Gotcha:☆ Static Taint Analysis for Go

Ka I Pun[a], Martin Steffen[a], Volker Stolz[a,b], Anna-Katharina Wickert[c], Eric Bodden[d,e], Michael Eichberg[c]

[a] *Dept. of Informatics, University of Oslo, Norway*
[b] *Western Norway University of Applied Sciences, Norway*
[c] *Technische Universität Darmstadt, Germany*
[d] *Paderborn University, Germany*
[e] *Fraunhofer IEM, Germany*

Abstract

The Go programming language combines well-known imperative features with useful, well understood concurrency primitives based on channels. We discuss the state of the art of information flow analyses, and combine techniques from monotone frameworks with aliasing analysis, context-sensitive analysis of function calls, and message passing-based concurrency. An example illustrates how these components play together to obtain a reasonably precise taint analysis. We apply our analysis to the well-known scenario of SQL injections and compare our approach with two other tools for Go.

Keywords: taint analysis, static analysis, programming languages, Go language, concurrency

1. Introduction

The Go language [11, 13, 47], a relative newcomer to the programming languages stage, has gained traction in recent years. With concurrency features in
5 the form of channels and so called goroutines built in at the core of the language, chief application areas include network applications, webservers, or distributed software in general. Especially for cloud computing and virtualization frameworks, Go plays out its strengths, with the Docker container framework as one

☆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".

Email addresses: `violet@ifi.uio.no` (Ka I Pun), `msteffen@ifi.uio.no` (Martin Steffen), `vsto@hvl.no` (Volker Stolz), `anna-katharina.wickert@stud.tu-darmstadt.de` (Anna-Katharina Wickert), `eric.bodden@uni-paderborn.de` (Eric Bodden), `eichberg@informatik.tu-darmstadt.de` (Michael Eichberg)

prominent example realized in Go. Backed by Google, Go is also speculated to be the future language for Android and iOS development, and a corresponding framework and library *gomobile* [14] has been published as part of Go 1.5.

Whether on web-facing servers or soon increasingly on mobile phones, Go programs are exposed to "hostile" environments. Both are constant targets for attacks by hackers and malicious applications, which may try to break into the system through specially crafted input or interactions. SQL-injections (and similar attacks) are a well-known form of this kind of vulnerability, taking the crown on OWASPs (Open Web Application Security Project) top ten vulnerabilities.[1] A further security challenge for programs in networked or virtualized platforms is to prevent leakage of sensitive data such as passwords or privacy-related data to the outside world or unauthorized third parties. In both violations — unchecked malicious input reaching points in an application where it can wreak havoc and undesired output of sensitive data from the program — certain kind of data originating from some point in the program (unchecked malicious input, privacy information . . .) end up at other points where it is not supposed to. Not all data is considered "dangerous" or "sensitive" in the sense described; data whose unrestricted flow through points in the program violates safety or security restrictions is generically called *tainted,* the rest untainted. Which particular pieces, resp. origins and usage of information are considered tainted depend on the application.

The corrupting dangers of undisciplined and uncontrolled data flow have been known since ages, and countermeasures are well-known, too, such as programming practices like sanitizing all input ("never trust user input or generally input from the outside, always check for well-formedness"). The problem persists nonetheless. Furthermore, modern language features like high-level control structures, non-local control flow and in particular concurrency make it hard for a programmer to track data through a program and to foresee and cater for all imaginable interactions.

Automatic methods tracing data to mitigate the mentioned ill-effects are known as *taint analyses,* a variation or application of the concept of data flow analysis. An effective way to perform that type of analysis is at compile time, i.e., *statically.* We report on the design and implementation of a static taint analysis of Go programs (and implemented in Go). In the conceptual presentation we focus on the more interesting and challenging features of the Go language, such as *channel-based* communication of concurrent goroutines and deferred execution. This analysis identifies 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. It can be seen as the foundation for a monitoring framework to thwart such attacks.

The work here is an extended version of [5], giving a more detailed account of the analysis and in particular describing and evaluating the *Gotcha* analysis tool. The tool has been developed in the context of the master thesis [51].

[1] see `https://www.owasp.org/index.php/Top_10_2013-A1-Injection`

Related work

Static analyses for information flows have been widely studied: Denning and Denning [9, 10] 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 [30], which is also the starting point for our approach. Andrews and Reitman [2] 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. [50] formulate Denning's work in the form of a type system for a core imperative language; Pottier and Simonet [40] 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 ideas: Arzt et al. [3] describe and implement a static taint analysis for Java-based Android applications. The analysis is formulated as an IFDS problem. IFDS ("interprocedural finite distributive subset") by Reps et al. [41] is a prominent example of the *functional* approach to context-sensitive, interprocedural data flow analysis (the alternative is based on call-strings) [43]. IFDS is efficient and precise for *distributive* transfer functions, where being precise refers to the fact that the solution of such a problem coincides with the solution given by the "meet-over-all-*valid*-paths (MVP). It reduces the interprocedural flow problem to graph reachability in a compact representation of the interprocedural flow, called *exploded supergraph.* IFDS based taint analysis has also been used for other languages, like for JavaScript by Guarnieri et al. [15] and for web-applications in Java by Tripp et al. [48]. Like the work presented here, the TAJ tool (taint analysis for Java) of [48] operates on an SSA intermediate representation and is field-sensitive. IFDS handles data flow domains with subset lattices of the form 2^D. IDE ("inter-procedural distributive environment transformer") [42] is an extension by Sagiv et al., dealing with contexts that are of the type $Var \to D$ ("environments", called abstract states in this paper), where Var are variables of the program being analyzed and D the domain of abstract values. The domain is required to be a finite lattice, which can be lifted to domains of environments.

Livshits and Lam propose a variant of SSA to discover bugs in C programs [23], and use a context-sensitive pointer alias analysis to detect security violations in Java applications [24]. Pistoia et al. [34] 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 [22] and JFlow [29], 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; Section 3 presents the concepts of our analysis for Go

programs; Section 4 illustrates our implementation with examples, and finally Section 6 concludes the paper.

2. Preliminaries

2.1. The Go language

At its core, Go is a lexically scoped, concurrent, imperative language with higher-order functions, supporting object-oriented design, while notably not supporting classes nor inheritance. Concerning concurrency, Go's primary feature is *asynchronous* function (resp. method) calls executed as *goroutines* which are 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 *channel* communication, in the tradition of calculi resp. languages like CSP [17, 18] or Occam [19]. Since (references to) channels can be sent over channels, Go allows "mobile channel" flexibility for communication as known from the π-calculus [26].

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 since version 1.8 supports a static single assignment (SSA) intermediate format [8, 28, 35] to facilitate flow analyses. Shared variable concurrency is featured by Go but frowned upon. The more dignified and recommended way of concurrent programming is 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 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 [9, 2] 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.

Dynamic information flow analysis attempts to detect such leaks by monitoring an application's execution. However, monitoring does not guarantee an application to be *safe*, i.e., no leakage of sensitive information for any executions at runtime. Static code analysis, however, can analyse all of a program's possible executions, and detect all control-flow dependencies. 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 by 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.

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 usually expensive to compute, and to yield appropriate precision, it must share certain design properties with the alias analysis it seeks to support. In general, 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 presentation here is based on a suitable subset of the full language, concentrating on 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 form of a static analysis that can be defined in terms of the program *source code*.

3.1. Abstract syntax

To simplify the exposition, we discuss the analysis using a simplified representation of the language, assuming for example that each statement contains at most one 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 built 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 sending to resp. receiving from a channel, return from function, and of course sequential composition of statements.

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

Table 1: Abstract syntax

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.

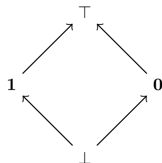
We can then restate the problem as follows: we would like to report a warning, if the return value of a function call labelled as source is assigned to a variable, and the value *may* be propagated through assignments and function calls to a variable which is used as an actual parameter in a function 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 [21], 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.

3.2. Data structures and ingredients of the taint analysis

The taint analysis as a special form of data flow analysis can be can represented as an instance of the concept of a monotone framework [20], a generic setting of program analysis operating on lattices of abstract values and states. A value in the language, i.e., a concrete piece of data is either tainted or untainted (represented as **1**, resp. **0**). Which values are considered tainted depends on the intended application. Typical data assumed initially to be tainted may be user input or passwords. In a concrete application, initial taint values are based on custom black-/whitelist of API calls, marked as either tainted or not. That leads to the 4-valued lattice $Val^\sharp$ of abstract values, with $\top$ and $\bot$ the top value

of the lattice (representing uncertainty, or an uninitialized value, respectively) and the bottom element of the lattice.



We write $\sqsubseteq$ for the partial order of the lattice, and notations $\sqsubset$, $\sqsupseteq$ etc. accord-
220 ingly; $\sqcup$ and $\sqcap$ stands for the least upper bound and the greatest lower bound, respectively.

3.3. Aliasing and channels

The Go language has several reference types, including structs, slices, arrays and maps. We do not formalize the required tracking of aliasing explicitly,
225 but assume appropriate static alias information as available in the presentation. That corresponds to the situation in the implementation, where our analysis relies on Go's built-in points-to analysis.[2] It is a context-insensitive Andersen-style points-to analysis [1], perhaps the most well known and widely used form of pointer analysis. See [45] for investigation on the complexity of this kind
230 of pointer analysis. The analysis is also known as "subset-based" analysis, to distinguish it from a variant due to Steensgard [46], which uses equality instead of inclusion and is therefore coarser. In the terminology of [27], using flow type systems, the equality-based formulation corresponds to *simple* flow analyses. See [44] for a recent, extensive overview over various pointer analyses,.

Thus we make use of the following points-to function

$$pt : Var \times Loc \to 2^{Loc} , \qquad (1)$$

235 which, given a variable at a particular location, returns an approximation of set of "objects" the variable may currently refer to. The set of actual objects is statically represented by the set of locations the objects have been allocated:

Definition 1 (May alias). *A variable x_1 at l_1 may be an alias with a variable x_2 at l_2, written as $x_1^{l_1} \sim_a x_2^{l_2}$, iff $pt(x_1^{l_1}) \cap pt(x_2^{l_2}) \neq \emptyset$. We write $x_1 \sim_a^l x_2$ for*
240 *$x_1^l \sim_a x_2^l$, comparing two variables at the same location l.*

We also simply speak of aliases (instead of may-aliases).

Another example of a reference typed data structure are, of course, Go's (typed) channels. Our treatment of assignments via transfer functions, which are defined later in the section, track the taint information associated with a
245 channel when aliasing (e.g., `ch := makeChan;ch':= ch`) because of the points-to analysis described above. Additional processing that does not follow the

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

control flow is now required when writing a tainted value into the alias `ch'`. A 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]

3.4. Taint analysis via the control-flow graph

For the *intraprocedural* part of our analysis, we can set up the monotone framework [20] with the help of the control-flow graph (CFG). We write $[s]^l$ for a statement s at location or label l. We use Loc for the infinite set of locations or labels l. In this presentation, we make the assumption that the graph is built-up as *single-instruction graph*, i.e., each node in the control-flow graph represents a single instruction. That means labelled statements $[s]^l$ are nodes of the CFG where s is an elementary, single statement. The actual intermediate representation in Go, our implementation builds upon, works with sequences of instructions of straight-line code, so-called basic blocks, as forming the nodes of the CFG. This is a commonly used improved representation; the basic principles of the analysis are unaffected by that optimization. Besides elementary statements at location l (written $[s]^l$) we also need uniquely labelled variables x^l. The 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 [30] and recapitulate the essential ingredients.

The analysis works with "abstract" values, i.e., values from the abstract domain $Val^\sharp$ which constitutes the lattice of taint values introduced in Section 3.2. An abstract state $\sigma^\sharp$ maps variables to abstract values, i.e., it is a function of type $Var \to Val^\sharp$. We write $\Sigma^\sharp$ for the set of abstract states.

The result of the taint analysis is thus given by a function of type

$$taint : Loc \to (Var \to Val^\sharp)$$

which gives for each node (respresenting an basic statement) the current taint information associated per variables in scope at the particular node (to be precise, the analysis captures the taint values at the *entry* of each particular node, as the analysis is formulated as a forward analysis).

The core of the analysis is given by interpreting, for each basic statement, the effect it has on the taint status of each variable. This is captured by the so-called *transfer functions*, translating statements case by case into ultimately a state-transformer, operating on the abstract states of type $\Sigma^\sharp$. We write $[\![_]\!]$ for the translation of statements and, in abuse of notation, use the same notation for translating expressions occurring on the right-hand sides of basic statements.

[3] See `https://go.googlesource.com/tools/+/master/godoc/analysis/peers.go`

The analysis therefore makes use of two functions of the following types:

$$\begin{array}{lcl} [\![\cdot]\!]^{\cdot}_{\cdot} & : & (Var \times Var \rightarrow bool) \rightarrow Exp \rightarrow \Sigma^{\sharp} \rightarrow Val^{\sharp} \\ [\![\cdot]\!]^{\cdot}_{\cdot} & : & (Var \times Var \rightarrow bool) \rightarrow Stmt \rightarrow (\Sigma^{\sharp} \rightarrow \Sigma^{\sharp}) \end{array} \quad (2)$$

In both cases, the first argument is a relation on variables (i.e., of type $Var \times Var \rightarrow bool$). This argument, clearly, is intended to represent the alias relationship between variables, more precisely, when applying the transfer function at a given node labelled l, the alias relation $\sim^{l}_{a}$ is used. Cases for the state-transforming transfer function, which are quite straightforward, are given in Table 2. We write $\sigma^{\sharp}_1[x \mapsto v]$ for the state $\sigma^{\sharp}_2$ which agrees with $\sigma^{\sharp}_1$ on the values of all variables, except for x, whose value is replaced by v. Note the treatment of the send-statement $ch \leftarrow x$: a channel ch is marked tainted if the value being sent into it (from variable x) is tainted, and furthermore, once tainted, the channel will not become untainted in the future, as the taint-value is monotonously increased only. That treatment is different from ordinary (non-channel) variables, which can become untainted again. In the context of tainted data and concomitant vulnerabilities, one also speaks of *sanitizing* data. In that terminology, the analysis treats channels as unsanitizable.

$$\begin{array}{rcl} [\![x := e]\!]^{\sim_a}_{\sigma^{\sharp}} & = & \sigma^{\sharp}[x \mapsto [\![e]\!]^{\sim_a}_{\sigma^{\sharp}}] \\ [\![x.f := e]\!]^{\sim_a}_{\sigma^{\sharp}} & = & \sigma^{\sharp}[x.f \mapsto [\![e]\!]^{\sim_a}_{\sigma^{\sharp}}] \\ [\![\texttt{defer}((\lambda x.s)\ v)]\!]^{\sim_a}_{\sigma^{\sharp}} & = & \sigma^{\sharp} \\ [\![\texttt{go}\ s]\!]^{\sim_a}_{\sigma^{\sharp}} & = & \sigma^{\sharp} \\ [\![ch \leftarrow x]\!]^{\sim_a}_{\sigma^{\sharp}} & = & \sigma^{\sharp}[ch \mapsto \sigma^{\sharp}(x) \sqcup \sigma^{\sharp}(ch)] \\ [\![\texttt{return}\ v]\!]^{\sim_a}_{\sigma^{\sharp}} & = & \sigma^{\sharp} \end{array}$$

Table 2: Transfer function

The abstract evaluation of possible right-hand sides is shown in Table 3. Values, including abstractions, uniformly are untainted, i.e., marked as **0**; likewise channel references freshly created via `makeChan`. In case of reference data (field access or channel variables), the evaluation builds the least upper bound of the value of all potential aliases. In the definition, evaluations of the form $[\![x]\!]_{\sigma^{\sharp}}$ or $[\![ch]\!]_{\sigma^{\sharp}}$ *without* a reference to a relation correspond to the direct look-up $\sigma^{\sharp}(x')$ or $\sigma^{\sharp}(ch)$. Note that, in case of records, the taint information does not distinguish which field contains tainted data and which not; in case a field is tainted, the corresponding record is summarily treated as tainted itself. This treatment is known as being *field-insensitive.*

Generally, the interprocedural data flow for the taint analysis is to satisfy the constraints of equation (3), where $l_1 \rightarrow l_2$ represents an edge in the cfg from l_1 to l_2.

$$[s]^{l_2} \quad \text{and} \quad l_1 \rightarrow l_2 \quad \text{implies} \quad taint(l_1) \sqsupseteq [\![s]\!]^{\sim^{l_1}_{a}}_{taint(l_1)} \quad (3)$$

$$
\begin{array}{rcl}
[\![x]\!]^{\sim_a}_{\sigma^\sharp} & = & \bigsqcup\{[\![x']\!]_{\sigma^\sharp} \mid x' \sim_a x\} \\
[\![x.f]\!]^{\sim_a}_{\sigma^\sharp} & = & \bigsqcup\{[\![x']\!]_{\sigma^\sharp} \mid x'' \sim_a x\} \\
[\![?ch]\!]^{\sim_a}_{\sigma^\sharp} & = & \bigsqcup\{[\![ch']\!]_{\sigma^\sharp} \mid ch' \sim_a ch\} \\
[\![v]\!]^{\sim_a}_{\sigma^\sharp} & = & \mathbf{0} \\
[\![v_1\ v_2]\!]^{\sim_a}_{\sigma^\sharp} & = & \begin{cases} \mathbf{1} & \text{if } v_1 \text{ is a } \textit{source} \\ \Phi^{v_1}_{exit} & \text{otherwise} \end{cases}
\end{array}
$$

Table 3: Evaluation of right-hand sides

305 *3.5. Interprocedural analysis*

Control-flow connects not only statements within the body of one procedure, but also the caller with the call. Functions typically are called multiple times in a program, and with different arguments, and interprocedural analyses allow to differentiate between different calls to the same function, depending on the 310 call-site, referred to as the calling *context*. In designing an interprocedural analysis, there is the obvious trade-off between the precision, i.e., to which extent one differentiates between calling contexts, and scalability. In the trivial case, ignoring contexts, the analysis is called context insensitive. According to the well-known classification of Sharir and Pnueli [43], one can identify two standard 315 approaches: contexts based on *call strings* (taking an abstract representation of the call stack in the form of pending function calls into account), and the *functional* approach, where different call-sites are distinguished in the analysis by different (more or less) abstract representation of their actual parameters, especially the flow information pertaining to them.

320 The functional approach has variously been implemented, investigated, and applied in different settings; also our implementation follows this approach to deal with functions and goroutines, roughly similar to the value context approach of Padhye and Khedker [31] (implemented in Soot [49]).

Calls to `go`-routines can be handled through an additional control flow-edge 325 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 330 a sink marked as possibly tainted on the entry to the statement?
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 335 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 $x.f := e$. 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* ($x \mapsto \mathbf{1}$). Otherwise, we derive the taint information of the called function with *interprocedural analysis* as shown in Table 3.

The function $\Phi^{v_1}_{exit}$ is integrated with the worklist algorithm as developed by Padhye and Khedker [31], which we have implemented to achieve a flow- and context-sensitive analysis. The idea is to distinguish 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 $X := (S, \textit{actual param})$, where S is the parent context, 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 through channel communications. Sending values or variables to a channel $ch \leftarrow x$ will propagate the taint information of x to ch (c.f. Table 2). To read from a channel $?ch$, we have to gather the knowledge of all the possible aliases of the channel to which tainted data may be sent as defined in Table 3. The statements, including finalizers (`defer`) and initiators of concurrent execution (`go`), do not affect the taint information.

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)[4] [8], interprocedural call-graph[5] , 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 uses values and performs computation, it implements the `Instruction` interface.

[4] `https://godoc.org/golang.org/x/tools/go/ssa`
[5] `https://godoc.org/golang.org/x/tools/go/callgraph`

Through the fact that we consider the distinction between calling contexts, we need to differ whether a node is a function call or not. For this aim, we use the `CallInstruction` interface that allows 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 illustrate our current taint analysis with the program in Figure 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. The functions `os.File.Read` and `fmt.Print` are a *source* and a *sink*, respectively.

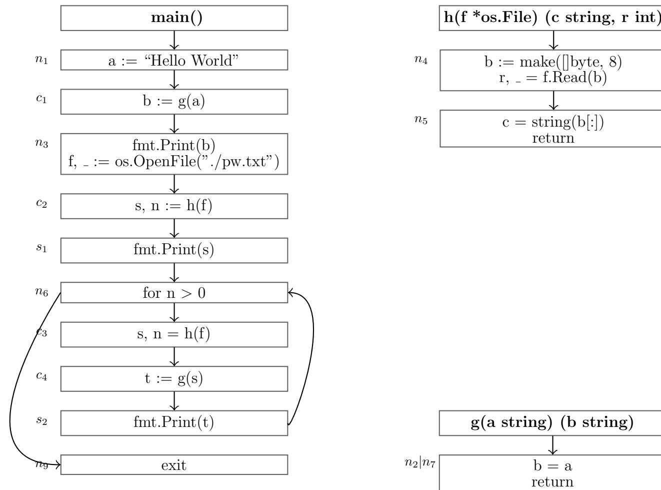


Figure 1: A simplified control flow graph with different contexts

To analyse the example and find all possible (tainted) flows, we use the transfer functions and the lattice described in Section 3. The entry point of

our analysis is the main function within a Go program, where all variables are initialised with $\perp$ (we call it "empty", for short). Thus the lattice at this point is empty and the worklist contains nodes $n_1, c_1, n_3, c_2, s_1, n_6, c_3, c_4, s_2$ and n_9. 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 Section 3. 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 processed, and the exit lattice of X_1 is $[a \mapsto \mathbf{0}], [b \mapsto \mathbf{0}]$ 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 s_1 for processing and detects that a `tainted` value reaches a sink. Our analysis will report a warning, and proceed with n_6 in the same context. 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 function 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 s_2 and contains again a call to a sink, which consequently reports a warning. Since all contexts with a matching entry parameter already exist, the remaining nodes/iterations do not lead to the creation of new contexts, but only additional transitions.

4.2. Concurrency in Go

Concurrency is at the core of the Go language [13], supporting message passing and asynchronous execution of functions (in the form of goroutines). That is, Go encourages use of channels for message passing instead of shared variables and our analysis does not cover shared memory communication. Channels, however, as far as their treatment for data-flow is concerned, bear some resemblance to reference-typed shared variables: they are dynamically created, first-class reference data, and sending to resp. reading from is analogous to writing and reading from shared variables. Data flow and the possible propagation of tainted values may now involve channel communication (besides intra-

procedural and inter-procedural flow via parameter passing). Our treatment of data flow through message passing resembles conceptually the one in Pascual and Hascoët [33]. In contrast to their work, we cover the corresponding data flow by additional edges, whereas they work on a control flow graph without such edges; instead, the corresponding additional flow is treated by adjusting the propagation algorithm appropriately, dealing with send-statements in a special way, namely by restarting the data propagation when solving the data flow equations.

In addition to the intra-procedural edges and inter-procedural edges —from caller to the entry of a callee and furthermore, for standard, synchronous calls, back from the exit of the callee to the caller— we add another class of edges covering data flow via channel communication between goroutines. Since shared variable communication is discouraged in Go, the taint analysis does not attempt to track tainted data through shared memory. The analysis assumes that, when spawning a new goroutine, the channel it uses for communication are handed over explicitly as actual parameters (or later communicated via message passing). The buffered nature of a channel regarding the fact that it basically is used analogously to a shared variable makes it necessary that it cannot be treated the same as ordinary values. Handing over a currently *untainted* channel to a goroutine does not mean it can be treated as untainted in the body of the callee. This is due to the fact that the parent of the goroutine, running in parallel, may well communicate tainted data (directly or indirectly) to the child after handing over the arguments in the asynchronous call. This delay is captured in the control flow graph by *additional* edges and additional flow functions. The edges connect the nodes containing channel send statements $c \leftarrow x$ to the asynchronous call to a function using the channel as argument.

The flow function covering the corresponding new edge is given by

$$\lambda\sigma^\sharp.\sigma^\sharp[\vec{x} \mapsto \vec{\mathbf{0}} \mid \vec{x} = dom(\sigma^\sharp) \setminus \{c\}] \tag{4}$$

i.e., it "untaints" all variables (setting them to $\mathbf{0}$) except the channel variable, on which it acts as identity.

The program in Figure 2 illustrates the approach. The main program (with the graph on the left) and the function f executed in a separate goroutine share the channel c, which is handed over from main program as argument. Variable x, originally untainted, is updated with a taint value, which is then sent from the parent to the child goroutine. Via this communication, a tainted value may reach the sink in the instance of f's body (and in absence of other communicating partners reading from channel c, it will reach it). Note also, that the tainted value will *not* reach the sink involving x in the main program. That is due to the fact that the transfer function from equation (4) connected to the loop-back edge erases all taint information (except the one relevant for the channel variable c.

Unless specified otherwise, channels are "bidirectional" in the sense that a goroutine in possession of a channel can use it for sending as well as for receiving values. This fact is the reason why the channel-send edges are added from

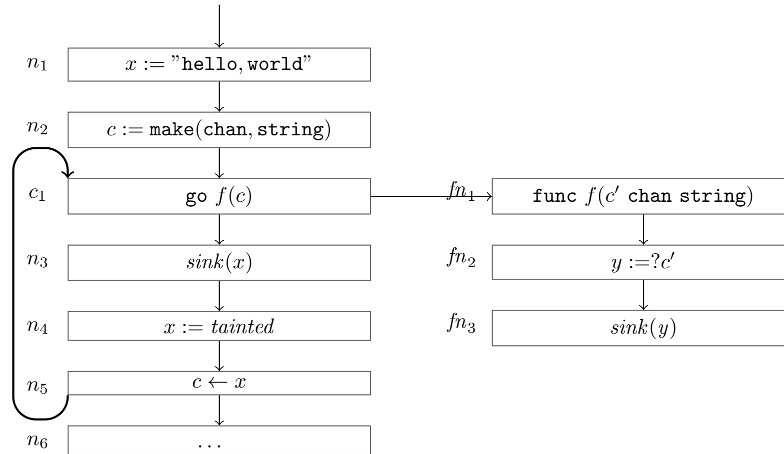


Figure 2: Goroutines and channel communication.

the sender nodes to the nodes with corresponding nodes initiating concurrent activities on a channel, as opposed to the entry node of the function body, like in Figure 2, from n_5 back to c_1, as opposed to fn_1. If function f would do a send on the channel, this may influence the statements in the main procedure following node c_1 (though not in this example).

Go's type system allows to restrict channel usages (as send-only or receive only. With the information, that c is handed over to f as *read-only* channel, the graph of Figure 2 could be improved by having the channel-send edge starting from n_5 to go to fn_1, instead of looping back to c_1.

5. Evaluation

In this section we compare our tool *Gotcha* with two other particular analysis tools for Go programs. These two tools support a specific form of a taint analysis, namely a check against the well-known SQL injection pattern [6]. This is a common vulnerability in applications that execute queries against an SQL database where the query string is directly taken (or constructed from) user input.

The *SafeSQL*[6] tool identifies all methods of the package `database/sql` having a string parameter with the name "query". For a function call without a static parameter, the tool assumes that the function is vulnerable to a SQL injection.

The *GAS — Go AST Scanner*[7] tool, which we will call *GAS* in the remainder of the section, has a rule-based approach. They offer two rules for detecting SQL injections, which we summarize as follows:

[6] `https://github.com/stripe/safesql`
[7] `https://github.com/GoASTScanner/gas`

The first rule matches against calls of the function *fmt.Sprintf*. For every call of this function, *GAS* checks whether the first parameter matches a regular expression. The regular expression will succeed if the string contains SQL keywords like SELECT, DELETE, INSERT and UPDATE. If both conditions are true, *GAS* will report a warning. The second rule does not check against a specific function call. Instead the check is against a concatenated string build up by a string conforming the above describe pattern and a non constant value.

Our tool *Gotcha* is not specifically an SQL injection checker, and hence is more similar to the rule-based *GAS* than the single-purpose *SafeSQL*-tool. We can achieve detecting potentially dangerous use of SQL effect by choosing sources and sinks for this particular problem accordingly: as sinks we take the various functions from the `database/sql` package that are actually able to execute a query on the underlying database. Listing 1 shows the necessary definitions used in our experiment. Each declaration consists of a function signature with arguments and return type, (qualified) name, and the indication whether it is a sink, i.e. any tainted input will be reported, or a source, i.e. marked as producing tainted output.

```
<func(query string, args ...interface{}) (*database/sql.Rows, error);
   (*database/sql.DB).Query> → _SINK_
<func(query string, args ...interface{}) *database/sql.Rows;
   (*database/sql.DB).QueryRow> → _SINK_
<func(args ...interface{}) (*database/sql.Result, error );
   (*database/sql.DB).Query> → _SINK_
<func(format string, a ...interface{}) (int, error );
   fmt> → _SINK_
<func() string;
   github.com/akwick/sqinco/sqlInjections.source> → _SOURCE_
```

Listing 1: Sources and sinks from the case study

Table 4 shows the example programs that we have analysed with all three tools. Each sample encodes a particular scenario, such as string concatenation with or without a constant argument (the former is unsafe if it depends on user input, the latter is always safe). For details on the examples, we refer the reader to [51] and the accompanying repository with the experimental setup at `https://github.com/akwick/sqinco` ("SQL Injection Comparison"). We have provided a virtual machine image via Docker that allows for easy reproduction of our experiment.

Execution times. As the example programs are quite small and do not require analysis of the imported libraries, analysis time is not an issue. However, analysis times rapidly become an issue when a program with all its imports from the standard libraries is being analysed. Currently, only careful pruning of imports through a runtime configuration keeps the analysis tractable. A single descent

Test feature	*safesql*	*gas*	*gotcha*	Expected result
constant variables	✗	✗	✗	✗
variable with hard-coded string	✓	✓	✗	✗
variable is return value of a function call	✓	✓	✓	✓
flow-sensitive changes so that the variable has a hard-coded string	✓	✓	✗	✗
constant variable to QueryRow	✗	✗	✗	✗
example above with hard-coded string	✓	✓	✗	✗
wrapped function	✓	✗	✗	✗
variable is result of Sprintf	✓	✗	✗	✗
query string with lower letters	✓	✗	✗	✗
non-query with string containing SELECT	✗	✓	✗	✗

Table 4: The results for the examples

e.g. into the frequently used string functions from the `fmt` package incurs virtually intolerable analysis times (also see future work on how we intend to tackle this problem by pre-computing/caching analysis results for e.g. the standard
560 library).

6. Conclusion

In this paper we present 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,
565 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
570 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.[8]

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

575 *Future work.*

We see possible future work in several directions: improving the static analysis, and refining static intermediate results through dynamic checks.

Execution time of our analysis is one of the potentials for improvement. As reported above, the analysis will currently descend into the standard library and 580 analyse all invoked functions in a new run of the *Gotcha*, even though usually the libraries do not change. We are currently working on persisting intermediate analysis results for library code, essentially lifting the reuse of results for existing value contexts to another level.

The specification mechanism for sources and sinks (see Section 5) can also 585 be made more fine-grained, as currently we do not have the possibility to specify that, for example, a function *propagates* the taint-value from an argument to one of its outputs. This is relevant in the aforementioned setting where we want a modular analysis with summaries and use pre-computed results (instead of re-analysing identical code between runs), and for specifying the behaviour of 590 opaque functions that we do not have the source code of, e.g. from compiled libraries. Likewise, a specification mechanism for *sanitizers* is useful, for example, to indicate that a hash-function turns tainted input into untainted output. Ideally, we hope to achieve a useful degree of flexibility, though most likely not going as far as, for instance, the SPLINT tool for the C language [12], which even 595 offers a way of specifying user-defined lattices.

Also, 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 600 single framework, we may benefit from some synergy. Leveraging available type information about channels as hinted at at the end of Section 4.2 may also yield some improvements

Our implementation does not cover all language features yet. It is unclear how higher-order functions (and their application) could be analysed successfully 605 at compiled time. Therefore, runtime monitoring may prove as an effective solution there.

Another – not necessarily competing – school of thought implements *dynamic* taint tracking, by attaching meta data at runtime, and checking it when reaching sinks, like in the Perl language. There, a tainted flow to a sink leads to an 610 exception. Improving dynamic tracking by reducing runtime overhead through improved placement of runtime checks has already been mentioned above, such as Livshits et al. [23, 24].

A very interesting approach that does not require any additional monitors would be to integrate tighter with the Go runtime system: it already contains a 615 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. Some of the

620 performance penalties could possibly avoided by using definite results from the static analysis phase and disable runtime checking in parts that have been proven safe.

Another interesting idea is to actively influence scheduling to avoid tainted paths. If we assume the example from Figure 2, the potential leak will only 625 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 630 capacity to consume a tainted item without becoming blocked. This problem is closely related to deadlock avoidance in schedulers [7]. Though theoretically interesting, it would mostly apply in scenarios with multiple readers (from a single channel, unlike publish/subscribe), which we do not deem that likely and hence relevant in practice.

635 ## References

[1] Andersen, L. O., 1994. Program analysis and specialization for the C programming language. Ph.D. thesis, DIKU, University of Copenhagen, available as DIKU report 94/19.

[2] Andrews, G. R., Reitman, R. P., Jan. 1980. An axiomatic approach to infor- 640 mation flow in programs. ACM Transactions on Programming Languages and Systems 2 (1), 56–76.

[3] Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Traon, Y. L., Octeau, D., McDaniel, P., 2014. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. 645 In: ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). pp. 259–269.

[4] Black, A. P. (Ed.), 2005. Proceedings of the Conference on Object-Oriented Programming Systems, Languages (ECOOP 2005). Vol. 3586 of Lecture Notes in Computer Science. Springer.

650 [5] Bodden, E., Pun, K. I., Steffen, M., Stolz, V., Wickert, A.-K., Oct. 2016. Information flow analysis for Go. In: [25], pp. 431–445.

[6] Clarke-Salt, J., Jun. 2009. SQL Injection Attacks and Defence, first edition Edition. Elsevier.

[7] Coffman Jr., E. G., Elphick, M., Shoshani, A., Jun. 1971. System deadlocks. 655 Computing Surveys 3 (2), 67–78.

[8] Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., Zadeck, F. K., 1991. Efficiently computing static single assignment form and the control dependence graph. ACM Transactions on Programming Languages and Systems 13 (4), 451–490.

660 [9] Denning, D. E., 1976. A lattice model of secure information flow. Communications of the ACM 19 (5), 236–242.

[10] Denning, D. E., Denning, P. J., Jul. 1977. Certification of programs for secure information flow. Communications of the ACM 20 (7), 504–513.

[11] Donovan, A. A. A., Kernighan, B. W., 2015. The Go Programming Language. Addison-Wesley.
665

[12] Evans, D., Larochelle, D., 2002. Improving security using extensible lightweight static analysis. IEEE Software 19 (1), 42–51.

[13] Go, 2016. Effective Go – the Go programming language.
URL `https://golang.org/doc/effective_go.html`

670 [14] Go, 2017. Gomobile. `https://godoc.org/golang.org/x/mobile/cmd/gomobile`.

[15] Guarnieri, S., Pistoia, M., Tripp, O., Dolby, K., Teilhet, S., Berg, R., 2011. Saving the world wide web from vulnerable JavaScript. In: Proceedings of ISSTA'11. pp. 177–187.

675 [16] Hind, M., Diwan, A. (Eds.), 2009. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)'09. ACM.

[17] Hoare, C. A. R., 1978. Communicating sequential processes. Communication of the ACM 21 (8), 666–677.

[18] Hoare, C. A. R., 1985. Communicating Sequential Processes. Prentice-Hall.

680 [19] Jones, G., Goldsmith, M., 1988. Programming in occam2. Prentice-Hall International, Hemel Hampstead.

[20] Kam, J. B., Ullman, J. D., 1977. Monotone data flow analysis frameworks. Acta Informatica 7, 305–317.

[21] Kobayashi, N., 2005. Type-based information flow analysis for the π-calculus. Acta Informatica 42 (4-5), 291–347.
685

[22] Livshits, B., Chong, S., 2013. Towards fully automatic placement of security sanitizers and declassifiers. In: [36], pp. 385–398.

[23] Livshits, V. B., Lam, M. S., 2003. 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. ACM Press, pp. 317–326.
690

[24] Livshits, V. B., Lam, M. S., 2005. Finding security vulnerabilities in Java applications with static analysis. In: Proceedings of the 14th Conference on USENIX Security Symposium. SSYM'05. USENIX Association, pp. 18–18.

[25] Margaria, T., Steffen, B. (Eds.), Oct. 2016. 7th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISOLA'16). Vol. 9952 of Lecture Notes in Computer Science. Springer Verlag.

[26] Milner, R., Parrow, J., Walker, D., Sep. 1992. A calculus of mobile processes, part I/II. Information and Computation 100, 1–77. URL `http://www.lfcs.informatics.ed.ac.uk/reports/89/ECS-LFCS-89-85/`

[27] Mossin, C., 1997. Flow analysis of typed higher-order programs. Ph.D. thesis, DIKU, University of Copenhagen, technical Report DIKU-TR-97/1.

[28] Muchnik, S. S., 1997. Advanced Compiler Design Implementation. Morgan Kaufmann Publishers.

[29] Myers, A. C., 1999. JFlow: Practical mostly-static information flow control. In: [39], pp. 228–241.

[30] Nielson, F., Nielson, H.-R., Hankin, C. L., 1999. Principles of Program Analysis. Springer Verlag.

[31] Padhye, R., Khedker, U. P., 2013. Interprocedural data flow analysis in Soot unsing value contexts. In: Proceedings of the 2nd ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis (SOAP'13). ACM Press, pp. 31–36.

[32] Palsberg, J., Su, Z. (Eds.), 2009. SAS'09. Vol. 5673 of Lecture Notes in Computer Science. Springer Verlag.

[33] Pascual, V., Hascoët, L., 2012. Native handling of message-passing communication in data-flow analysis. In: Forth, S., Hovland, P., Phipps, E., Utke, J., Walther, A. (Eds.), Recent Advances in Algorithmic Differentiation. Vol. 87 of Lecture Notes in Computational Science and Engineering. Springer Verlag, pp. 83–92.

[34] Pistoia, M., Flynn, R. J., Koved, L., Sreedhar, V. C., 2005. Interprocedural analysis for privileged code placement and tainted variable detection. In: [4], pp. 362–386.

[35] Pop, S., Dec. 2006. The SSA representation framework: Semantics, analysis, and GCC implementation. Ph.D. thesis, Ecole des Mines de Paris.

[36] POPL'13, 2013. 42th Symposium on Principles of Programming Languages (POPL). ACM.

[37] POPL'95, Jan. 1995. Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL) (San Francisco, California). ACM.

[38] POPL'96, Jan. 1996. 23rd Symposium on Principles of Programming Languages (POPL) (St. Petersburg Beach, Florida). ACM.

[39] POPL'99, 1999. 26th Symposium on Principles of Programming Languages (POPL) (San Antonio, TX). ACM.

[40] Pottier, F., Simonet, V., 2003. Information flow inference for ML. ACM Transactions on Programming Languages and Systems 25 (1), 117–158.

[41] Reps, T., Horwitz, S., Sagiv, M., Jan. 1995. Precise interprocedural dataflow analysis via graph reachability. In: [37], pp. 49–61.

[42] Sagiv, M., Reps, T., Horwitz, S., 1996. Precise interprocedural dataflow analysis with applications to constant propagation. Theoretical Computer Science 167, 131–170, a preliminary version of this paper appeared in Proc. of FASE 95, LNCS 915, Springer.

[43] Sharir, M., Pnueli, A., 1981. Two approaches to interprocedural analysis of parallel programs. In: Muchnik, S. S., Jones, N. D. (Eds.), Program Flow Analysis: Theory and Applications. Prentice-Hall, Ch. 7, pp. 189–234.

[44] Smaragdakis, Y., Balatsouras, G., Apr. 2015. Pointer analysis. Foundations and Trends Programming Languages 2 (1), 1–69.

[45] Sridharan, M., Fink, S. J., 2009. The complexity of Andersen's analysis in practice. In: [32], pp. 205–221.

[46] Steensgard, B., Jan. 1996. Points-to analysis in almost linear time. In: [38], pp. 31–42.

[47] Summerfield, M., 2012. Programming in Go. Addison-Wesley.

[48] Tripp, O., Pistoia, M., Fink, S. J., Sridrahan, M., Weisman, O., 2009. TAJ: Effective taint analysis of web applications. In: [16], pp. 87–97.

[49] Vallée-Rai, R., Hendren, L., Sundaresan, V., Lam, P., Gagnon, E., Co, P., 1999. Soot – a Java optimization framework. In: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research (CASCON 99). pp. 125–135, iBM Centre for Advanced Studies Conference.

[50] Volpano, D., Irvine, C., Smith, G., 2009. A sound type system for secure flow analysis. Journal of Computer Security 4 (2), 167–187.

[51] Wickert, A. K., Jan. 2017. Information flow analysis in Go. Master's thesis, TU Darmstadt, submitted for defense in December 2016.