RIFLE

An Architectural Framework for User-Centric Information-Flow Security

Vachharajani, N. and Bridges, M.J. and Chang, J. and Rangan, R. and Ottoni, G. and Blome, J.A. and Reis, G.A. and Vachharajani, M. and August, D.I.

Information Flow Analysis Workshop / Software Safety & Security, (DD2460)

- Oleksandr Bodriagov
- Benjamin Greschbach
- Guillermo Rodríguez Cano
- Oliver Schwarz
- Meidi Tõnisson

Overview

1. Information Flow Security
2. Static Analysis vs. Dynamic Analysis
3. RIFLE approach
    1. Security Registers
    2. Binary Translation
    3. Implicit Flows and Loops
4. Evaluation and Performance
5. Comparison and Limitations

Information flow: some reasons

- Trusting programs is difficult (and unfair)
  - No guarantees on data usage
- Who should decide how the data shall be accessed?
  - The user or the analyst/programmer?
- An example...
  - Windows XP activation: submit signature or uninstall
- Therefore...
  Why not granting access to data but preventing its leakage?
  - This is what IFS aims for

Information flow: some solutions

- What if we label the information for these flow policies?
  - Label everything!
- Getting back to the previous example...
  - By labeling Alice can know if Windows XP access more than needed...
- These flows can be detected, and prevented!
  - Statically (compile time) or...
  - Dynamically (run time)

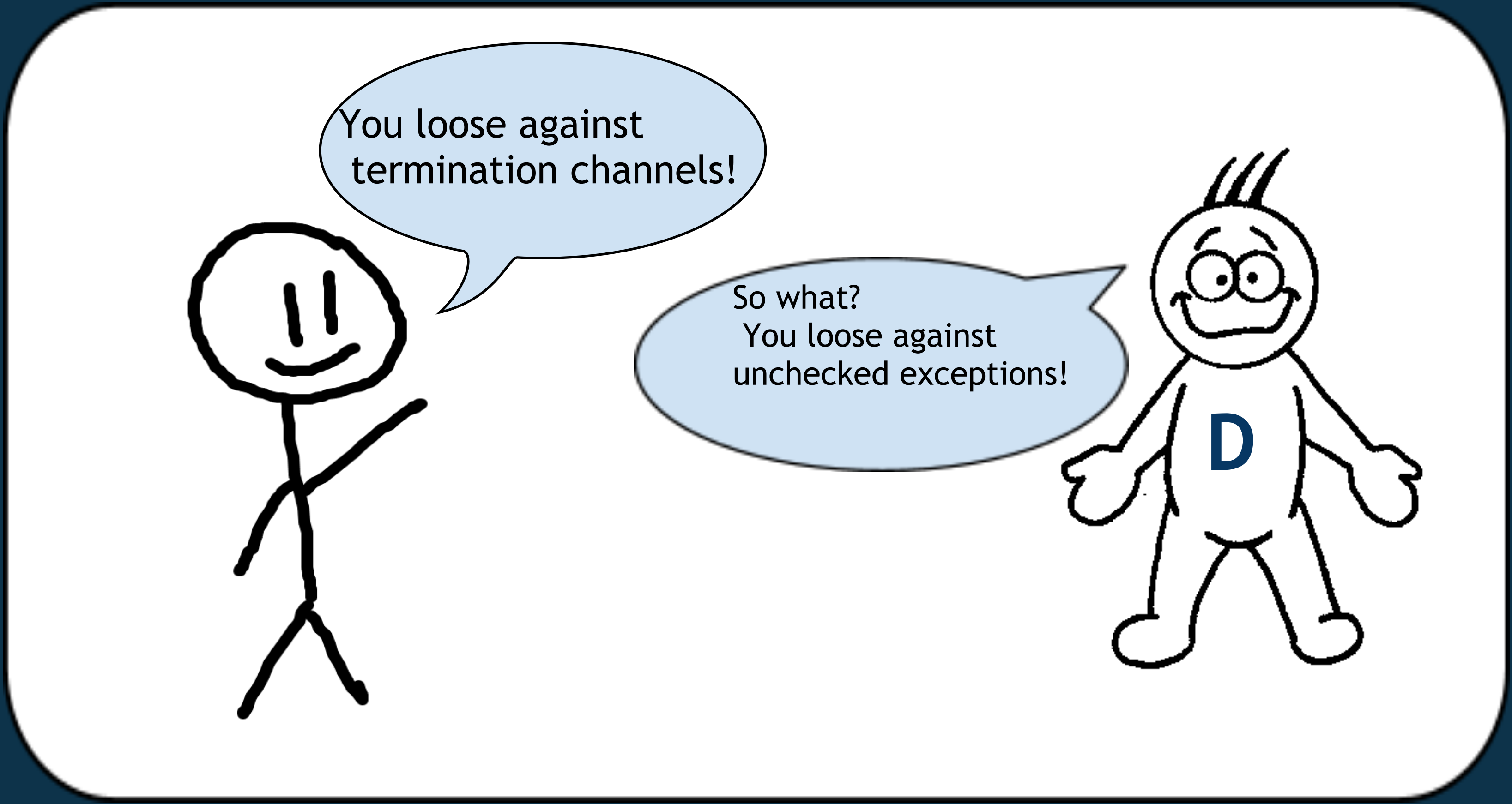
Static analysis

- Main focus for research in the area
  - Information leaks are verified at compilation time
- Provide security to programmer but not to the user
  - Programmer decides policies (legal/illegal flows)
  - Too conservative or too lax approach
- Requires specific languages
  - Only strong type languages can be extended
  - For instance, C/C++ could not be checked

Dynamic analysis

- Very few run-time options have been studied as they are believed to be less secure
- Tracking mechanisms at program run time
  - Labels are read from input and propagated during execution up to storage location
  - Enforcing security depends on checking whether it is allowed to write data on an output channel
- However, the user is in control of the information flows
  - In the end, it is the user who decides not the programmer (user-centric approach)

Who is more secure?



Termination channels

```
secret = ...;
a = 0;

for(i=min; i<max; i++){
   if(i == secret)
      low = high;
   pr
}
```

Have to terminate

`low = 30/a;`

exceptions:

dividing by zero
dereferencing null
array out of bounds
exhausting resource
...

how to discover those without being too restrictive?

Implicit flows

a ★
b
c ★

```
| a = false;
| b = false;
| c = false;
| if(!a)
|    c = true;
| if(!c)
     b = true;
| print b;
```

`b == a`

Implicit flows

a ★
b
c

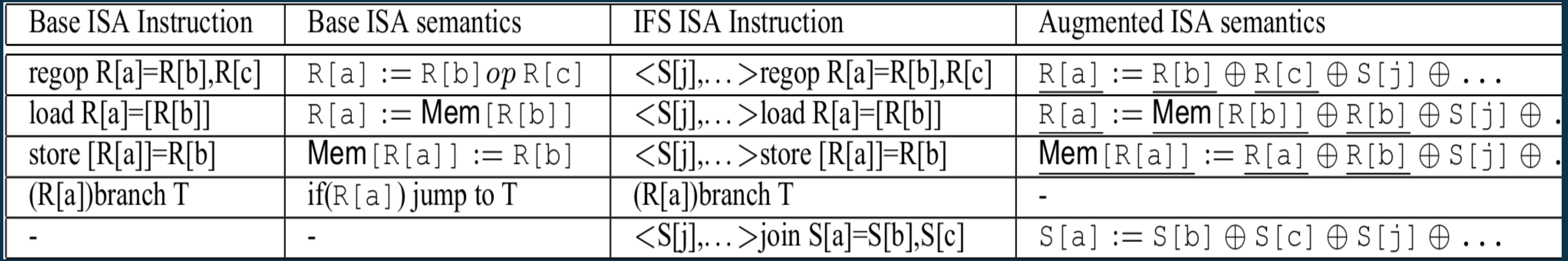
```
| a = true;
| b = false;
| c = false;
| if(!a)
     c = true;
| if(!c)
|    b = true;
| print b;
```

`b == a`

RIFLE

- translates ordinary binary code to a binary for processors that support IFS
- translates all implicit flows to explicit
- OS is augmented (registry, memory, IFS instruction set) to use labels
- OS does enforcement

RIFLE: binary translation



Base ISA Instruction	Base ISA semantics	IFS ISA Instruction	Augmented ISA semantics
regop R[a]=R[b],R[c]	$R[a] := R[b]\ op\ R[c]$	<S[j],…>regop R[a]=R[b],R[c]	$\underline{R[a]} := \underline{R[b]} \oplus \underline{R[c]} \oplus S[j] \oplus \ldots$
load R[a]=[R[b]]	$R[a] := \mathrm{Mem}[R[b]]$	<S[j],…>load R[a]=[R[b]]	$\underline{R[a]} := \underline{\mathrm{Mem}[R[b]]} \oplus \underline{R[b]} \oplus S[j] \oplus .$
store [R[a]]=R[b]	$\mathrm{Mem}[R[a]] := R[b]$	<S[j],…>store [R[a]]=R[b]	$\underline{\mathrm{Mem}[R[a]]} := \underline{R[a]} \oplus \underline{R[b]} \oplus S[j] \oplus .$
(R[a])branch T	if($R[a]$) jump to T	(R[a])branch T	-
-	-	<S[j],…>join S[a]=S[b],S[c]	$S[a] := S[b] \oplus S[c] \oplus S[j] \oplus \ldots$

R[i] -general register, S[j] - security register (stores a label), Mem[a] - memory location at address a, $\underline{X}$ - label of data element x.

- augmented state contains a label(ex. label $\underline{R[a]}$)
- one additional instruction = join of two labels
- semantics will be identical after translation
- each branch instruction is replaced:

(R[a])branch T	join S[c] = labelof (R[a]), $\underline{\perp}$ (R[a]) branch T

Handling implicit flows

- add appropriate label regardless of path taken
- append security register to list of security operands on instructions that potentially use control dependent values

```
// Assume R[1] contains a
// b will be stored in R[2]
// c will be stored in R[3]
     mov R[2] = 0
     mov R[3] = 0
     (R[1]) branch .L1
     mov R[3] = 1
.L1: (R[3]) branch .L2
     mov R[2] = 1
.L2: store [R[5]] = R[2]
```

```
// Assume R[1] contains a
// b will be stored in R[2]
// c will be stored in R[3]
             mov R[2] = 0
             mov R[3] = 0
             mov S[1] = labelof(R[1])
             (R[1]) branch .L1
      <S[1]> mov R[3] = 1
.L1:  <S[1]> mov S[3] = labelof(R[3])
             (R[3]) branch .L2
      <S[3]> mov R[2] = 1
.L2:  <S[3]> store [R[5]] = R[2]
```

Handling loops

- security registers may potentially be used after back edge is crossed
- values computed under earlier conditions might become accessible under the new label
- information leak!
- easily avoided by defining the security operand before each branch as the join of the branch predicate and the previous value of the security operand

join S[c] = R[a], S[c]

Evaluation

- **wc** (unix word count tool)
  Input: different files with different labels
  Output: according file labels, join of labels for summary output

- **PGP** (encryption tool)
  Created pair of key rings, unique label per key and input file.
  Problem: Scan over all keys before the matching one. Fixed by labeling all keys equally.
  Expected behavior: output labeled with join of labels from input file, public key (encryption) and private key (signature)

- **thttpd** (tiny webserver)
  Two files, each protected by a password
  Unauthorized request: output labeled with request+document+usernames
  Authorized request: +password (misleading, only 1 bit: correct or not)

Performance

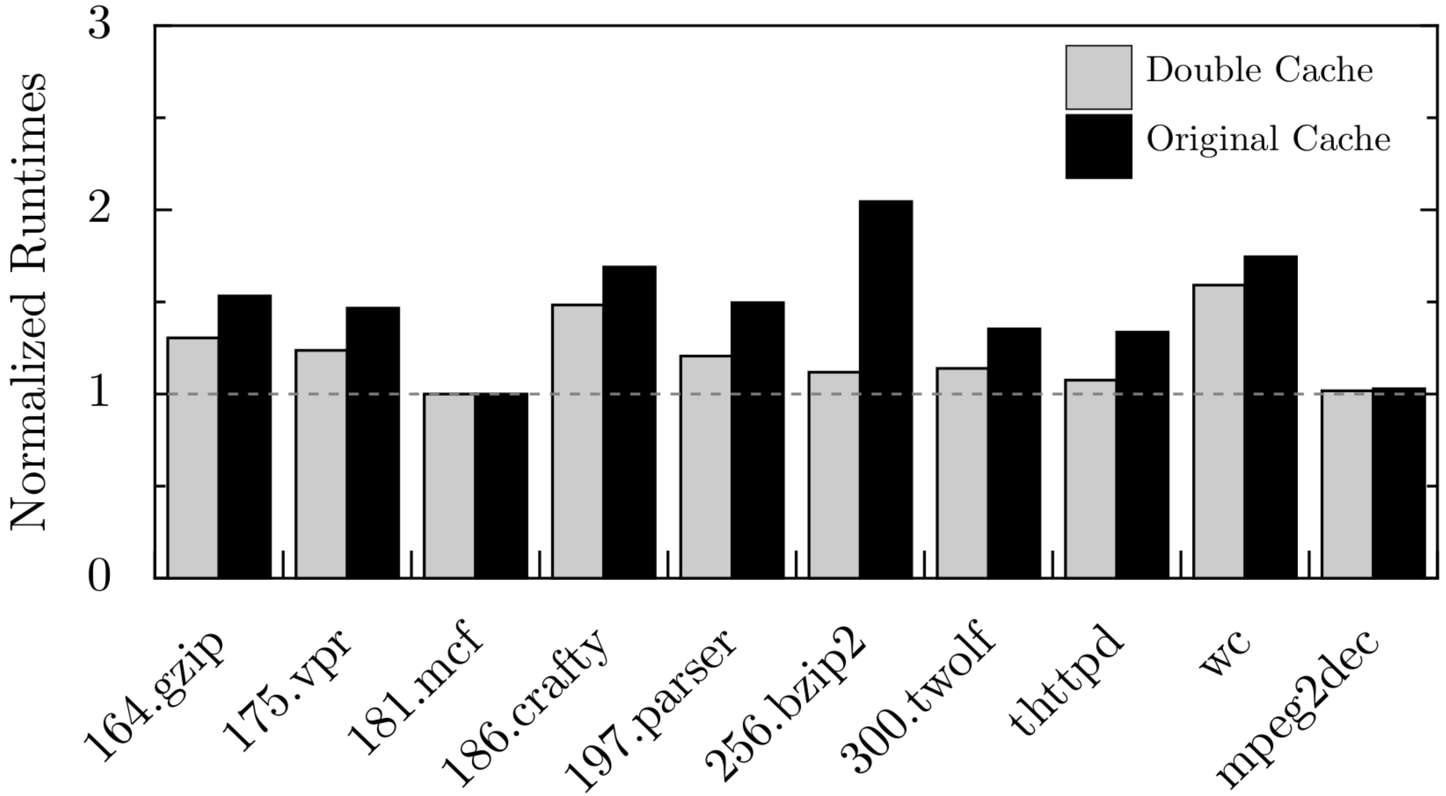


Figure 7. Performance of secured programs

Double Cache: all data caches duplicated to store security labels
Original Cache: data cache partitioned into two equally sized pieces

Comparison with other techniques

	static analysis (e.g. Jif)	**proof-carrying code**	**RIFLE (dynamic)**
verification:	compile time	compile time / pre-run	runtime
source requirements:	source code	source code	binary
policy decisions:	developer	developer	user
trust:	developer	user	user

Limitations of RIFLE

- **Covert channels not detected**
    - Timing based
    - Termination

- **User must be educated**
    - Proper labeling of inputs and outputs
    - Interpreting results (e. g. thttpd problem: password not leaked, only correctness)

- **Assumptions** (realistic?)
    - Hardware support (by virtualization?)
    - Operation system support (suppress illegal outputs)

Conclusion

- RIFLE = run-time analysis of information flow (labels propagate through computation instead of being statically assigned to storage locations)
- dynamic analysis is not less secure than static approach
- RIFLE consists of three parts:
  1. Architecture (security registers and semantics on them)
  2. Binary translation
  3. OS support for enforcing the policies

advantages:

- language independent, no source code required
- user sets policy and does not have to trust the developer

disadvantages:

- runtime performance decrease
- results have to be interpreted by an educated user