NOKIA Bell Labs

Plugging Information Leaks Introduced by Compiler Transformations

Kedar Namjoshi (Bell Labs) + Chaoqiang Deng (NYU)

Dagstuhl

May 2018

 © 2017 Nokia Public

“To talk of many things…”

1. **Leaky Transformations**
   Dead store elimination (DSE), SSA conversion

2. **Translation Validation for Leak-Freedom?**
   With compiler-generated witnesses

3. **Plugging Leaks**
   Designing leak-free forms of DSE and SSA

4. **(Many) (Open?) Questions**
   Mostly exposing my ignorance of this topic ☺

“Leak this against my wishes.”

[New Yorker, 2003]

© 2018 Nokia

Leaky Transformations
Example: Dead Store Elimination (DSE)

```
x := password();
check(x);
x := 0; // clear secret
  .
  .
  .
```


```
x := password();
check(x);
skip;
  .
  .
  .
```


© 2018 Nokia

Leaky Transformations

Example: Static Single Assignment (SSA) and Live Range Splitting

```
x := password();
check(x);
x := 0; // clear secret
  .
  .
  .
```


```
x1 := password();
check(x1);
x2 := 0;
  .
  .
  .
```


© 2018 Nokia

When is a Transformation Leaky?

Leaky Program

A deterministic program P with secret input H and non-secret input L

P is **leaky** if there are input values (H=a, L=c) and (H=b, L=c) such that on the corresponding executions, either

1. the sequence of output values differ, or
2. both executions terminate but in states with different non-secret values

We'll call (H=a, H=b, L=c) a **leaky triple** for P.

Leaky Transformation

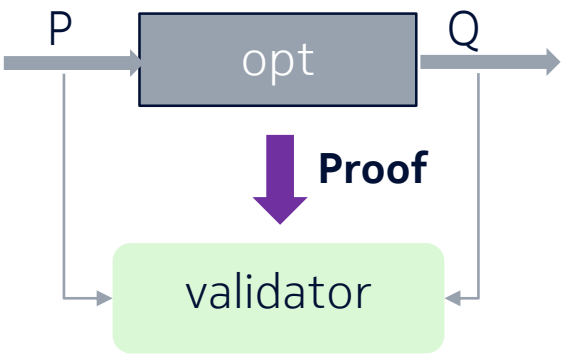
A transformation from program P to program Q (with input variables unchanged) is **leak-free** if every leaky triple for Q is also a leaky triple for P.

The DSE transform is leaky as the triple (pwd=0, pwd=1, -) is leaky for Q but not for P.

Leak-freedom **does not** assert that P (or Q) have no leaks – only that no *new* leaks are introduced by the transformation.

© 2018 Nokia Public

Aside: Translation Validation with Proof Witnesses



With proof witnesses, validation amounts to proof checking.

Theorem: If the proof object R is a refinement relation from Q to P, the transformation is correct.

Questions: Could one similarly validate leak-freedom?
What is the corresponding proof object?

"Credible Compilation"
[Rinard-Marinov, 1999]

"Witnessing" [N.-Zuck, 2013]

© 2018 Nokia

Translation Validation is Undecidable for Leak-Freedom
(even restricted to DSE)

```
x := password();
check(x);
M;
x := 0;
```

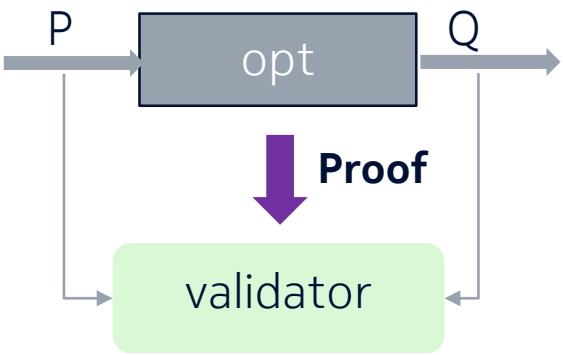

```
x := password();
check(x);
M;
skip;
```

- If M terminates, this transformation is leaky
- If M does not terminate, the transformation is leak-free

Note: validating correctness is easy for DSE

© 2018 Nokia Public

Refinement Witnesses Ensuring Leak-Freedom



Theorem: Given a proof object R, if

- R is a refinement, and
- R(t,s) and R(t',s') implies that $t \sim_Q t'$ iff $s \sim_P s'$,

then the transformation is leak-free.

$t \sim_P t'$ if t and t' are states of P with identical low values

[see also de Amorim et al, POPL 2014]

Several common optimizations meet this condition.

© 2018 Nokia

A Leak-Free DSE Optimization

Idea: Combine taint and control-flow information to determine stores that can be removed

```
{x:U}
x := password();
{x:T}
check(x);
{x:T}
x := 0;
{x:U}
x := 5;
{x:U}
```


```
{x:U}
x := password();
{x:T}
check(x);
{x:T}
skip;
{x:T}
x := 5;
{x:U}
```

[Deng-N., SAS 2016]

© 2018 Nokia

Heuristics, in a bit more detail…

Consider a dead store to variable x. It may be removed if:

1. This store is post-dominated by some other store to x

   Justification: any leak through x must arise from the dominating stores

2. Variable x is untainted before this store and untainted at the exit from the program

   Justification: the taint proof is unchanged, so a leak cannot arise from x; other flows are preserved

3. Variable x is untainted before this store, other stores to x are unreachable, and this store post-dominates the entry node

   Justification: the taint proof is unchanged, so a leak cannot arise from x; other flows are preserved

4. … *your heuristic here* …

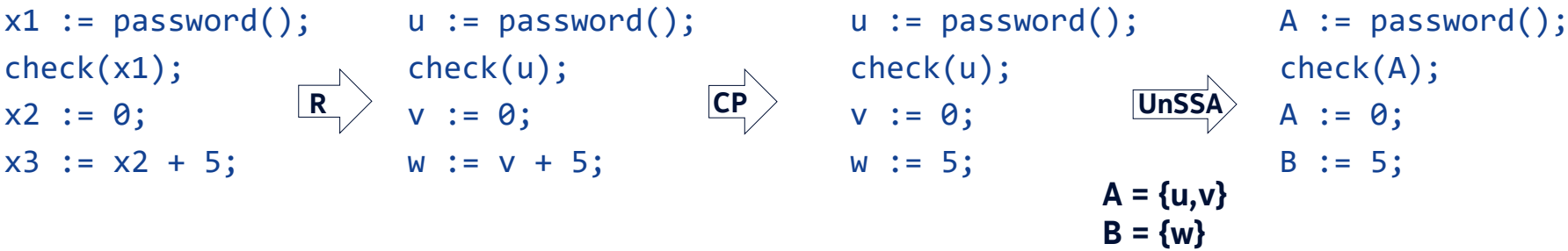
© 2018 Nokia

A Leak-Free SSA Conversion

Idea: regroup SSA variables, minimizing grouping using taint and control-flow information.

E.g., the group {`x1`,`x2`} is treated as a single fresh variable, say `z`.

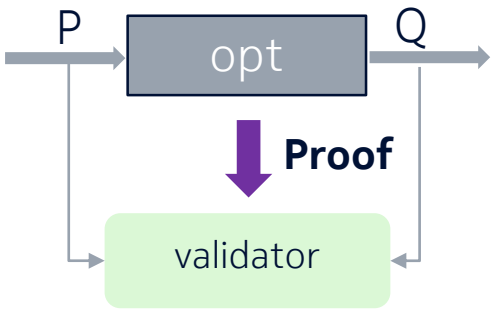
Regrouping can be done only **after** completing all SSA-based transformations.
Track SSA-induced leaks using the refinement proofs generated by the transformations.



[Deng-N., SAS 2017]

© 2018 Nokia

(Open?) Questions



- Proof methods that preserve a large class of security properties? (Back-translation? Simpler sufficient conditions?)
- Are these proof methods usable for translation validation?
- What is the witness format?
- What is the complexity of witness generation and checking?
- What is the analogue of (static) taint analysis for other security properties?

© 2018 Nokia

Formal Acknowledgements

This work was supported, in part, by DARPA under agreement number FA8750-12-C-0166. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

This material is based upon work supported by the National Science Foundation under Grant No. (NSF CCF-1563393). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

 © 2018 Nokia

NOKIA