Don’t let data Go astray
A Context-Sensitive Taint Analysis for Concurrent Programs in Go

Volker Stolz
Bergen University College, Norway
& University of Oslo, Norway

28th Nordic Workshop on Programming Theory (NWPT’16)
1st November 2016

Supported by the bilateral Norwegian/German project
GoRETech — Go Runtime Enforcement Techniques
& EU COST Action IC1402 “ARVI — Applied Runtime Verification”
Don’t let data Go astray

Violet Ka I Pun       Martin Steffen       Volker Stolz
Anna-Katharina Wickert       Eric Bodden       Michael Eichberg
Motivation

- **Taint analysis**: data flow analysis
- Secure information flow
- Prevent untrusted/sensitive data from reaching sensitive locations
- Examples (the usual suspects):
  - SQL injection (user input flows unfiltered into SQL query)
  - leaks (clear-text password ending up in log/debugging output)
The Go language

- Backed by Google
- imperative (C-programmers should be able to read it)
- object-oriented (maybe...)
- concurrent (goroutines)
- structurally typed
- garbage collected; dynamic race checker
- higher-order functions and closures
What are methods?

- procedures – functions – methods
- methods are “specific” functions
- Example: add1 / add2 not that much different from each other

```go
type Number struct { n int }

func add1 (x Number, y Number)
{
    return x.n+y.n
}
```

```go
func (x Number) add2 (y Number) int
{
    return x.n+y.n
}
```

method ~ function with special first argument

\[ f(o, v) \text{ vs. } o.f(v) \]

- elsewhere often: special keyword for first argument: this (or self)
Higher-order functions

- known from functional languages
- languages with higher-order functions
  - functions as “first-class” data ⇒

```go
func add (x int) (func (y int) int) {
    return func (y int) int {
        return y + x
    }
}
```

\[
add : \text{int} \rightarrow (\text{int} \rightarrow \text{int}) = \lambda x : \text{int} . \lambda y : \text{int} . x + y
\]
Deferred functions

Each function/method can be called:

1. conventionally
2. deferred
3. asynchronously (see later)
   - Also in Apple’s Swift language

```python
func main() {
    defer fmt.Println("1")
    fmt.Println("2")
}
```

- Deferred call (guaranteed to be) executed when the surrounding function body returns
- Eval’d for side-effect only, returned value irrelevant
- Deferred calls can be nested, too
Go’s concurrency mantra

“Don’t communicate by sharing memory, share memory by communicating!”

Go concurrency

goroutines + channels

- claimed to be “easy”
- first-class, typed channels
Channels

- “named pipes”
- FIFO, bounded, non-lossy communication
- crucial data type with synchronization power
- taking a back-seat:
  - locks
  - mutexes
  - monitors
  - semaphores...
- channels: first-class data
  - channels can send (references to) channels
  - inspired by CSP (and CCS, and, actually $\pi$)
- directed channels
Channels

Simple example: synchronized channel for strings

```go
package main
import "fmt"

func main () {
  messages := make(chan string, 0)

  go func () {
    messages <- "ping"
  } ()

  msg := <- messages
  fmt.Println(msg)
}
```
Back to Taint Analysis...

- Identifies flows of private information to untrusted places

- Terminology:
  - Source (produce tainted data)
  - Sink (consumer of data)
  - Flow (from source to sink)

- Both given as user-defined sets of methods

- (Q: How to untaint data? Sanitizers, future work)
Abstract syntax captures essence of Go

Gory details in SSA representation in implementation

\[
\begin{align*}
  s & ::= \quad x := e \mid x.f := e \mid \text{if } v \text{ then } s \text{ else } s \mid \text{defer}((\lambda x.s)v) \\
      & \quad \mid \text{go } s \mid x \leftarrow y \mid x \rightarrow y \mid \text{return } v \mid s ; s \\
  e & ::= \quad v \mid v v \mid \text{makeChan} \\
  v & ::= \quad x \mid x.f \mid () \mid \text{true} \mid \text{false} \mid \lambda x.s
\end{align*}
\]
Lattice for Taint Analysis

Simple lattice:

<table>
<thead>
<tr>
<th>(\mathcal{U})</th>
<th>(\bot)</th>
<th>1</th>
<th>0</th>
<th>(\top)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\bot)</td>
<td>(\bot)</td>
<td>1</td>
<td>0</td>
<td>(\top)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>(\top)</td>
<td>(\top)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>(\top)</td>
<td>0</td>
<td>(\top)</td>
</tr>
<tr>
<td>(\top)</td>
<td>(\top)</td>
<td>(\top)</td>
<td>(\top)</td>
<td>(\top)</td>
</tr>
</tbody>
</table>

\[ \text{Volker Stolz} \]

Don’t let data \text{ Go astray}
Standing on the Shoulders...

- Existing Go compiler infrastructure:
  - SSA representation
    https://godoc.org/golang.org/x/tools/go/ssa
  - Basic blocks / call-graph
    https://godoc.org/golang.org/x/tools/go/callgraph
  - Points-to analysis à la Andersen
    https://godoc.org/golang.org/x/tools/go/pointer

- Interprocedural, context-sensitive analysis:
  Padhye / Khedker, SOAP@PLDI 2013
  - standard worklist algorithm w/ calling context & value
Existing Go compiler infrastructure:

- SSA representation
  https://godoc.org/golang.org/x/tools/go/ssa
- Basic blocks / call-graph
  https://godoc.org/golang.org/x/tools/go/callgraph
- Points-to analysis à la Andersen
  https://godoc.org/golang.org/x/tools/go/pointer

Interprocedural, context-sensitive analysis:
Padhye / Khedker, SOAP@PLDI 2013

- standard worklist algorithm w/ calling context & value

... or their toes?
Our analysis – example

```
main()

n1  a := "Hello World"

n3  fmt.Print(b) // sink

f, _ := os.OpenFile("./pw.txt")

c1  b := g(a)

n4  b := make([]byte, 8)
r, _ = f.Read(b)

c2  s, n := h(f)

c3  s, n := h(f)

c4  t := g(s)

n6  fmt.Print(s) // sink

for n > 0

n7  g(a string) (b string)

n8  fmt.Print(t) // sink

n9  exit

h(f *os.File) (c string, r int)

n5  c = string(b[:])
    return

```
Our analysis – example

```
main()

n1  a := "Hello World"

c1   b := g(a)

n3   fmt.Print(b)  // sink
   f, _ := os.OpenFile("./pw.txt")

c2   s, n := h(f)

n6   fmt.Print(s)  // sink
       for n > 0

c3   s, n := h(f)

c4   t := g(s)

n8   fmt.Print(t)  // sink

n2 | n7

exit

h(f *os.File) (c string, r int)

n4   b := make([]byte, 8)
       r, _ = f.Read(b)

n5   c = string(b[:])
       return

Tainted

Untainted

g(a string) (b string)

n2 | n7

b = a
       return

Volker Stolz

Don’t let data Go astray

NWPT’16 13 / 20
Our analysis – example

```go
main()
a := "Hello World"
b := g(a)
fmt.Print(b)  // sink
f, _ := os.OpenFile("./pw.txt")
s, n := h(f)
fmt.Print(s)  // sink
for n > 0
s, n = h(f)
t := g(s)
fmt.Print(t)  // sink
exit
```

```
h(f *os.File) (c string, r int)
b := make([]byte, 8)
r, _ = f.Read(b)
c = string(b[:])
return
```

Tainted

```go
g(a string) (b string)
b = a
return
```

Untainted
Q: Can we handle flow through channels in the same framework?

```
x := “Hello World”
ch := make(chan, string)
go f(ch)
sink(x)
x = tainted
ch ← x
```

A: Add feedback from writes to reads

Overapproximation; only feed back taint value of channel along this edge
Channel Handling

Q: Can we handle flow through channels in the same framework?

A: Add feedback from writes to reads

Overapproximation; only feed back tainted value of channel along this edge
Our analysis

“Nielson-style” data-flow analysis (monotone framework)

Analysis info of a node $= \text{transfer function of node applied to (union over all incoming flows)}$

$\text{TA}(\ell) = \Phi(S, N^\ell)$

where

$S = \bigcup \{ \text{TA}(\ell') \mid (\ell', \ell) \in \text{FLOW}^*(P) \}$ and

$N^\ell \in \text{NODES}(P)$. 
Assignments, from/to struct members, calls:

$$\Phi(S, [x := e]^\ell) = \phi(S, x, e, \ell)$$
Our analysis

Assignments, from/to struct members, calls:

\[ \Phi(S, [x := e]^{\ell}) = \phi(S, x, e, \ell) \]

\[ \phi(S, x, y.f, \ell) = S[x \mapsto \bigcup \{ \text{TA}(l')_{\downarrow y'} \mid [y' := e]^{\ell'} \lor [y' \leftarrow ch]^{\ell'} \}
\text{f.a. } (l' : y') \in \text{aliases}(l : y) \} \]

\[ \phi(S, x.f, y.f', \ell) = S[x.f \mapsto \bigcup \{ \text{TA}(l')_{\downarrow y'} \mid [y' := e]^{\ell'} \lor [y' \leftarrow ch]^{\ell'} \}
\text{f.a. } (l' : y') \in \text{aliases}(l : y) \} \]

\[ \phi(S, x, v_1 v_2, \ell) = \begin{cases} S[x \mapsto 1] & \text{if } v_1 \text{ is a } \textbf{source} \\ S[x \mapsto \Phi^v_{\text{exit}}] & \text{otherwise} \end{cases} \]

- \text{aliases}(l : u) \text{ produced by existing context-} \textbf{insensitive} \text{ PTA.}
- tainting a struct-member taints the entire struct
- same for slices & built-in key/value map data structure
Our analysis

Assignments, from/to struct members, calls:

\[
\Phi(S, [x := e]) = \phi(S, x, e, \ell)
\]

\[
\phi(S, x, y.f, \ell) = S[x \mapsto \bigcup \{ TA(l') \downarrow y' \mid [y' := e]^{l'} \lor [y' \leftarrow ch]^{l'} \}
\]

\[
\phi(S, x, v_1 v_2, \ell) = \begin{cases} 
S[x \mapsto 1] & \text{if } v_1 \text{ is a source} \\
S[x \mapsto \Phi_{\text{exit}}^{v_1}] & \text{otherwise}
\end{cases}
\]

- \texttt{aliases}(\ell : u) \text{ produced by existing context-insensitive PTA.}
- tainting a struct-member taints the entire struct
- same for slices & built-in key/value map data structure
Channel Handling

\[ \Phi(S, [x \rightarrow ch]^l) = S[ch \mapsto S(x)] \]

\[ \Phi(S, [x \leftarrow ch]^l) = S[x \mapsto \mathcal{A}] \]

where

\[ \mathcal{A} = \bigcup \{ TA(l')_{ch'} \mid [x' \rightarrow ch']^l \} \]

f.a. \((l' : ch') \in \text{aliases}(l : ch)\)

- existing PTA knows about channels
- we lose information about ordering of messages in channels
  (even in obvious cases)
Sanitizers & Monitoring

- Taint analysis with only sources and sinks too restrictive
- Certain operations *untaint* data:
  - SQL injections: filter out dangerous characters
  - Password example: data flowing through hash-function sanitized
- Natural extension through third set of operations: *sanitizers*

- Interesting questions:
  - Where to report tainted flow at runtime (early/late)
  - Minimize taint-tagging at runtime
  - Automated placement of sanitizers to “repair” programs
    (for C#: Livshits, POPL’13)
Practical Evaluation

- Extensive list of sources & sinks from Eric’s security projects
- Test suite with hand-crafted examples: ✓
- Runtime bounded by lattice height, calling contexts
- Real-world case study: well...
  - fine-grained analysis descends into imported packages (fmt.Print)
  - case study requires domain specific properties
    (and by now everyone is avoiding SQL injections...)
  - SSA-based infrastructure requires setup that actually compiles
    (git clone github.com/random/goproject not enough)

- High-profile targets:
  - Docker, Dropbox, Android development
Summary: Challenge in Go

- Traditional data-flow analysis, for a newer language
- Information flow: sources to sinks (via sanitizers)
- Mixes problems from C (structs, pointers) with problems from $\pi$-calculus (channels)
- Combination of off-the-shelf components (SSA representation & PTA from Go compiler infrastructure, vasco interprocedural analysis)

- "More research is needed on...":
  - Fine tune white-list/black-list for imports
  - Or: precompute & cache standard libraries?
  - Collect (performance) data on real-world Go code

---

1E. Bodden, K. I Pun, M. Steffen, V. Stolz, A.-K. Wickert
Summary: Challenge in Go

- Traditional data-flow analysis, for a newer language
- Information flow: sources to sinks (via sanitizers)
- mixes problems from C (structs, pointers)
  with problems from $\pi$-calculus (channels)
- combination of off-the-shelf components (SSA representation & PTA from Go compiler infrastructure, vasco interprocedural analysis)

- “More research is needed on…”:
  - fine tune white-list/black-list for imports
  - or: precompute & cache standard libraries?
  - collect (performance) data on real-world Go code

---

1E. Bodden, K. I Pun, M. Steffen, V. Stolz, A.-K. Wickert