Information Flow Analysis for Go

Eric Bodden[1,2], Ka I. Pun[3], Martin Steffen[3], Volker Stolz[3,4(✉)], and Anna-Katharina Wickert[1]

[1] Technical University of Darmstadt, Darmstadt, Germany
[2] University of Paderborn, Paderborn, Germany
[3] University of Oslo, Oslo, Norway
stolz@ifi.uio.no
[4] Bergen University College, Bergen, Norway

Abstract. We present the current state of the art of information flow analyses for Go applications. Based on our findings, we discuss future directions of where static analysis information can be used at runtime to for example achieve higher precision, or optimise runtime checks. We focus specifically on outstanding language features such as closures and message-based communication via channels.

1 Introduction

The Go language [7,8,25] is a relative newcomer to the programming language stage. Nonetheless, it has been quickly taken up for application development by big players, most notably, Docker. There, it is used as a language for their popular software container framework. Backed by Google, it is also speculated to be the future language for Android development, and recently the mobile development kit was published.

Like any other software, Go programs are frequently exposed to "hostile" environments, whether it is on a web-facing server, or soon on a mobile phone. Both are constant targets for attacks by hackers and malicious applications, which try to break into the system through malicious input or specially crafted interactions. To prevent the violations of program executions induced by malicious data, one effective way is to *statically* analyse the flows of such data within programs.

We present here our information flow analysis of Go programs, where we focus on the more interesting features of the Go language, such as channel-based communication and deferred execution. This analysis is the foundation for a monitoring framework to thwart such attacks, by identifying the flow of potential malicious (tainted) data from a set of pre-defined sources to a set of sinks that this data must not reach unprocessed. Based on our findings on the precision

The work was partially supported by the Norwegian-German bilateral PPP project GoRETech (GoRuntime Enforcement Techniques), the EU COST Action IC1402 "ARVI—Runtime Verification Beyond Monitoring" and the EU project FP7-610582 ENVISAGE: Engineering Virtualized Services.

T. Margaria and B. Steffen (Eds.): ISoLA 2016, Part I, LNCS 9952, pp. 431–445, 2016.
DOI: 10.1007/978-3-319-47166-2_30

of our analysis, we give recommendations how monitoring can complement the inevitable gaps in a static analysis.

Related Work. Static analysis for information flows has been widely studied: Denning and Denning [5,6] present a mechanism in terms of a lattice model to guarantee secure information flows for sequential statements. Such a construct is the foundation of many static analysis frameworks, e.g., the Monotone Framework [18], which is also the starting point for our approach. Andrews and Reitman [1] propose an axiomatic approach to certifying flows in both sequential and parallel programs. Type systems are also a common approach to ensure noninterference for well-typed programs, e.g., Volpano et al. [27] formulate Denning's work in the form of a type system for a core imperative language; Pottier and Simone [21] propose a type-based analysis for a call-by-value λ-calculus.

Apart from guaranteeing program security by tracking the flow of sensitive data, our approach identifies potential tainted data flows within programs at compile time, which helps in detecting bugs as well as avoiding attacks by malicious applications. A number of work has been done to analyse flow information of tainted data using similar idea: Arzt et al. [2] propose a static taint analysis for Android applications. Livshits and Lam propose a variant of SSA to discover bugs in C programs [15], and use a context-sensitive pointer alias analysis to detect security violations in Java applications [14]. Pistoia et al. [20] present a control- and data-flow framework to find tainted variables in Java bytecode. Information flow analyses have also been applied for languages like PHP [13] and JFlow [17], which is an extension to the Java language. While Go shares some of the general features with those imperative languages, we also take a look at some of its novel constructs, which are mostly related to concurrency.

Paper Overview. Section 2 provides the background of the Go language and information flow analysis; Sect. 3 presents the abstract syntax of the language and the analysis for Go programs; Sect. 4 illustrates our implementations with examples, Sect. 5 discusses the potential for monitoring for Go programs, and finally Sect. 6 concludes the paper.

2 Preliminaries

2.1 The Go Language

Go, a language backed by Google, has gained a certain amount of traction after its inception. Its advertised design principles as being simple and concise together with its surface syntax make the language identifiable in the tradition of C. At its core, Go is a lexically scoped, concurrent, imperative language with higher-order functions, supporting object-oriented design (while notable not supporting classes nor inheritance). Concerning concurrency, Go's primary feature is *asynchronous* function calls (resp. asynchronous method calls), called *goroutines* (basically a lightweight form of threads with low overhead and lacking known thread synchronisation mechanisms such as wait and signal). The second core concurrency construct is (typed) *channel* communication, in the tradition

of languages like CSP [9,10] or Occam. Since (references to) channels can be sent over channels, Go allows "mobile channel" flexibility for communication as known from the π-calculus [16].

Thus, despite the "simple" surface syntax in the tradition of C, Go combines features which are challenging from a program analysis perspective: Reference-data, imperative features, arrays, and slices require point-to analyses. Control-flow analyses are needed to obtain data-flow analyses of acceptable precision in the presence of higher-order functions. The Go compiler (at least in a developer branch) supports a static-single assignment intermediate format to facilitate flow analyses. Shared variable concurrency is featured by Go but frowned upon. The more dignified and recommended way of concurrent programming via message passing, using either synchronous or buffered channels of finite capacity. The static analysis of such channel communication has similarities to pointer analysis, as channels are a referenced shared data where channel pointers themselves can be communicated via channels (or stored and handed over to procedures as other references, as well). The analysis of data flow in the context of channel communication is challenging in itself, but at least avoids unprotected concurrent access to shared mutable data and shields the programmer from the subtleties of Go's weak memory model. In this work, we do *not* consider shared variable concurrency.

2.2 Information Flow Analysis

We discuss here in particular the challenges of information flow analysis when applied to Go. Information flow analysis [1,5] attempts to determine whether a given program can leak sensitive information, either directly or through indirect channels, for instance when secret values influence timing behaviour or power consumption, so-called indirect information flows. Dynamic information flow analysis attempts to detect such leaks by monitoring an application's execution. Such dynamic analyses are generally to detect direct information flows only, i.e., such flows that occur through direct memory copies. This is due to the fact that indirect flows occur through control-flow dependencies on secret values, and in particular because a program can leak information as it does *not* execute a certain behaviour at runtime. Since behaviour that does not execute cannot be monitored, this precludes the detection of certain indirect flows.

Static code analysis, however, can analyse all of a program's possible executions, detecting control-flow dependencies and also such "missing behaviour". It is for that reason that static analysis can detect not only direct but also indirect information flows. Recent research has shown that a static pre-analysis can assist a subsequent dynamic analysis by finding control-flow dependencies that can leak secret information and defining a special instrumentation scheme at runtime that signals when the respective branches are taken. Depending on some properties of the monitored programming language, and depending on the scope of the static pre-analysis this can allow the dynamic analysis to even monitor all possible information leaks at runtime. Indirect information flows, however, have the tendency to cause so-called "overtainting", where an analysis

ends up tracking many—typically too many—information flows, the majority of which is to the security analyst often irrelevant. The underlying problem is a deeply semantic one: an indirect information flow signals not that a program leaks data but it signals that a program leaks *information about* data. But how much information will allow for a practical attack? This question is extremely hard to decide in terms of a program's structure. Recent work has thus focused on making the analysis of indirect information flows more precise, for instance by also regarding so-called *declassification*, i.e., the intentional disclosure of information about secret data. Since such declassification is intentional, a program analysis should avoid signalling it as leak of secret information. Declassification is generally quite essential. Without declassification, for instance, a password dialog may not even signal to the user whether or not the password was entered correctly, as this would signal some information about the secret password, even though this information is essential to reveal.

When conducting information-flow analysis for programs with pointers, it is essential to pair it with a pointer analysis, as otherwise the analysis would fail to resolve aliasing relationships. Consider the code sequence `a.f = secret(); print(b.f);`. In this code, to determine whether the program may `print` the `secret`, an analysis must know whether `a` and `b` alias, i.e., point to the same object. Pointer analysis is generally expensive to compute, and to yield appropriate precision must share certain design properties with the alias analysis it seeks to support. Generally, a high-precision analysis should be context sensitive and flow sensitive, for instance. If the accompanying alias analysis does not share the same level of context and flow sensitivity, then this can cause imprecision to creep into the information-flow analysis, ultimately resulting in false warnings that threaten to distract the security analyst from the important true warnings.

3 Analysis

In this section, we present our information flow analysis for Go programs, and illustrate its use with some examples in the next section. The analysis is based on a suitable subset of the full language which is easy to formalise yet covers the most important features.

Information flow describes a dynamic property: in our setting, it is any value that originates from a particular API call (as denoted by a list of *sources*), and is used within the execution of the program. If the execution reaches a call to any of our denoted *sinks*, and the value is passed as a parameter, we would like to report an error or a warning. Of course, such tracking of data flow can happen at runtime, but naturally we are interested in whether we can give certain guarantees for a program *before* it is run. We thus need to reformulate this problem in the terms of a static analysis that can be defined in terms of the program *source code*.

To simplify the discussion, we assume in the paper a simplified representation of (Go) source code, assuming for example that each statement contains at most a single function call, with only variables or constants as arguments. Also, we stipulate that all variables must be initialised when declared.

In the following, we will handle expressions representatively build up by using primitive types, structs, channels and function types. We elide the other useful, built-in datatypes in Go, such as slices (arrays) and key/value maps, and appeal to the reader's intuition that common approaches to over-approximation of reference types as in the case of structs and channels can be applied.

The abstract syntax is given by Table 1. We shall concern ourselves with statements that are assignments to locally declared variables or struct members, conditionals, finalizers (`defer`), or initiators of concurrent execution (`go`). In addition, we have the channel operations read and write, return from function, and of course sequential composition of statements.

Table 1. Abstract syntax

$s ::= x := e \mid x.f := e \mid \texttt{if}\ v\ \texttt{then}\ s\ \texttt{else}\ s \mid \texttt{defer}((\lambda x.s)\ v)$	statements
$\quad \mid \texttt{go}\ s \mid x \leftarrow y \mid x \rightarrow y \mid \texttt{return}\ v \mid s; s$	
$e ::= v \mid v\,v \mid \texttt{makeChan}$	expressions
$v ::= x \mid x.f \mid () \mid \texttt{true} \mid \texttt{false} \mid \lambda x.s$	values

Expressions may be variables or values of the aforementioned supported types, functions calls (written as application here), or channel initialisation. Go's multiple return values from function calls would require a minor extension of the syntax which would not add much for our discussion, as would slice- and map manipulation. Function definitions straightforwardly have typed formal parameters, and bodies composed of statements.

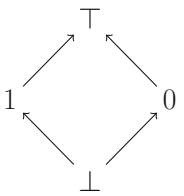
We can then restate the problem as follows: we would like to report a warning, if the return value of a function call labelled as source is assigned to a variable, and the value *may* be propagated through assignments and function calls to a variable which is used as an actual parameter in a function call to a sink.

Furthermore, our analysis must take *channels* into account in a sound way: if a sensitive (tainted) value is written into a channel, as an over approximation, we assume that a read from that channel may return the tainted value. As static analysis of channel-based communication has been studied extensively for example in [11], we do not go into the details here and leave specialising this part of our analysis towards a more precise solution using those techniques for future (implementation) work.

3.1 Lattice

Our intended analysis can easily be expressed with the well-known concept of Monotone Framework [18]. For our taint analysis, we define a simple lattice where a value in a variable (attribute) is initially marked as "undefined" ($\bot$), and based on custom black-/whitelist of API calls, marked as either "tainted" (1), "untainted" (0), or "both" ($\top$). We define the least upper bound ($\cup$) of two taint values in a straight-forward manner:

$\cup$	$\bot$	1	0	$\top$
$\bot$	$\bot$	1	0	$\top$
1	1	1	$\top$	$\top$
0	0	$\top$	0	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$



3.2 Aliasing and Channels

The Go language has several reference types, most prominently: structs, slices (arrays), and maps. Again, we do not model the required tracking of aliasing explicitly, but assume availability through a sound, context-insensitive over-approximation. Thus we make use of the following function[1], which, given a variable at a particular statement, over-approximates the set of allocation sites of objects and the variable they have been assigned to:

$$\textsc{pta} : \textsc{Var} \times \textsc{Lab} \to \mathcal{P}(\textsc{Loc}).$$

We use this function to additionally maintain the function

$$\mathit{aliases} : \textsc{Lab} \times \textsc{Var} \to \mathcal{P}(\textsc{Lab} \times \textsc{Var}),$$

which we require to identify potential aliases created through field-use in `structs`.

Another example of a reference type are of course Go's (typed) channels. Our rules for assignments, which are defined later in the section, track correctly the taint information associated with a channel when aliasing (e.g., `ch := makeChan; ch':= ch`) because of the points-to analysis described above. Additional processing that does not follow the control flow is now required when writing a tainted value into the alias `ch'`. A very coarse and obvious solution to achieve the required dataflow is to add dependencies between a write to a channel to all reads from it. A related analysis built on top of that allows a sound over-approximation of the *peers* of a channel referenced by a variable in a particular location, that is, all uses of the same channel reference in read or write statements.

3.3 Taint Analysis via the Control-Flow Graph

For the *intraprocedural* part of our analysis, we can set up the Monotone Framework with the help of the control-flow graph (CFG). As ultimately our analysis should warn on particular statements (function calls to sinks with tainted actual parameters), we decide on a *single-instruction graph*, i.e., each node in the control-flow graph represents a single, normalised (as per our grammar)

[1] https://godoc.org/golang.org/x/tools/go/pointer.

instruction. Conditionals result in branches in the control-flow graph, and loops lead to (additional) back-edges to nodes earlier in the graph. We do not describe how to obtain the graph, but rather refer to [18] and recapitulate the essential ingredients. We assume that the function NODES returns a set of labeled statements $[s]^l$ of a program. Furthermore, the flow-function

$$\text{FLOW} : \text{LAB} \to \mathcal{P}(\text{LAB} \times \text{LAB})$$

returns the edges in the CFG for a uniquely labeled statement $[s]^l$, and its extensions FLOW*, yielding the CFG for the entire program. Calls to `go`-routines can be handled through an additional control flow-edge from the caller to the body, as control does not return, and we only permit channel-based communication. Full Go also supports—but discourages—locks and shared variables.

To effectively be able to emit a warning, our analysis framework must yield the following information: is any of the actual parameters in a function call to a sink marked as possibly tainted on the entry to the statement? Information by our taint analysis is thus given for each node (statement) in the CFG by the partial TA function of type: LAB $\to$ (VAR $\to \mathcal{L}$), which yields the taint information associated with variables in scope at the particular node:

$$\mathsf{TA}(l) = \Phi(S, N^l) \quad \text{where } S = \bigcup \{\mathsf{TA}(l') \mid (l', l) \in \text{FLOW}^*(P)\} \text{ and } N^l \in \text{NODES}(P).$$

Note that we collect all the taint information flowing to the statement labelled with l in the set S. The function $\Phi(S, N^l)$ defined in Table 2 derives the equations for the standard programming constructs. In essence, the analysis resembles the well-known analyses of Def-Use chains or Reaching Definitions, extended by the required notions of transitivity and aliasing. A standard worklist algorithm can be used to generate the smallest solution to our dataflow problem, which we can then query for all actual parameters, at each statement that is marked as a sink.

One way to propagate taint information in our core language is to assign an expression to either a variable $x := e$ or to a struct member $x.f := e$. We use in Table 2 a function ϕ to derive the taint information of the expression on the right-hand side of an assignment. The expressions, including creating new channels `makeChan`, unit, boolean values and function definitions, do not taint any variable, and therefore the variable x (or struct member $x.f$) on the left-hand side is marked as *untainted* ($x \mapsto 0$). In the case where the expression is a variable y, the function ϕ updates the analysis result of x with the one of y at the current state. For assigning a struct member $y.f$, we have to collect the taint information from all the aliases of the reference y with the help of the *aliases* function described above. For function calls $v_1 v_2$, in case the called function v_1 is a *source*, the assigned variable x is marked as *tainted* ($x \mapsto 1$). Otherwise, we derive the taint information of the called function with *interprocedural analysis*.

Interprocedural Analysis. The function $\Phi^{v_1}_{exit}$ is integrated with the worklist algorithm as developed by Padhye and Khedker [19], which we have implemented to achieve a flow- and context-sensitive analysis. The idea of the authors is to

Table 2. Taint analysis

$$
\begin{aligned}
\Phi(S,\ [x := e]^l) &= \phi(S, x, e, l)\\
\Phi(S,\ [x.f := e]^l) &= \phi(S, x.f, e, l)\\
\Phi(S,\ [\texttt{defer}((\lambda x.s)v)]^l) &= id(S)\\
\Phi(S,\ [\texttt{go}\ s]^l) &= id(S)\\
\Phi(S,\ [x \rightarrow ch]^l) &= S[ch \mapsto S(x)]\\
\Phi(S,\ [x \leftarrow ch]^l) &= S[x \mapsto \bigcup\{\mathsf{TA}(l')_{\downarrow ch'} |\ [x' \rightarrow ch']^{l'} \text{ f.a. } (l' : ch') \in aliases(l : ch)\}]\\
\Phi(S,\ [\texttt{return}\ v]^l) &= id(S)
\end{aligned}
$$

$$
\begin{aligned}
\phi(S, x, y, l) &= S[x \mapsto S(y)]\\
\phi(S, x, y.f, l) &= S[x \mapsto \bigcup\{\mathsf{TA}(l')_{\downarrow y'} |\ [y' := e]^{l'} \vee\ [y' \leftarrow ch]^{l'}\\
&\qquad\qquad \text{f.a. } (l' : y') \in aliases(l : y)\}]\\
\phi(S, x.f, y, l) &= S[x.f \mapsto S(y)]\\
\phi(S, x.f, y.f', l) &= S[x.f \mapsto \bigcup\{\mathsf{TA}(l')_{\downarrow y'} |\ [y' := e]^{l'} \vee\ [y' \leftarrow ch]^{l'}\\
&\qquad\qquad \text{f.a. } (l' : y') \in aliases(l : y)\}]\\
\phi(S, x, (), l) &= S[x \mapsto 0]\\
\phi(S, x, \texttt{true}, l) &= S[x \mapsto 0]\\
\phi(S, x, \texttt{false}, l) &= S[x \mapsto 0]\\
\phi(S, x, \lambda x.s, l) &= S[x \mapsto 0]\\
\phi(S, x, v_1 v_2, l) &= \begin{cases} S[x \mapsto 1] & \text{if } v_1 \text{ is a } \textit{source}\\ S[x \mapsto \Phi^{v_1}_{exit}] & \text{otherwise}\end{cases}\\
\phi(S, x, \texttt{makeChan}, l) &= S[x \mapsto 0]
\end{aligned}
$$

differ between calls and to save the data flow values for every context. Therefore, the algorithm can avoid that a function with identical input parameters is analysed multiple times. This is built upon the assumption that equivalent input parameters of a function will yield the same data flow values at the exit node of the function. Their approach increases precision over the trivial approach, where every exit-value from a return-statement flows back to *all* call sites, not just the actual caller. The algorithm uses an additional calling context $X := (S, \textit{actual param})$, which guarantees that identical contexts produce identical results. We will later describe the actual working on an example.

Another way to pass on taint data is to through channel communications. Sending values or variables to a channel $x \rightarrow ch$ will propagate the taint information of x to ch. To read from a channel $x \leftarrow ch$, we have to gather the knowledge of all the possible aliases of the channel to which tainted data may be sent. The statements, including finalizers (`defer`) and initiators of concurrent execution (`go`), do not affect the taint information.

Soundness. Here we elide formal claims and proofs with regard to the soundness of the analysis. The small-step operational semantics by Steffen [23] could serve as a starting point for such a formalisation, and the corresponding properties.

4 Implementation

Our information flow analysis relies partly on existing technologies and libraries: although our above analysis is formulated in terms of the single-instruction control-flow graph, our prototype implementation uses existing libraries from the Go compiler tool-chain and tools that go beyond this simplistic view. With the help of those libraries, we obtain the static single assignment-form (SSA)[2] [4], interprocedural call-graph[3], and the necessary points-to information. We shortly describe the APIs available to us.

The SSA library consists primarily of four interfaces. Firstly, the `Member` interface holds the member of a Go package being functions, types, global variables and constants. Secondly, the `Node` interface describes a node from the SSA graph. Valid values for the `Node` interface are types fitting either to the `Value` or `Instruction` interface. An expression which leads to a value is of type `Value`. A statement using a value and computes are part of the `Instruction` interface.

Through the fact that we consider the distinction between calling contexts, we need to differ whether a node is a call or not. For this aim, we use the `CallInstruction` interface allowing us to distinguish between a function call and Go specific calls being a goroutine and a defer statement. To define the desired behaviour, we need two additional inputs for our analysis: a blacklist of API calls that produce tainted values, and a whitelist of calls that either produce untainted values, or turn tainted ones into untainted.

A common property that is investigated with a taint analysis is whether unsanitized user input can e.g. reach SQL queries, where it could lead to SQL injection attacks. In that case, any user input, that is, console input, or e.g. data submitted through an HTML form, is marked as tainted. Correspondingly, we add those calls to our blacklist and call them *sources*. Our analysis shall report a warning if a tainted value reaches a *sink*. Sinks are again specified separately, just like sources.

4.1 Example

In this section, we explain our current taint analysis approach with the program in Fig. 1, which primarily reads a file and prints the file content to the standard output. The program consists of a main function and two additional functions h and g. The function h reads the first eight bytes of a file and returns the bytes as a string c and the status r. The function g copies the input string to another variable b and returns the variable. The main function calls the function g with a constant string value a and once with eight bytes from a file s. The last input parameter for g is obtained with the help of function h. `os.File.Read` is a *source*, and `fmt.Print` a *sink*.

To get the results, we use the functions and the lattice described in Sect. 3. The entry point of our analysis is the main function within a Go program, where

[2] https://godoc.org/golang.org/x/tools/go/ssa.
[3] https://godoc.org/golang.org/x/tools/go/callgraph.

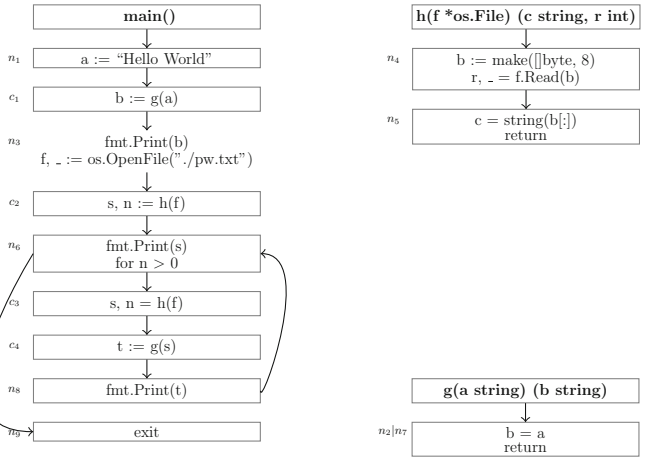


Fig. 1. A simplified control flow graph with different contexts

nothing is initialised. Thus the lattice at this point is empty and the worklist contains n_1, c_1, n_3, c_2, n_6, c_3, c_4, n_{10} and n_{11}. At the beginning a context X_0 is created. The entry value is the empty lattice and the exit lattice is currently not set because the execution is not yet finished.

The first element n_1 is removed from the worklist and then processed. It receives `untainted` for the variable a from the transfer function defined in Table 2. Through the next removal c_1 is obtained, being the first call in the example. A new context X_1 having an untainted value as input is created and a new transition from X_0 to X_1 is added. After the context is created, all nodes of the function are added to the worklist. In the following step n_2 is removed to be processed, and the exit lattice of X_1 is set to a^0, b^0 because n_2 contains a return statement.

As a subsequent step, the algorithm selects node n_3 for processing and creates a lattice with the `tainted` variable f. Afterwards c_2 is collected and produces a new value context X_2. The transfer function of the call passes a tainted parameter. Therefore, the entry lattice of the new context contains the tainted parameter. In the context X_2, n_4 is processed first. The transfer function returns for the variables b and r the `tainted` value and updates the lattice for the node. The next node n_5 is then selected, and the transfer function computes that the variable c is also `tainted`. The exit lattice for X_2 is updated such that b, r and c are `tainted`.

Back in context X_0, the algorithm picks node n_6 for processing and detects that a `tainted` value reaches a sink. Afterwards, c_3 with a `tainted` value as a parameter is handled as context X_3. The algorithm checks whether a call to function h with a `tainted` parameter already exists, finds X_2 and adds the transition from $\langle X_3, c_3 \rangle$ to X_2. For the succeeding call c_4 the value context currently does not exist, so a new context X_4 is created and the nodes of g are added to the worklist. In the ensuing step, the algorithm processes n_7, which leads to

variable b becoming `tainted`. The next node from the worklist is n_8, which also reports a warning. Since all contexts with a fitting entry parameter already exist, the remaining nodes do not lead to the creation of new contexts, but only the transitions are added.

4.2 Concurrency in Go

A language specific characteristics of Go is that it supports concurrent programming by design [8]. The idea is that only one goroutine is allowed to access a value. Hence, Go encourages use of channels for message passing instead of concurrent access to shared variables. The channels are first-class values in Go.

As channels are an essential part of concurrent Go programs, our analysis must be able to handle channels correctly. The challenge is that channels are used for concurrency and therefore multiple different execution paths are possible. Our current idea is to add additional information in case of writing values to a channel. Every goroutine which uses the channel should get an entry node with the identity function. This allows the analysis to build the least upper bound of all incoming edges.

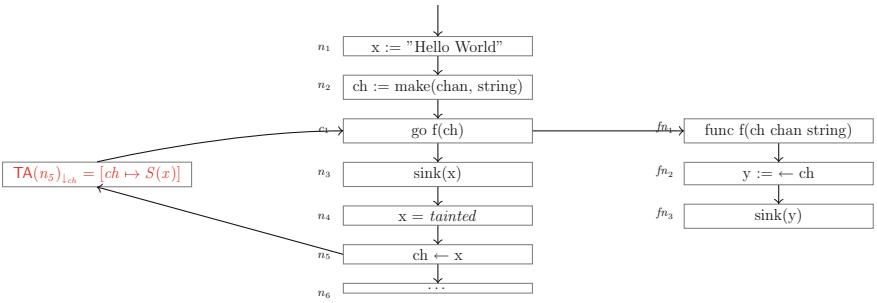


Fig. 2. An example which explains the challenges of channels for our analysis.

The program in Fig. 2 is a simple example which uses channels. First a variable x and a channel ch are created. The channel is used in a goroutine which calls function f. The function f reads a value from the channel. The value reaches a sink at node fn_3. The red node indicates the sub-equation for the flow-function that we use to propagate the taint-state of the channels back to (after) its declaration site, by dropping all other information from the exit of the `readChan` through this exit-branch.

While in this example, due to the unbuffered nature of synchronised channels, the value read from ch will always be tainted, in a larger program, non-deterministic execution orders may yield tainted or untainted results from read-statements, which the static analysis over-approximates to $\top$. In other words, a sink in a goroutine that consumes data from a channel, may only be reached by a tainted value in some cases. This illustrates the need for a runtime component, or sanitizers that takes care of those false positives.

In addition to the problem described above, the example illustrates a challenge for a precise solution of our analysis. We assume that x is untainted when it reaches the sink at node n_3 and is tainted at node n_4. Through the fact that the channel *ch* gets x as a value, we should assume that *every* read to *ch* could possibly produce a tainted value. As a conclusion the node fn_3 is reached by a tainted value.

Consequently, our analysis must know which goroutines uses *ch*. This can be achieved by a backward analysis. The challenge is to update only the channel with the taint information and not the tainted value x. Such an imprecision would yield to a report at node n_3, which is obviously a false positive.

A correct statically implementation of the analysis should therefore only report a potential flow at node fn_3. The precision of the analysis could be increased through a dynamic observation of the potential dangerous paths of concurrent execution. Then fn_3 should only be reported if fn_3 is executed after node n_5. This would make the analysis more precise, but will in general not be possible to deduce statically.

5 Potential for Monitoring

We discuss in the following potential techniques to introduce monitoring in those paths when a static information analysis cannot determine whether this path is safe. Such monitoring then effectively fulfils the role of sanitizers, and can in a second pass be put on the whitelist and used to check that all dangerous paths are indeed covered.

Instrumenting Go Programs. Instrumentation of source code is often used in Go since Go 1.2 [26]. One usage of instrumentation is collecting data for determining coverage. The placement of the instrumentation, however, may suffer from similar problems as static analysis, in that optimal placement of instrumentation is difficult to determine.

Such test coverage instrumentation tools can give us hints for our instrumentation to monitor flow as to whether paths we are interested in are executed or not. A further step is to instrument the source code with the required functionality for our taint analysis. Being interested in preventing tainted data from reaching a sink, an obvious instrumentation is to stop the execution of the program when the additional data recorded by the instrumentation reports that we are about to reach a sink.

A traditional approach to instrumentation, especially for object-oriented languages, is aspect-oriented programming. This declarative technique has been used with success e.g. for Java [24], but relies on extensive infrastructure. Although developers are experimenting with developing similar frameworks for Go, the prototypes are far from the necessary flexibility and convenient syntax, as offered for example by AspectJ [12].

Also object-oriented design techniques can be helpful in instrumenting either source code or code on the level of shared libraries: for example, as the sources

and sinks in a taint analysis will mainly be API calls, it may be easy to generate wrappers for them, and recompile an application using those proxies.

What to Monitor. Another question is of course which other properties could be interesting to monitor in Go programs. Here, we have focused on a taint analysis that is useful for services processing sensitive data.

Currently we think that two different things could be interesting to monitor: First, it can be used to add sanitizers and secondly to monitor and influence the concurrent behaviour of a program run.

The first idea is to dynamically add *sanitizers.* A typical taint analysis can only add a taint status and does not remove it. The latter is important to avoid too many false positives because the taint status spreads during the program execution. An example which produces false positives is where the program logic should enforce sanitisation. This could be achieved e.g. by a password which is entered and then send via a hash function to a server. Classically, the taint analysis will report the flow described above because the taint status of the password is propagated to the send function [22].

The second idea is to actively influence scheduling to avoid tainted paths. If we assume the example from Fig. 2, the potential leak will only be reported if n_5 is reached before fn_3 is executed. In a more complicated setting, where there are more consumers to a channel, and some of them will not pass on data to any sink, we could develop a scheduler which routes tainted data along safe paths only. Of course this requires a more advanced analysis of the communication behaviour to be able to enable processes which have the capacity to consume a tainted item without becoming blocked. This problem is closely related to deadlock avoidance in schedulers [3].

6 Conclusion

In this paper we presented our attempt at implementing an information flow analysis for the Go language. The combination of object-based language constructs such as structs and arrays, and message-passing through typed channels, requires a combination of various techniques.

In one dimension, we have static analysis components that combine intra- and context-sensitive interprocedural techniques with reference-based analyses to capture aliasing effects. In the other dimension, we need dynamic checks that compensate for the over-approximation of the taint-analysis in the case where either tainted or untainted flows come from a source to a sink can occur.

Currently, our analysis only implements the static analysis part, and we are actively investigating the alternatives for monitoring the running application, for example through instrumentation. *The code for the analysis and examples area available from the project website.*[4]

Future Work. A very interesting approach that does not require instrumentation would be to integrate tighter with the Go runtime system: the Go runtime

[4] See http://www.mn.uio.no/ifi/english/research/projects/goretech/.

already contains a sophisticated, tuned framework for tracking data races in concurrent programs. Although due to its invasiveness it incurs a noticeably performance penalty, it could reasonably be extended to taint-tracking. The runtime would only need to be informed of sources and sinks. That could be achieved by introducing annotations, as an alternative to a global (runtime-wide) list.

Since our current prototype analysis is a combination of the worklist-based analysis for intra- and interprocedural data flow, yet we rely on existing Go analyses for aliasing, the program may be effectively traversed multiple times, for each analysis separately. If the different analyses could be integrated into a single framework, we may benefit from some synergy.

Also, we do not cover all language features yet. It is unclear how higher-order functions (and their application) could be analysed successfully statically, so runtime monitoring may prove as an effective solution there.

An alternative approach to providing warnings, or termination, when reaching a sink with tainted data, could be to fail as early as possible, that is, as soon as it becomes clear that a policy will be inevitable in the future. Here, static analysis would be the main contributor to identify the best places to insert such checks, once more illustrating the need for a combined approach, with static and dynamic aspects playing together to achieve a common goal.

References

1. Andrews, G.R., Reitman, R.P.: An axiomatic approach to information flow in programs. ACM Trans. Program. Lang. Syst. **2**(1), 56–76 (1980)
2. Arzt, S., et al.: FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Androidapps. In: ACM SIGPLAN Conference on Programming Language Design and Implementation (2014)
3. Coffman Jr., E.G., Elphick, M., Shoshani, A.: System deadlocks. Comput. Surv. **3**(2), 67–78 (1971)
4. Cytron, R., et al.: Efficiently computing static single assignment form and the control dependence graph. ACM Trans. Program. Lang. Syst. **13**(4), 451–490 (1991)
5. Denning, D.E.: A lattice model of secure information flow. Commun. ACM **19**(5), 236–243 (1976)
6. Denning, D.E., Denning, P.J.: Certification of programs for secure information flow. Commun. ACM **20**(7), 504–513 (1977)
7. Donovan, A.A.A., Kernighan, B.W.: The Go Programming Language (2015)
8. Effective Go - The Go Programming Language. https://golang.org/doc/effective_go.html#concurrency. Accessed 29 Apr 2016
9. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Upper Saddle River (1985)
10. Hoare, C.A.R.: Communicating sequential processes. Commun. ACM **21**(8), 666–677 (1978)
11. Kobayashi, N.: Type-based information flow analysis for the π-calculus. Acta Informatica **42**(4), 291–347 (2005)
12. Laddad, R.: AspectJ in Action: Practical Aspect-Oriented Programming. Manning Publications Co., Greenwich (2003)

13. Livshits, B., Chong, S.: Towards fully automatic placement of security sanitizers and declassifiers. In: The 40th Annual ACMSIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 385–398. ACM (2013)
14. Livshits, V.B., Lam, M.S.: Finding security vulnerabilities in Java applications with static analysis. In: Proceedings of the 14th Conference on USENIX Security Symposium. SSYM 2005. USENIX Association (2005)
15. Livshits, V.B., Lam, M.S.: Tracking pointers with path and context sensitivity for bug detection in C programs. In: Proceedings of the 9th European Software Engineering Conference. ESEC/FSE-11, pp. 317–326. ACM (2003)
16. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, Part I/II. Inf. Comput. **100**, 1–77 (1992)
17. Myers, A.C.: JFlow: practical mostly-static information flow control. In: Proceedings of the 26th ACM Symposium on Principles of Programming Languages, pp. 228–241 (1999)
18. Nielson, F., Nielson, H.-R., Hankin, C.L.: Principles of Program Analysis. Springer, Heidelberg (1999)
19. Padhye, R., Khedker, U.P.: Interprocedural data flow analysis in SOOT using value contexts. In: Proceedings of the 2nd ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis. ACM (2013)
20. Pistoia, M., Flynn, R.J., Koved, L., Sreedhar, V.C.: Interprocedural analysis for privileged code placement and tainted variable detection. In: Gao, X.-X. (ed.) ECOOP 2005. LNCS, vol. 3586, pp. 362–386. Springer, Heidelberg (2005)
21. Pottier, F., Simonet, V.: Information flow inference for ML. ACM Trans. Program. Lang. Syst. **25**(1), 117–158 (2003)
22. Schwartz, E.J., Avgerinos, T., Brumley, D.: All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In: 2010 IEEE Symposium on Security and Privacy (SP), pp. 317–331. IEEE (2010)
23. Steffen, M.: A small-step semantics of a concurrent calculus with goroutines and deferred functions. In: Abraham, E., Bonsangue, M., Johnsen, E.B. (eds.) Theory and Practice of Formal Methods: Essays Dedicated to Frank de Boer on the Occasion of His 60th Birthday. LNCS, vol. 9660, pp. 393–406. Springer, Heidelberg (2016)
24. Stolz, V., Bodden, E.: Temporal assertions using AspectJ. Electron. Notes Theor. Comput. Sci. **144**(4), 109–124 (2006)
25. Summerfield, M.: Programming in Go (2012)
26. The cover story - The Go Blog. https://blog.golang.org/cover. Accessed 29 Apr 2016
27. Volpano, D., Irvine, C., Smith, G.: A sound type system for secure flow analysis. J. Comput. Secur. **4**(2–3), 167–187 (1996)