
- interpreted at runtime as a domain

<table>
<thead>
<tr>
<th>Contacts</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f_\delta = M : \text{dom}_\bot )</td>
</tr>
<tr>
<td>tok = 0xc19a0013 : AuthToken ( T )</td>
</tr>
<tr>
<td>list = “my data” : Data ( f_\delta )</td>
</tr>
</tbody>
</table>

Example:

\[
\begin{array}{c}
\text{H} \\
\uparrow \\
\text{M} \\
\uparrow \\
\text{L} \\
\end{array}
\]
Visible fields in objects

- type environments assign security types \( \{ T, \bot, f_\delta \} \) to fields
- interpreted at runtime as a domain

Contacts

\[
\begin{align*}
\delta_f &= M : \text{dom}_\bot & L \\
\text{tok} &= 0xc19a0013 : \text{AuthToken}_T & H \\
\text{list} &= \text{“my data”} : \text{Data}_{f_\delta}
\end{align*}
\]

example:
Visible fields in objects

- type environments assign security types \( \{ \top, \bot, f_\delta \} \) to fields
- interpreted at runtime as a domain

<table>
<thead>
<tr>
<th>Contacts</th>
<th>example:</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f_\delta = M : \text{dom}_\bot )</td>
<td>( \text{H} \downarrow \text{M} \downarrow \text{L} )</td>
</tr>
<tr>
<td>tok = 0xc19a0013 : AuthToken_{\top}</td>
<td>( \text{H} \downarrow \text{M} \downarrow \text{L} )</td>
</tr>
<tr>
<td>list = “my data” : Data_{f_\delta}</td>
<td>( \text{H} \downarrow \text{M} \downarrow \text{L} )</td>
</tr>
</tbody>
</table>

\( \rightsquigarrow = \)
Visible fields in objects

- type environments assign security types \( \{ \top, \bot, f_\delta \} \) to fields
- interpreted at runtime as a domain
- define visibility with respect to domain \( k \) and policy \( \rightsquigarrow \)

<table>
<thead>
<tr>
<th>Contacts</th>
<th>example:</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f_\delta = M : \text{dom}_\bot )</td>
<td>( \rightsquigarrow = )</td>
</tr>
<tr>
<td>( \text{tok} = 0xc19a0013 : \text{AuthToken}_\top )</td>
<td>( H )</td>
</tr>
<tr>
<td>( \text{list} = \text{“my data”} : \text{Data}<em>{f</em>\delta} )</td>
<td>( M )</td>
</tr>
</tbody>
</table>

\( k = M \)
Visible fields in objects

- type environments assign security types \( \{ \top, \bot, f_\delta \} \) to fields
- interpreted at runtime as a domain
- define visibility with respect to domain \( k \) and policy \( \rightsquigarrow \)

<table>
<thead>
<tr>
<th>Contacts</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f_\delta = M : \text{dom}_\perp )</td>
</tr>
<tr>
<td>( \text{tok} = 0xc19a0013 : \text{AuthToken}_\top )</td>
</tr>
<tr>
<td>( \text{list} = \text{“my data” : Data}<em>{f</em>\delta} )</td>
</tr>
</tbody>
</table>

**Example:**

\[
\rightsquigarrow = \begin{array}{c}
H \\
M \\
L
\end{array}
\]

\( k = M \)
Visible fields in objects

- Type environments assign security types \( \{ \top, \bot, f_\delta \} \) to fields.
- Interpreted at runtime as a domain.
- Define visibility with respect to domain \( k \) and policy \( \rightsquigarrow \).

<table>
<thead>
<tr>
<th>Contacts</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f_\delta = M : \text{dom}_\bot )</td>
</tr>
<tr>
<td>( \text{tok} = 0xc19a0013 : \text{AuthToken}_\top )</td>
</tr>
<tr>
<td>( \text{list} = ) “my data” : Data( f_\delta )</td>
</tr>
</tbody>
</table>

**Example:**

\[
\begin{array}{c}
\text{H} \\
\uparrow \\
\text{M} \\
\uparrow \\
\text{L} \\
\uparrow \\
\bot
\end{array}
\]

\( \rightsquigarrow = \)

\( k = M \)
Visible fields in objects

- type environments assign security types \{\top, \bot, f_\delta\} to fields
- interpreted at runtime as a domain
- define visibility with respect to domain \(k\) and policy \(\rightsquigarrow\)

<table>
<thead>
<tr>
<th>Contacts</th>
<th>example:</th>
</tr>
</thead>
<tbody>
<tr>
<td>(f_\delta = M : \text{dom} \bot)</td>
<td>(H \leftrightarrow M \leftrightarrow L)</td>
</tr>
<tr>
<td>(\text{tok} = 0xc19a0013 : \text{AuthToken}\top)</td>
<td>(H \leftrightarrow M \leftrightarrow L)</td>
</tr>
<tr>
<td>(\text{list} = \text{“my data”} : \text{Data} f_\delta)</td>
<td>(M \leftrightarrow M \leftrightarrow L)</td>
</tr>
</tbody>
</table>

\(k = M\)
Indistinguishable program states

Two indistinguishable objects:
- equality on all $k$, $\sim \Rightarrow$ - visible fields

Contacts

\[
\begin{align*}
  f_\delta &= M : \text{dom}_{\bot} \\
  \text{tok} &= 0xc19a0013 : \text{AuthToken} \\
  \text{list} &= \text{“my data”} : \text{Data}_{f_\delta}
\end{align*}
\]

Contacts

\[
\begin{align*}
  f_\delta &= M : \text{dom}_{\bot} \\
  \text{tok} &= 0xc18a0107 : \text{AuthToken}_T \\
  \text{list} &= \text{“my data”} : \text{Data}_{f_\delta}
\end{align*}
\]
Indistinguishable program states

Two indistinguishable objects:
- equality on all \( k \), \( \sim \) - visible fields

Contacts

\[
\begin{align*}
\delta_f &= \text{M} : \text{dom}_\perp \\
\text{tok} &= 0xc19a0013 : \text{AuthToken} \\
\text{list} &= \text{“my data”} : \text{Data}_{f_\delta}
\end{align*}
\]

Contacts

\[
\begin{align*}
\delta_f &= \text{M} : \text{dom}_\perp \\
\text{tok} &= 0xc18a0107 : \text{AuthToken}_\top \\
\text{list} &= \text{“my data”} : \text{Data}_{f_\delta}
\end{align*}
\]

Two indistinguishable program states: \( \sigma_1 \equiv_{k, \sim} \sigma_2 \)
- all corresponding objects are indistinguishable
Indistinguishable program states

Two indistinguishable objects:

- equality on all $k$, $\sim$ - visible fields

\[
\begin{array}{l}
\text{Contacts} \\
\quad f_δ = M : \text{dom}_{⊥} \\
\quad \text{tok} = 0xc19a0013 : \text{AuthToken} \\
\quad \text{list} = "my data" : \text{Data}_{f_δ}
\end{array}
\]

\[
\begin{array}{l}
\text{Contacts} \\
\quad f_δ = M : \text{dom}_{⊥} \\
\quad \text{tok} = 0xc18a0107 : \text{AuthToken}_{⊥} \\
\quad \text{list} = "my data" : \text{Data}_{f_δ}
\end{array}
\]

Two indistinguishable program states: $σ_1 \equiv_{k, \sim} σ_2$

- all corresponding objects are indistinguishable
- in fact parametrized with partial bijections $β$
  to handle different address allocations [Banerjee/Naumann 2003]
- states also contain local variables
A statement $S$ is universally noninterferent if for all policies $\rightsquigarrow$, security domains $k$, program states $\sigma_1, \sigma_2, \sigma'_1, \sigma'_2$:

$$\text{if } \sigma_1 =_{k, \rightsquigarrow} \sigma'_1 \text{ and } \sigma_1 \xrightarrow{S} \sigma_2 \text{ and } \sigma'_1 \xrightarrow{S} \sigma'_2, \text{ then } \sigma_2 =_{k, \rightsquigarrow} \sigma'_2$$
A statement $S$ is universally noninterferent if for all policies $\rightsquigarrow$, security domains $k$, program states $\sigma_1, \sigma_2, \sigma'_1, \sigma'_2$:

$$\text{if } \sigma_1 =_{k, \rightsquigarrow} \sigma'_1 \text{ and } \sigma_1 \xrightarrow{S} \sigma_2 \text{ and } \sigma'_1 \xrightarrow{S} \sigma'_2, \text{ then } \sigma_2 =_{k, \rightsquigarrow} \sigma'_2$$

indistinguishable with respect to $k, \rightsquigarrow$

big-step operational semantics for statements
Universal noninterference

A statement $S$ is universally noninterferent if for all policies $\rightsquigarrow$, security domains $k$, program states $\sigma_1, \sigma_2, \sigma'_1, \sigma'_2$:

$$\text{if } \sigma_1 =_k \rightsquigarrow \sigma'_1 \text{ and } \sigma_1 \xrightarrow{S} \sigma_2 \text{ and } \sigma'_1 \xrightarrow{S} \sigma'_2, \text{ then } \sigma_2 =_k \rightsquigarrow \sigma'_2$$

- follows standard termination-insensitive definition of noninterference
- extended notion of state indistinguishability:
  - must hold for all policies and all possible domains of data objects stored in their $f_\delta$ field values
Type system for DSD

class Contacts {
  \( f_\delta : \text{dom}_{\perp} \);
  list : Data_{f_\delta};
}

class ServerBuffer {
  \( f_\delta : \text{dom}_{\perp} \);
  buf : Data_{f_\delta};
}

if \( \text{contacts} \cdot f_\delta \sqsubseteq \text{server} \cdot f_\delta \) then

server.buf := contacts.list;

Type system for DSD

```java
class Contacts {
    f_δ: dom⊥;
    list : Data_{f_δ};
}

class ServerBuffer {
    f_δ: dom⊥;
    buf : Data_{f_δ};
}
```

\[
\text{if } \text{contacts}.f_δ \sqsubseteq \text{server}.f_δ \text{ then }
\]

\[
\text{server.buf} := \text{contacts.list};
\]

**expected type:** `server.f_δ`

**type of expression:** `contacts.f_δ`

## 1) types: symbolic expressions that refer to a security domain
**Type system for DSD**

```plaintext
class Contacts {
    \( f_\delta : \text{dom}_{\bot}; \)
    list : Data_{f_\delta};
}
class ServerBuffer {
    \( f_\delta : \text{dom}_{\bot}; \)
    buf : Data_{f_\delta};
}
```

```plaintext
if \( \text{contacts}.f_\delta \sqsubseteq \text{server}.f_\delta \) then

\( \text{server}.buf := \text{contacts}.list; \)
```

- **expected type:** \( \text{server}.f_\delta \)
- **type of expression:** \( \text{contacts}.f_\delta \)

1. **types:** symbolic expressions that refer to a security domain
2. **type system generates symbolic flow conditions**
Type system for DSD

class Contacts {
    \(f_\delta: \text{dom}_{\bot};\)
    list : Data_{f_\delta};
}

class ServerBuffer {
    \(f_\delta: \text{dom}_{\bot};\)
    buf : Data_{f_\delta};
}

\textbf{if } \textit{contacts.}f_\delta \sqsubseteq \textit{server.}f_\delta \textbf{ then}

\textit{server.}buf := \textit{contacts.}list;

\textbf{flow information for } \textbf{then} \textbf{ branch}

\textbf{secure if: } \textit{contacts.}f_\delta \sqsubseteq \textit{server.}f_\delta

\textbf{expected type: } \textit{server.}f_\delta

\textbf{type of expression: } \textit{contacts.}f_\delta

1. types: symbolic expressions that refer to a security domain
2. type system generates symbolic flow conditions
3. solved by using collected information about allowed flows
1. Expressions as types

Labels

- values of domain fields not statically available
- the type system uses a subclass of expressions as types, called “labels”

\[ \Gamma \vdash e : \ell \]

where \[ \ell ::= \top | \bot | e.f_\delta | \ell \sqcup \ell \]
2. Side conditions: flows between labels

Type system handles among others:

- **direct flows:**
  
  \[ y := x \times 2 \]

- **indirect flows:**
  
  \[ x \cdot f := y \times 4 \]

Type system checks assignments.
2. Side conditions: flows between labels

Type system handles among others:

- **direct flows:**
  - \( y := x \times 2 \)
  - \( x \cdot f := y \times 4 \)

- **indirect flows:**
  - if \( x > 0 \) then
    - \( y := 1 \)

  type system checks assignments

  type system maintains *program counter label*
2. Side conditions: flows between labels

Type system handles among others:

- **direct flows:** \( y \top := x \bot * 2 \)
  
  \[ x \bot \cdot f_x \cdot \delta := y \top * 4 \]

- **indirect flows:**

  **if** \( x \top > 0 \) **then**

  \[ y \bot := 1 \quad \] \[ pc = \top \]

Side conditions on the order of labels

- **example:** \( \top \sqsubseteq x \cdot f_\delta \)
3. Solving flow conditions on labels

Typing judgements for statements: $\Gamma, pc \vdash Q \{S\} Q'$

- collect flow guard information in sets $Q$

\[
\Gamma, pc \vdash Q \cup (\ell_1 \sqsubseteq \ell_2) \{S_1\} Q' \\
\Gamma, pc \vdash Q \{S_2\} Q' \\
\Gamma, pc \vdash Q \{\text{if } \ell_1 \sqsubseteq \ell_2 \text{ then } S_1 \text{ else } S_2\} Q'
\]

- simple state predicates, similar to Hoare logic
3. Solving flow conditions on labels

Typing judgements for statements: $\Gamma, pc \vdash Q \{S\} Q'$

- collect flow guard information in sets $Q$

$$
\begin{align*}
\Gamma, pc & \vdash Q \cup (\ell_1 \sqsubseteq \ell_2) \{S_1\} Q' \\
\Gamma, pc & \vdash Q \{S_2\} Q' \\
\Gamma, pc & \vdash Q \{\text{if } \ell_1 \sqsubseteq \ell_2 \text{ then } S_1 \text{ else } S_2\} Q'
\end{align*}
$$

- simple state predicates, similar to Hoare logic
- soundness: if a statement is typable, then it is universally noninterferent
3. Solving flow conditions on labels

Typing judgements for statements: $\Gamma, \text{pc} \vdash Q \{S\} Q'$
- collect flow guard information in sets $Q$

$$
\Gamma, \text{pc} \vdash Q \cup (\ell_1 \sqsubseteq \ell_2) \{S_1\} Q' \\
\Gamma, \text{pc} \vdash Q \{S_2\} Q' \\
\Gamma, \text{pc} \vdash Q \{\text{if } \ell_1 \sqsubseteq \ell_2 \text{ then } S_1 \text{ else } S_2\} Q'
$$

- simple state predicates, similar to Hoare logic
- soundness: if a statement is typable, then it is universally noninterferent ...for pre-states that satisfy $Q$
Design aspects

light-weight extension to Java-like language and classic IF type system
  - few new language constructs
  - $Q$ sets only contain simple positive flow facts, no complex predicates

goals:
  - type system is decidable, suggests inference algorithm
  - language and type system easy to understand by the programmer
Bytecode language

```
if contacts.f_δ ⊆ server.f_δ then server.buf := contacts.list; else skip;
```

- stack-based, unstructured language

<table>
<thead>
<tr>
<th>i</th>
<th>pre-stack ρ</th>
<th>instruction</th>
<th>post-stack ρ'</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[]</td>
<td>load contacts</td>
<td>[a_{14}]</td>
</tr>
<tr>
<td>2</td>
<td>[a_{14}]</td>
<td>getf f_δ</td>
<td>[L]</td>
</tr>
<tr>
<td>3</td>
<td>[L]</td>
<td>load server</td>
<td>[a_8, H]</td>
</tr>
<tr>
<td>4</td>
<td>[a_8, L]</td>
<td>getf f_δ</td>
<td>[H, L]</td>
</tr>
<tr>
<td>5</td>
<td>[H, L]</td>
<td>prim ⊑</td>
<td>[1]</td>
</tr>
<tr>
<td>6</td>
<td>[1]</td>
<td>bnz 8</td>
<td>[]</td>
</tr>
<tr>
<td>7</td>
<td>[]</td>
<td>jmp 12</td>
<td>[]</td>
</tr>
<tr>
<td>8</td>
<td>[]</td>
<td>load contacts</td>
<td>[a_{14}]</td>
</tr>
<tr>
<td>9</td>
<td>[a_{14}]</td>
<td>getf list</td>
<td>[“…”]</td>
</tr>
<tr>
<td>10</td>
<td>[“…”]</td>
<td>load server</td>
<td>[a_8, “…”]</td>
</tr>
<tr>
<td>11</td>
<td>[a_8, “…”]</td>
<td>putf buf</td>
<td>[]</td>
</tr>
<tr>
<td>12</td>
<td>[]</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
Bytecode language: challenges for the type system

```
if contacts.f_δ ⊑ server.f_δ  then  server.buf := contacts.list;  else  skip ;
```

<table>
<thead>
<tr>
<th>i</th>
<th>instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>load contacts</td>
</tr>
<tr>
<td>2</td>
<td>getf f_δ</td>
</tr>
<tr>
<td>3</td>
<td>load server</td>
</tr>
<tr>
<td>4</td>
<td>getf f_δ</td>
</tr>
<tr>
<td>5</td>
<td>prim ⊑</td>
</tr>
<tr>
<td>6</td>
<td>bnz 8</td>
</tr>
<tr>
<td>7</td>
<td>jmp 12</td>
</tr>
<tr>
<td>8</td>
<td>load contacts</td>
</tr>
<tr>
<td>9</td>
<td>getf list</td>
</tr>
<tr>
<td>10</td>
<td>load server</td>
</tr>
<tr>
<td>11</td>
<td>putf buf</td>
</tr>
<tr>
<td>12</td>
<td>...</td>
</tr>
</tbody>
</table>

Problems:
1. control flow structure is lost, needed to handle indirect flows
2. high-level expressions are lost, needed to determine side conditions and to collect allowed flows

Solutions:
1. confluence point stack
2. intermediate representation
Bytecode language: challenges for the type system

```plaintext
if \( \text{contacts}.f_\delta \sqsubseteq \text{server}.f_\delta \) then  
  \text{server}.buf := \text{contacts}.list;  
else  
  \text{skip};
```

<table>
<thead>
<tr>
<th>( i )</th>
<th>instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>load \text{contacts}</td>
</tr>
<tr>
<td>2</td>
<td>getf ( f_\delta )</td>
</tr>
<tr>
<td>3</td>
<td>load \text{server}</td>
</tr>
<tr>
<td>4</td>
<td>getf ( f_\delta )</td>
</tr>
<tr>
<td>5</td>
<td>prim ( \sqsubseteq )</td>
</tr>
<tr>
<td>6</td>
<td>bnz 8</td>
</tr>
<tr>
<td>7</td>
<td>jmp 12</td>
</tr>
<tr>
<td>8</td>
<td>load \text{contacts}</td>
</tr>
<tr>
<td>9</td>
<td>getf \text{list}</td>
</tr>
<tr>
<td>10</td>
<td>load \text{server}</td>
</tr>
<tr>
<td>11</td>
<td>putf \text{buf}</td>
</tr>
<tr>
<td>12</td>
<td>...</td>
</tr>
</tbody>
</table>

Problems:

1. control flow structure is lost, needed to handle indirect flows
Bytecode language: challenges for the type system

```
if \( \text{contacts}.f_\delta \sqsubseteq \text{server}.f_\delta \) then \( \text{server.buf} := \text{contacts.list}; \) else \text{skip} ;
```

<table>
<thead>
<tr>
<th>( i )</th>
<th>instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>load ( \text{contacts} )</td>
</tr>
<tr>
<td>2</td>
<td>getf ( f_\delta )</td>
</tr>
<tr>
<td>3</td>
<td>load ( \text{server} )</td>
</tr>
<tr>
<td>4</td>
<td>getf ( f_\delta )</td>
</tr>
<tr>
<td>5</td>
<td>prim ( \sqsubseteq )</td>
</tr>
<tr>
<td>6</td>
<td>bnz 8</td>
</tr>
<tr>
<td>7</td>
<td>jmp 12</td>
</tr>
<tr>
<td>8</td>
<td>load ( \text{contacts} )</td>
</tr>
<tr>
<td>9</td>
<td>getf ( \text{list} )</td>
</tr>
<tr>
<td>10</td>
<td>load ( \text{server} )</td>
</tr>
<tr>
<td>11</td>
<td>putf ( \text{buf} )</td>
</tr>
<tr>
<td>12</td>
<td>...</td>
</tr>
</tbody>
</table>

Problems:

1. control flow structure is lost, needed to handle indirect flows
2. high-level expressions are lost, needed to determine side conditions and to collect allowed flows
Bytecode language: challenges for the type system

```
if \( \mathit{contacts} \cdot f_\delta \subseteq \mathit{server} \cdot f_\delta \) then \( \mathit{server} \cdot \mathit{buf} := \mathit{contacts} \cdot \mathit{list} \); else skip;
```

<table>
<thead>
<tr>
<th>i</th>
<th>instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>load ( \mathit{contacts} )</td>
</tr>
<tr>
<td>2</td>
<td>getf ( f_\delta )</td>
</tr>
<tr>
<td>3</td>
<td>load ( \mathit{server} )</td>
</tr>
<tr>
<td>4</td>
<td>getf ( f_\delta )</td>
</tr>
<tr>
<td>5</td>
<td>prim ( \subseteq )</td>
</tr>
<tr>
<td>6</td>
<td>bnz 8</td>
</tr>
<tr>
<td>7</td>
<td>jmp 12</td>
</tr>
<tr>
<td>8</td>
<td>load ( \mathit{contacts} )</td>
</tr>
<tr>
<td>9</td>
<td>getf ( \mathit{list} )</td>
</tr>
<tr>
<td>10</td>
<td>load ( \mathit{server} )</td>
</tr>
<tr>
<td>11</td>
<td>putf ( \mathit{buf} )</td>
</tr>
<tr>
<td>12</td>
<td>...</td>
</tr>
</tbody>
</table>

Problems:

1. control flow structure is lost, needed to handle indirect flows
2. high-level expressions are lost, needed to determine side conditions and to collect allowed flows

Solutions:

1. confluence point stack
2. intermediate representation
### Confluence point stack [Medel et al, 2005]

<table>
<thead>
<tr>
<th>i</th>
<th>BC instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>cpush 14</td>
</tr>
<tr>
<td>2</td>
<td>load contacts</td>
</tr>
<tr>
<td>3</td>
<td>getf $f_\delta$</td>
</tr>
<tr>
<td>4</td>
<td>load server</td>
</tr>
<tr>
<td>5</td>
<td>getf $f_\delta$</td>
</tr>
<tr>
<td>6</td>
<td>prim $\sqsubseteq$</td>
</tr>
<tr>
<td>7</td>
<td>bnz 9</td>
</tr>
<tr>
<td>8</td>
<td>cpop 14</td>
</tr>
<tr>
<td>9</td>
<td>load contacts</td>
</tr>
<tr>
<td>10</td>
<td>getf list</td>
</tr>
<tr>
<td>11</td>
<td>load server</td>
</tr>
<tr>
<td>12</td>
<td>putf $buf$</td>
</tr>
<tr>
<td>13</td>
<td>cpop 14</td>
</tr>
<tr>
<td>14</td>
<td>...</td>
</tr>
</tbody>
</table>

- pseudo-instructions inserted by compiler
- mark control dependence regions
- verified by type system
### Stackless Intermediate Representation (IR) [Demange et al, 2010]

<table>
<thead>
<tr>
<th>$i$</th>
<th>BC Instruction</th>
<th>$\rightarrow$</th>
<th>IR Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>cpush 14</td>
<td></td>
<td>cpush 14</td>
</tr>
<tr>
<td>2</td>
<td>load contacts</td>
<td></td>
<td>nop</td>
</tr>
<tr>
<td>3</td>
<td>getf $f_\delta$</td>
<td></td>
<td>nop</td>
</tr>
<tr>
<td>4</td>
<td>load server</td>
<td></td>
<td>nop</td>
</tr>
<tr>
<td>5</td>
<td>getf $f_\delta$</td>
<td></td>
<td>nop</td>
</tr>
<tr>
<td>6</td>
<td>prim $\sqsubseteq$</td>
<td></td>
<td>nop</td>
</tr>
<tr>
<td>7</td>
<td>bnz 9</td>
<td></td>
<td>if contacts.$f_\delta \sqsubseteq$ server.$f_\delta$ 9</td>
</tr>
<tr>
<td>8</td>
<td>cpop 14</td>
<td></td>
<td>cpop 14</td>
</tr>
<tr>
<td>9</td>
<td>load contacts</td>
<td></td>
<td>nop</td>
</tr>
<tr>
<td>10</td>
<td>getf list</td>
<td></td>
<td>nop</td>
</tr>
<tr>
<td>11</td>
<td>load server</td>
<td></td>
<td>nop</td>
</tr>
<tr>
<td>12</td>
<td>putf buf</td>
<td></td>
<td>server.buf := contacts.list</td>
</tr>
<tr>
<td>13</td>
<td>cpop 14</td>
<td></td>
<td>cpop 14</td>
</tr>
<tr>
<td>14</td>
<td>...</td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

- “decompile” expressions and some statements; reversible operation
- proof: semantics is preserved
- type system shows noninterference of IR program, thus of bytecode
The framework revisited

privacy-aware DSD program
The framework revisited

privacy-aware DSD program → type check → DSD type derivation
The framework revisited

privacy-aware DSD program

compile

dtype check

DSD type derivation

privacy-aware bytecode program

IR version of bytecode program

translate

IR type derivation

type check

high-level program is universally noninterferent

soundness

IR program is universally noninterferent

semantic correspondence between BC and IR

typability is preserved
The framework revisited

privacy-aware DSD program

compile

privacy-aware bytecode program

translate

IR version of bytecode program

DSD type derivation

type check

high-level program is universally noninterferent

soundness

IR program is universally noninterferent

semantic correspondence

typability is preserved

IR version of bytecode program
privacy-aware DSD program

dtype check

DSD type derivation

compile

privacy-aware bytecode program

translate

IR version of bytecode program

type check

IR type derivation
The framework revisited

privacy-aware DSD program

 compile

privacy-aware bytecode program

 translate

IR version of bytecode program

type check

IR type derivation

dtype check

DSD type derivation

soundness

IR program is universally noninterferent

typability is preserved

semantic correspondence between BC and IR

high-level program is universally noninterferent
The framework revisited

privacy-aware DSD program
- type check
  - DSD type derivation
    - soundness
      - high-level program is universally noninterferent

compile
privacy-aware bytecode program
- translate
  - IR version of bytecode program
    - type check
      - IR type derivation
        - soundness
          - IR program is universally noninterferent
The framework revisited

privacy-aware DSD program

compile

privacy-aware bytecode program

translate

IR version of bytecode program

type check

IR type derivation

soundness

IR program is universally noninterferent

semantic correspondence between BC and IR

IR program is universally noninterferent

high-level program is universally noninterferent

dotted line indicates soundness

type check

DSD type derivation

dotted line indicates soundness

dotted line indicates typability is preserved
The framework revisited

- Privacy-aware DSD program
  - Compile
  - Translate
  - IR version of bytecode program
    - Type check
    - IR type derivation
      - Typability is preserved
      - Soundness
      - IR program is universally noninterferent
        - Semantic correspondence between BC and IR
        - Bytecode program is universally noninterferent

- DSD type derivation
  - Type check
  - Soundness
  - High-level program is universally noninterferent
Implementation

OCaml implementation of framework: interpreter, compiler, type inference (∼5000 loc)

class Buffer {
    fdelta : BOT;
    contents : FDELTA;

    Bot read(xdelta: BOT) : XDELTA {
        ret := this.contents;
    }

    Bot write(xdelta: BOT, s: XDELTA) : BOT {
        this.contents := s;
        ret := 0;
    }
}

class Main {
    ....
}
Summary

Results

- noninterference notion that is universal in flow policy and dynamic security domains of objects
- high-level and bytecode languages for programs that can adapt their behaviour to the security environment
- type systems to verify and certify universal noninterference
Towards dynamic security domains in practice

Technical aspects
- extend common programming languages, or provide library integration into existing development tools and distribution frameworks

Broader themes
- declassification of information
- covert channels, e.g. timing and termination behaviour
- user-friendly presentation of information flow security
References

D. Denning and P. Denning
Certification of Programs for Secure Information Flow

D. Volpano, G. Smith, and C. Irvine
A Sound Type System for Secure Flow Analysis

A. Myers
JFlow: Practical Mostly-Static Information Flow Control

A. Banerjee and D. Naumann
Stack-based access control and secure information flow
References

R. Grabowski and L. Beringer
Noninterference with Dynamic Security Domains and Policies

M. Hofmann and L. Beringer
Secure Information Flow and Program Logics

N. Kobayashi and K. Shirane
Type-Based Information Flow Analysis for Low-Level Languages
Workshop on Programming Languages and Systems, 2002.

D. Demange, T. Jensen, and D. Pichardie
A Provably Correct Stackless Intermediate Representation for Java Bytecode
Asian Symposium on Programming Languages and Systems, 2010.
Thank you for your attention!
More features of the analysis

Methods with complex signatures

class Contacts {
  f_δ: dom_⊥;
  list : Data_{f_δ};
  write(x_δ : dom_⊥, x: Data_{x_δ})
    requires { x_δ ⊑ this.f_δ }
    ensures { }
}

- type of formal parameter may refer to special $x_δ$ argument

Meta-label monotonicity

- if an expression $e$ has a label $\ell$, then $\ell$ is always at least as confidential as the label of $\ell$ itself

  “the fact that something is public cannot be private”
Runtime security monitoring of the application

- not trivial: data may leak even when application does nothing
  
  if address book contains 'Barack Obama' then
  send address book to server;

- needs to be combined with static analysis
- or rewrite program so that it can be exhaustively monitored
Expression typing

Labels

\[ \text{Lab} \ni \ell ::= \bot | \top | \pi.f_{\delta} | x_{\delta} | \ell_1 \sqcup \ell_2 \]

Typing judgement \( \Gamma \vdash e : \ell \)

\[
\frac{c \in \{n, \top, \bot\}}{\Gamma \vdash c : \bot}
\]

\[
\frac{}{\Gamma \vdash x : \Gamma(x)}
\]

\[
\frac{\Gamma \vdash \pi : \ell}{\Gamma \vdash \pi.f : \Phi^{\pi}(f) \sqcup \ell}
\]

\[
\frac{\circ \in \{\text{op}, \sqcup, \sqsubseteq\}}{\Gamma \vdash e_1 : \ell_1 \quad \Gamma \vdash e_2 : \ell_2}{\Gamma \vdash e_1 \circ e_2 : \ell_1 \sqcup \ell_2}
\]

where \( \Phi^{\pi}(f) \) is the qualified type of the field \( f \)
Reasoning with labels

\[
\begin{align*}
(l_1, l_2) &\in Q \\
\therefore l_1 &\sqsubseteq_Q l_2 \\
\hline
l &\sqsubseteq_Q l \\
\therefore l_1 &\sqsubseteq_Q l_3 \\
\hline
l_1 &\sqsubseteq_Q l_2 \\
l_2 &\sqsubseteq_Q l_1 \\
\therefore l_1 &\equiv_Q l_2 \\
\hline
\bot &\sqsubseteq_Q l \\
\hline
l &\sqsubseteq_Q \top \\
\hline
l &\equiv_Q l' \\
\hline
l_1 &\sqsubseteq_Q l_3 \\
l_2 &\sqsubseteq_Q l_4 \\
\therefore l_1 \sqcup l_2 &\sqsubseteq_Q l_3 \sqcup l_4 \\
\hline
l &\equiv_Q l \sqcup l' \\
\hline
l_1 \sqcup l_2 &\equiv_Q l_2 \sqcup l_1 \\
\hline
l_1 \sqcup (l_2 \sqcup l_3) &\equiv_Q (l_1 \sqcup l_2) \sqcup l_3 \\
\hline
l_1 &\equiv_Q l_2 \\
\therefore l_1 &\sqsubseteq_Q l_2 \\
\hline
\hline
l_1 &\equiv_Q l_2 \\
\therefore l_2 &\sqsubseteq_Q l_1 \\
\hline
\end{align*}
\]
Statement typing I

\[
\Gamma, pc \vdash \{Q_0\} \quad S \quad \{Q'_0\} \\
Q \Rightarrow Q_0 \quad Q'_0 \Rightarrow Q' \\
\frac{}{\Gamma, pc \vdash \{Q\} \quad S \quad \{Q'\}}
\]

\[
\Gamma, pc \vdash \{Q\} \quad \text{skip} \quad \{Q\}
\]

\[
\Gamma, pc \vdash \{Q\} \quad S_1 \quad \{Q'\} \\
\Gamma, pc \vdash \{Q'\} \quad S_2 \quad \{Q''\} \\
\frac{}{\Gamma, pc \vdash \{Q\} \quad S_1; S_2 \quad \{Q''\}}
\]

\[
\Gamma \vdash e : \ell \\
\Gamma, pc \sqcup \ell \vdash \{Q\} \quad S \quad \{Q\}
\]

\[
\Gamma, pc \vdash \{Q\} \quad \text{while } e \text{ do } S \quad \{Q\}
\]

\[
\Gamma \vdash e : \ell \\
\Gamma, pc \sqcup \ell \vdash \{Q\} \quad S_1 \quad \{Q'\} \\
\Gamma, pc \sqcup \ell \vdash \{Q\} \quad S_2 \quad \{Q'\} \\
\frac{}{\Gamma, pc \vdash \{Q\} \quad \text{if } e \text{ then } S_1 \text{ else } S_2 \quad \{Q'\}}
\]

\[
\Gamma \vdash \ell_1 \sqsubseteq \ell_2 : \ell \\
\Gamma, pc \sqcup \ell \vdash \{Q, (\ell_1, \ell_2)\} \quad S_1 \quad \{Q'\} \\
\Gamma, pc \sqcup \ell \vdash \{Q\} \quad S_2 \quad \{Q'\} \\
\frac{}{\Gamma, pc \vdash \{Q\} \quad \text{if } \ell_1 \sqsubseteq \ell_2 \text{ then } S_1 \text{ else } S_2 \quad \{Q'\}}
\]
Statement typing II

\[ \Gamma \vdash e : \ell \quad \ell \sqcup pc \sqsubseteq Q \quad \Gamma(x) \quad x \neq x_\delta \quad x \notin pc \]
\[ \Gamma, pc \vdash \{Q[e/x] \cup Q'\} \quad x := e \quad \{Q\} \]

\[ \Gamma \vdash \pi : \ell_1 \quad \Gamma \vdash e : \ell_2 \quad \ell_1 \sqcup \ell_2 \sqcup pc \sqsubseteq Q \quad \Phi^\pi(f) \]
\[ f \neq f_\delta \quad f \notin pc \quad f \notin Q[e/\pi.f] \]
\[ \Gamma, pc \vdash \{Q[e/\pi.f] \cup Q'\} \quad \pi.f := e \quad \{Q\} \]

\[ \Gamma \vdash \overline{e} : \overline{\ell} \quad \text{fields}(C) = \overline{f} \quad \Phi^* = \Phi^x[\overline{e}[1]/x.f_\delta] \]
\[ \overline{\ell} \sqsubseteq Q' \quad \Phi^*(\overline{f}) \quad pc \sqsubseteq Q' \quad \Gamma(x) \quad x \notin pc \quad x \neq x_\delta \quad x \notin Q[\overline{e}/x.\overline{f}] \]
\[ \Gamma, pc \vdash \{Q[\overline{e}/x.\overline{f}] \cup Q'\} \quad x := \text{new} \ C(\overline{e}) \quad \{Q\} \]

\[ \text{msig}(m) = [\Gamma_m, pc_m, Q_m, Q'_m] \quad \text{margs}(m) = \overline{x} \]
\[ \Gamma \vdash \pi : \ell_r \quad \Gamma \vdash \overline{e} : \overline{\ell} \quad \Gamma^* = \Gamma_m[\overline{e}[1]/x_\delta] \quad \ell_r \sqsubseteq Q' \quad \Gamma^*(\text{this}) \]
\[ \overline{\ell} \sqsubseteq Q' \quad \Gamma^*(\overline{x}) \quad pc \sqsubseteq Q' \quad pc_m[\overline{e}/\overline{x}][\pi/\text{this}] \quad \Gamma^*(\text{ret}) \sqcup pc \sqsubseteq Q \quad \Gamma(x) \]
\[ x \notin pc, Q \quad \forall f. \ f \neq f_\delta \Rightarrow f \notin pc, Q \quad x \neq x_\delta \]
\[ \Gamma, pc \vdash \{Q_m[\overline{e}/\overline{x}][\pi/\text{this}] \cup Q' \cup Q\} \quad x := \pi.m(\overline{e}) \quad \{Q'_m[x/\text{ret}] \cup Q\} \]
Confluence point stack

```plaintext
if \((\text{contacts}.f_\delta \sqsubseteq \text{server}.f_\delta)\top\) then \text{server}.buf := \text{contacts}.list; else skip;
```

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>pc</td>
<td>Δ</td>
<td>instruction</td>
<td>pc'</td>
<td>Δ'</td>
</tr>
<tr>
<td>---</td>
<td>----</td>
<td>----</td>
<td>-------------------</td>
<td>------</td>
<td>-----</td>
</tr>
<tr>
<td>1</td>
<td>⊥</td>
<td>[]</td>
<td>cpush 14</td>
<td>⊥</td>
<td>[(14, ⊥)]</td>
</tr>
<tr>
<td>2</td>
<td>⊥</td>
<td>[(14, ⊥)]</td>
<td>load \text{contacts}</td>
<td>⊥</td>
<td>[(14, ⊥)]</td>
</tr>
<tr>
<td>3</td>
<td>⊥</td>
<td>[(14, ⊥)]</td>
<td>getf (f_\delta)</td>
<td>⊥</td>
<td>[(14, ⊥)]</td>
</tr>
<tr>
<td>4</td>
<td>⊥</td>
<td>[(14, ⊥)]</td>
<td>load \text{server}</td>
<td>⊥</td>
<td>[(14, ⊥)]</td>
</tr>
<tr>
<td>5</td>
<td>⊥</td>
<td>[(14, ⊥)]</td>
<td>getf (f_\delta)</td>
<td>⊥</td>
<td>[(14, ⊥)]</td>
</tr>
<tr>
<td>6</td>
<td>⊥</td>
<td>[(14, ⊥)]</td>
<td>prim \sqsubseteq</td>
<td>⊥</td>
<td>[(14, ⊥)]</td>
</tr>
<tr>
<td>7</td>
<td>⊥</td>
<td>[(14, ⊥)]</td>
<td>bnz 9</td>
<td>⊤</td>
<td>[(14, ⊥)]</td>
</tr>
<tr>
<td>8</td>
<td>⊤</td>
<td>[(14, ⊥)]</td>
<td>cpop 14</td>
<td>⊥</td>
<td>[]</td>
</tr>
<tr>
<td>9</td>
<td>⊤</td>
<td>[(14, ⊥)]</td>
<td>load \text{contacts}</td>
<td>⊤</td>
<td>[(14, ⊥)]</td>
</tr>
<tr>
<td>10</td>
<td>⊤</td>
<td>[(14, ⊥)]</td>
<td>getf list</td>
<td>⊤</td>
<td>[(14, ⊥)]</td>
</tr>
<tr>
<td>11</td>
<td>⊤</td>
<td>[(14, ⊥)]</td>
<td>load \text{server}</td>
<td>⊤</td>
<td>[(14, ⊥)]</td>
</tr>
<tr>
<td>12</td>
<td>⊤</td>
<td>[(14, ⊥)]</td>
<td>putf \text{buf}</td>
<td>⊤</td>
<td>[(14, ⊥)]</td>
</tr>
<tr>
<td>13</td>
<td>⊤</td>
<td>[(14, ⊥)]</td>
<td>cpop \text{buf}</td>
<td>⊥</td>
<td>[]</td>
</tr>
<tr>
<td>14</td>
<td>⊥</td>
<td>[]</td>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

pseudo-instructions: inserted by compiler; verified by type system
Confluence point stack

if \((\text{contacts} \cdot f_\delta \sqsubseteq \text{server} \cdot f_\delta) \top\) then \(\text{server} \cdot \text{buf} := \text{contacts} \cdot \text{list}\); else skip;

<table>
<thead>
<tr>
<th>(i)</th>
<th>(pc)</th>
<th>(\Delta)</th>
<th>\text{instruction}</th>
<th>(pc')</th>
<th>(\Delta')</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(\bot)</td>
<td>[]</td>
<td>\text{cpush 14}</td>
<td>(\bot)</td>
<td>[(14, (\bot))]</td>
</tr>
<tr>
<td>2</td>
<td>(\bot)</td>
<td>[(14, (\bot))]</td>
<td>\text{load contacts}</td>
<td>(\bot)</td>
<td>[(14, (\bot))]</td>
</tr>
</tbody>
</table>

pseudo-instruction indicating begin of branching statement

\textbf{semantics}: no effect

\textbf{type system}: push given confluence point and current \(pc\) on stack

<table>
<thead>
<tr>
<th>(i)</th>
<th>(pc)</th>
<th>(\Delta)</th>
<th>\text{instruction}</th>
<th>(pc')</th>
<th>(\Delta')</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>(\bot)</td>
<td>[(14, (\bot))]</td>
<td>\text{prim }\sqsubseteq\</td>
<td>(\bot)</td>
<td>[(14, (\bot))]</td>
</tr>
<tr>
<td>7</td>
<td>(\bot)</td>
<td>[(14, (\bot))]</td>
<td>\text{bnz }9</td>
<td>(\top)</td>
<td>[(14, (\bot))]</td>
</tr>
<tr>
<td>8</td>
<td>(\top)</td>
<td>[(14, (\bot))]</td>
<td>\text{cpop }14</td>
<td>(\bot)</td>
<td>[]</td>
</tr>
<tr>
<td>9</td>
<td>(\top)</td>
<td>[(14, (\bot))]</td>
<td>\text{load contacts}</td>
<td>(\top)</td>
<td>[(14, (\bot))]</td>
</tr>
<tr>
<td>10</td>
<td>(\top)</td>
<td>[(14, (\bot))]</td>
<td>\text{getf }\text{list}</td>
<td>(\top)</td>
<td>[(14, (\bot))]</td>
</tr>
<tr>
<td>11</td>
<td>(\top)</td>
<td>[(14, (\bot))]</td>
<td>\text{load server}</td>
<td>(\top)</td>
<td>[(14, (\bot))]</td>
</tr>
<tr>
<td>12</td>
<td>(\top)</td>
<td>[(14, (\bot))]</td>
<td>\text{putf }\text{buf}</td>
<td>(\top)</td>
<td>[(14, (\bot))]</td>
</tr>
<tr>
<td>13</td>
<td>(\top)</td>
<td>[(14, (\bot))]</td>
<td>\text{cpop }14</td>
<td>(\bot)</td>
<td>[]</td>
</tr>
<tr>
<td>14</td>
<td>(\bot)</td>
<td>[]</td>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Confluence point stack

\[
\textbf{if} \ (\text{contacts}.f_{\delta} \sqsubseteq \text{server}.f_{\delta}) \top \ \textbf{then} \ \text{server}.\text{buf} := \text{contacts}.\text{list}; \ \textbf{else} \ \text{skip};
\]

<table>
<thead>
<tr>
<th>(i)</th>
<th>(pc)</th>
<th>(\Delta)</th>
<th>\textit{instruction}</th>
<th>(pc')</th>
<th>(\Delta')</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(\bot)</td>
<td>[]</td>
<td>cpush 14</td>
<td>(\bot)</td>
<td>([(14, \bot)])</td>
</tr>
<tr>
<td>2</td>
<td>(\bot)</td>
<td>([(14, \bot)])</td>
<td>load contacts</td>
<td>(\bot)</td>
<td>([(14, \bot)])</td>
</tr>
<tr>
<td>3</td>
<td>(\bot)</td>
<td>([(14, \bot)])</td>
<td>getf (f_{\delta})</td>
<td>(\bot)</td>
<td>([(14, \bot)])</td>
</tr>
<tr>
<td>4</td>
<td>(\bot)</td>
<td>([(14, \bot)])</td>
<td>load server</td>
<td>(\bot)</td>
<td>([(14, \bot)])</td>
</tr>
<tr>
<td>5</td>
<td>(\bot)</td>
<td>([(14, \bot)])</td>
<td>getf (f_{\delta})</td>
<td>(\bot)</td>
<td>([(14, \bot)])</td>
</tr>
<tr>
<td>6</td>
<td>(\bot)</td>
<td>([(14, \bot)])</td>
<td>prim (\sqsubseteq)</td>
<td>(\bot)</td>
<td>([(14, \bot)])</td>
</tr>
<tr>
<td>7</td>
<td>(\bot)</td>
<td>([(14, \bot)])</td>
<td>bnz 9</td>
<td>(\top)</td>
<td>([(14, \bot)])</td>
</tr>
<tr>
<td>8</td>
<td>(\top)</td>
<td>([(14, \bot)])</td>
<td>cpop 14</td>
<td>(\bot)</td>
<td>[]</td>
</tr>
<tr>
<td>9</td>
<td>(\top)</td>
<td>([(14, \bot)])</td>
<td>load contacts</td>
<td>(\top)</td>
<td>([(14, \bot)])</td>
</tr>
<tr>
<td>10</td>
<td>(\top)</td>
<td>([(14, \bot)])</td>
<td>getf list</td>
<td>(\top)</td>
<td>([(14, \bot)])</td>
</tr>
<tr>
<td>11</td>
<td>(\top)</td>
<td>([(14, \bot)])</td>
<td>load server</td>
<td>(\top)</td>
<td>([(14, \bot)])</td>
</tr>
<tr>
<td>12</td>
<td>(\top)</td>
<td>([(14, \bot)])</td>
<td>putf buf</td>
<td>(\top)</td>
<td>([(14, \bot)])</td>
</tr>
<tr>
<td>13</td>
<td>(\top)</td>
<td>([(14, \bot)])</td>
<td>cpop buf</td>
<td>(\bot)</td>
<td>[]</td>
</tr>
<tr>
<td>14</td>
<td>(\bot)</td>
<td>[]</td>
<td>. . .</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

pseudo-instructions: inserted by compiler; verified by type system
Confluence point stack

\[
\text{if } (\text{contacts}.f_\delta \sqsubseteq \text{server}.f_\delta) \top \text{ then server.buf := contacts.list; else skip;}
\]

<table>
<thead>
<tr>
<th>( i )</th>
<th>( pc )</th>
<th>( \Delta )</th>
<th>instruction</th>
<th>( pc' )</th>
<th>( \Delta' )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>\bot</td>
<td>[]</td>
<td>cpush 14</td>
<td>\bot</td>
<td>[(14, \bot)]</td>
</tr>
<tr>
<td>2</td>
<td>\bot</td>
<td>[(14, \bot)]</td>
<td>load contacts</td>
<td>\bot</td>
<td>[(14, \bot)]</td>
</tr>
<tr>
<td>3</td>
<td>\bot</td>
<td>[(14, \bot)]</td>
<td>getf ( f_\delta )</td>
<td>\bot</td>
<td>[(14, \bot)]</td>
</tr>
<tr>
<td>4</td>
<td>\bot</td>
<td>[(14, \bot)]</td>
<td>load server</td>
<td>\bot</td>
<td>[(14, \bot)]</td>
</tr>
<tr>
<td>5</td>
<td>\bot</td>
<td>[(14, \bot)]</td>
<td>getf ( f_\delta )</td>
<td>\bot</td>
<td>[(14, \bot)]</td>
</tr>
<tr>
<td>6</td>
<td>\bot</td>
<td>[(14, \bot)]</td>
<td>prim \sqsubseteq</td>
<td>\bot</td>
<td>[(14, \bot)]</td>
</tr>
<tr>
<td>7</td>
<td>\bot</td>
<td>[(14, \bot)]</td>
<td>\textbf{bnz 9}</td>
<td>\top</td>
<td>[(14, \bot)]</td>
</tr>
<tr>
<td>8</td>
<td>\top</td>
<td>[(14, \bot)]</td>
<td>cpush 14</td>
<td></td>
<td>[]</td>
</tr>
<tr>
<td>9</td>
<td>\top</td>
<td>[(14, \bot)]</td>
<td>\textbf{bnz 9}</td>
<td>\top</td>
<td>[(14, \bot)]</td>
</tr>
<tr>
<td>10</td>
<td>\top</td>
<td>[(14, \bot)]</td>
<td>getf list</td>
<td>\top</td>
<td>[(14, \bot)]</td>
</tr>
<tr>
<td>11</td>
<td>\top</td>
<td>[(14, \bot)]</td>
<td>load server</td>
<td>\top</td>
<td>[(14, \bot)]</td>
</tr>
<tr>
<td>12</td>
<td>\top</td>
<td>[(14, \bot)]</td>
<td>putf buf</td>
<td>\top</td>
<td>[(14, \bot)]</td>
</tr>
<tr>
<td>13</td>
<td>\top</td>
<td>[(14, \bot)]</td>
<td>cpop 14</td>
<td>\bot</td>
<td>[]</td>
</tr>
<tr>
<td>14</td>
<td>\bot</td>
<td>[]</td>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

pseudo-instructions: inserted by compiler; verified by type system
Confluence point stack

\[
\text{if } (\text{contacts}.f_\delta \sqsubseteq \text{server}.f_\delta) \top \text{ then server.buf := contacts.list; else skip;}
\]

<table>
<thead>
<tr>
<th>(i)</th>
<th>(pc)</th>
<th>(\Delta)</th>
<th>instruction</th>
<th>(pc')</th>
<th>(\Delta')</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(\bot)</td>
<td>([])</td>
<td>cpush 14</td>
<td>(\bot)</td>
<td>([(14, \bot)])</td>
</tr>
<tr>
<td>2</td>
<td>(\bot)</td>
<td>([(14, \bot)])</td>
<td>load contacts</td>
<td>(\bot)</td>
<td>([(14, \bot)])</td>
</tr>
<tr>
<td>3</td>
<td>(\bot)</td>
<td>([(14, \bot)])</td>
<td>getf (f_\delta)</td>
<td>(\bot)</td>
<td>([(14, \bot)])</td>
</tr>
<tr>
<td>4</td>
<td>(\bot)</td>
<td>([(14, \bot)])</td>
<td>load server</td>
<td>(\bot)</td>
<td>([(14, \bot)])</td>
</tr>
<tr>
<td>5</td>
<td>(\bot)</td>
<td>([(14, \bot)])</td>
<td>getf (f_\delta)</td>
<td>(\bot)</td>
<td>([(14, \bot)])</td>
</tr>
<tr>
<td>6</td>
<td>(\bot)</td>
<td>([(14, \bot)])</td>
<td>prim (\sqsubseteq)</td>
<td>(\bot)</td>
<td>([(14, \bot)])</td>
</tr>
<tr>
<td>7</td>
<td>(\bot)</td>
<td>([(14, \bot)])</td>
<td>bnz 9</td>
<td>(\top)</td>
<td>([(14, \bot)])</td>
</tr>
<tr>
<td>8</td>
<td>(\top)</td>
<td>([(14, \bot)])</td>
<td>cpop 14</td>
<td>(\bot)</td>
<td>([])</td>
</tr>
<tr>
<td>9</td>
<td>(\top)</td>
<td>([(14, \bot)])</td>
<td>load contacts</td>
<td>(\top)</td>
<td>([(14, \bot)])</td>
</tr>
<tr>
<td>10</td>
<td>(\top)</td>
<td>([(14, \bot)])</td>
<td>getf list</td>
<td>(\top)</td>
<td>([(14, \bot)])</td>
</tr>
<tr>
<td>11</td>
<td>(\top)</td>
<td>([(14, \bot)])</td>
<td>load server</td>
<td>(\top)</td>
<td>([(14, \bot)])</td>
</tr>
<tr>
<td>12</td>
<td>(\top)</td>
<td>([(14, \bot)])</td>
<td>putf buf</td>
<td>(\top)</td>
<td>([(14, \bot)])</td>
</tr>
<tr>
<td>13</td>
<td>(\top)</td>
<td>([(14, \bot)])</td>
<td>cpop buf</td>
<td>(\bot)</td>
<td>([])</td>
</tr>
<tr>
<td>14</td>
<td>(\bot)</td>
<td>([])</td>
<td>(\ldots)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Confluence point stack

\[
\text{if } (\text{contacts}.f_\delta \sqsubseteq \text{server}.f_\delta) \top \text{ then server.buf := contacts.list; else skip;}
\]

<table>
<thead>
<tr>
<th>(i)</th>
<th>(pc)</th>
<th>(\Delta)</th>
<th>instruction</th>
<th>(pc')</th>
<th>(\Delta')</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(\bot)</td>
<td>([],)</td>
<td>cpush 14</td>
<td>(\bot)</td>
<td>([(14, \bot)])</td>
</tr>
<tr>
<td>2</td>
<td>(\bot)</td>
<td>([(14, \bot)])</td>
<td>load contacts</td>
<td>(\bot)</td>
<td>([(14, \bot)])</td>
</tr>
<tr>
<td>3</td>
<td>(\bot)</td>
<td>([(14, \bot)])</td>
<td>getf (f_\delta)</td>
<td>(\bot)</td>
<td>([(14, \bot)])</td>
</tr>
<tr>
<td>4</td>
<td>(\bot)</td>
<td>([(14, \bot)])</td>
<td>load server</td>
<td>(\bot)</td>
<td>([(14, \bot)])</td>
</tr>
<tr>
<td>5</td>
<td>(\bot)</td>
<td>([(14, \bot)])</td>
<td>getf (f_\delta)</td>
<td>(\bot)</td>
<td>([(14, \bot)])</td>
</tr>
<tr>
<td>6</td>
<td>(\bot)</td>
<td>([(14, \bot)])</td>
<td>prim (\sqsubseteq)</td>
<td>(\bot)</td>
<td>([(14, \bot)])</td>
</tr>
<tr>
<td>7</td>
<td>(\bot)</td>
<td>([(14, \bot)])</td>
<td>bnz 9</td>
<td>(\top)</td>
<td>([(14, \bot)])</td>
</tr>
<tr>
<td>8</td>
<td>(\top)</td>
<td>([(14, \bot)])</td>
<td>cpop 14</td>
<td>(\bot)</td>
<td>([],)</td>
</tr>
<tr>
<td>9</td>
<td>(\top)</td>
<td>([(14, \bot)])</td>
<td>load contacts</td>
<td>(\top)</td>
<td>([(14, \bot)])</td>
</tr>
<tr>
<td>10</td>
<td>(\top)</td>
<td>([(14, \bot)])</td>
<td>getf list</td>
<td>(\top)</td>
<td>([(14, \bot)])</td>
</tr>
<tr>
<td>11</td>
<td>(\top)</td>
<td>([(14, \bot)])</td>
<td>load server</td>
<td>(\top)</td>
<td>([(14, \bot)])</td>
</tr>
<tr>
<td>12</td>
<td>(\top)</td>
<td>([(14, \bot)])</td>
<td>putf buf</td>
<td>(\top)</td>
<td>([(14, \bot)])</td>
</tr>
<tr>
<td>13</td>
<td>(\top)</td>
<td>([(14, \bot)])</td>
<td>cpop 14</td>
<td>(\bot)</td>
<td>([],)</td>
</tr>
<tr>
<td>14</td>
<td>(\bot)</td>
<td>([],)</td>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

pseudo-instruction indicating end of branch
semantics: jump to given address
type system: pop confluence point and restore \(pc\)

pseudo-instructions: inserted by compiler; verified by type system