Don't let data *Go* astray

A Context-Sensitive Taint Analysis
for Concurrent Programs in Go

Ka I Pun[1], Martin Steffen[1], Volker Stolz[1,2], Anna-Katharina Wickert[3], Eric Bodden[4,5], and Michael Eichberg[3]

[1] University of Oslo, Norway
[2] Bergen University College, Norway
[3] Technische Universität Darmstadt, Germany
[4] Paderborn University, Germany
[5] Fraunhofer IEM, Germany

Abstract

Taint analysis is a form of data flow analysis aiming at secure information flow. For example, unchecked user input is considered typically as "tainted", i.e., as untrusted and potentially dangerous. Untrusted data may lead to corrupt memory, undermine the correct functioning or privacy concerns of the software otherwise, if it reaches program points it is not supposed to. Many common attack vectors exploit vulnerabilities based on unchecked data and the programmer's negligence of foreseeing all possible user inputs (including malicious ones) and the resulting information flows through the program.

We present a static taint analysis for Go, a modern, statically typed programming language. Go in particular features concurrent programming, supporting light-weight threads dubbed "goroutines", and message-based communication. Beside a classical context-sensitive taint analysis, the paper presents a solution for analyzing channel communication in Go. A longer version of the material will appear in [2].

1 Motivation

The high-level programming language Go is gaining traction, being used in various software products like Docker and Dropbox, as well as for Android and iOS applications [4]. Support for concurrency based on goroutines and channel communication lies at the very core of Go's design. While standing, at least syntactically and in spirit, in the tradition of C, Go is lightyears ahead compared to the more low-level language C, when it comes to considering "secure" computation. Two notable improvements on that front are Go's advanced static type system and the absence of pointer arithmetic. But still, the concurrency features and the intended distributed and mobile applications and platforms, make the task to ensure secure computation more challenging. In open, interactive and even mobile applications, it is even more vital: One needs to ensure that unchecked, arbitrary user input does not reach sensitive points in the program, potentially leading to memory corruption or other violations, and that private data does not go astray, leaking to the outside or other applications.

To counter such vulnerabilities, we have developed and implemented a static taint analysis, which is a specific form of flow analysis, for Go. The analysis is context-sensitive, covers reference types and relies on the corresponding packages of the Go-compiler. In particular, we cover sound analysis of taint information flow for channel-based communication.

2 Static taint analysis

Information flow analysis [3] is widely studied and used for many languages, including Java and C, but is currently not implemented for Go. A taint analysis identifies flows of private information to untrusted places, e.g., a SMS to the attacker, or untrusted information such as user input passed on or processed without sanity checks. We will concentrate on the former type (privacy-based) of taint analysis. The analysis defines private information and untrusted places through a list of *sources* and *sinks.* In that it can be seen as a form of a *def-use*-analysis. Sources are API calls which could read sensitive data, and sinks are API calls which can write data to an untrusted place. Our taint analysis is static and context-sensitive, concentrating on direct information flows.

2.1 The Go programming language

The small example from Listing 1 illustrates the analysis, highlighting a few Go features. The main function spawns a call to f as a new goroutine (think "thread") using the keyword `go`, handing over the freshly created synchronous channel via the actual parameter `ch`. The channel is (obviously) shared between parent and child goroutine, and used to communicate string values from parent to child and, thereby, also synchronizes between them: sending to the channel is done in location n_6 and reading from it in location f_2. The example defines two functions *source* and *sink*, whose only purpose here is to illustrate the concepts of sources and sinks for the taint analysis. Go supports pointers (though no pointer arithmetic as in C): In the example, the argument to the *sink*-function is passed as a reference.

```
      func main() {
n1        x := "hello, world"
n2        ch := make(chan string)
n3        go f(ch)
n4        sink(&x)
n5        x = source()
n6        ch ← x
      }
f1    func f(ch_1 chan string) {
f2        y := ←ch_1
f3        sink(&y)
      }
      func sink(s *string) {}
      func source() string {
          return "secret"
      }
```

Listing 1: A Go program with channels, goroutines, pointers.

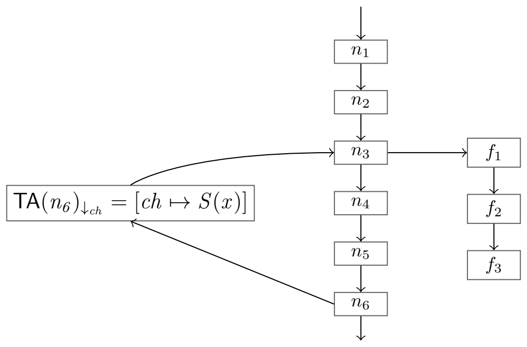

Figure 1: Flow graph for Listing 1 with the additional edge for a write to a channel. Edges to sinks/from sources elided for simplicity.

2.2 Implementation

Building upon similar analyses such as [5], the implementation is context sensitive, i.e., a function call (including *asynchronous* functions) is analyzed differently depending on the call-site *context*, which, in our case, is a "value context" containing information about the taint status of the actual parameters. The analysis partly relies on existing technologies and libraries.

First, we use Go's static single-assignment (SSA) intermediate representation for a worklist-based iteration algorithm. Secondly, we use an inclusion-based pointer analysis (see [1]) to handle pointers in a more precise way.

Figure 1 shows the flow graph for the example program, where the nodes correspond to the "line annotations" in the code. The implementation uses a worklist algorithm for fixpoint iteration, working through the nodes. In the example, the *sink*-function is called twice, once in the main function with the untainted string value `"hello, world"`, and in the child goroutine executing f. In the latter case, the value delivered to the sink is received via the channel connecting the two goroutines, and in that case the value originates from the *source*-procedure. To correctly have the analysis report a possible flow of tainted data from a source to a sink, we introduce for each created channel a new type of data flow node, with incoming edges from the send-operations to the channel and outgoing edges to read-operations that may access the channel. The transfer function for the new node only transfers the lattice value of the channel (see the node labelled TA(...) in Figure 1), and not the complete lattice of the writing node.

Listing 1 uses the unbuffered channel only once. In larger programs buffered channels or many writes to a channel are likely. If a channel receives a `tainted` and an `untainted` value, the static analysis will over-approximate the channel's taint status to the lattice's top value $\top$.

3 Potential for monitoring

For some paths, a static analysis overapproximates the results, and a dynamic analysis under-approximate the results. If we introduce monitoring for these parts, we can improve our results. A scenario is to use monitoring as a sanitizer. A sanitizer works like a white list and changes a tainted value to an untainted, e.g. applying the hash function on a password should change the lattice value to untainted. A classical taint analysis ignores that the hash function changes the taint status to untainted and produces a false positive. A more complicated approach is to monitor the concurrent behavior of a program. The idea is to influence the scheduling so that a tainted value will only take safe paths.

References

[1] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, University of Copenhagen, 1994. Available as DIKU report 94/19.

[2] E. Bodden, K. I. Pun, V. Stolz, M. Steffen, and A.-K. Wickert. Information flow analysis for Go. In *7th Intl. Symp. On Leveraging Applications of Formal Methods, Verification and Validation*, volume 9952 of *LNCS*. Springer Verlag, Oct. 2016.

[3] D. Denning. *Secure Information Flow in Computer Systems.* PhD thesis, Purdue University, 1976.

[4] Go Authors. Mobile golang/go Wiki GitHub, Feb. 2016. URL https://github.com/golang/go/wiki/Mobile.

[5] R. Padhye and U. P. Khedker. Interprocedural Data Flow Analysis in Soot Using Value Contexts. In *2nd ACM SIGPLAN Intl. Workshop on State Of the Art in Java Program Analysis*, SOAP '13, pages 31–36, New York, NY, USA, 2013. ACM.