ModInf: Exploiting Reified Computational Dependencies for Information Flow Analysis

Jens Van der Plas[1][a], Jens Nicolay[1][b], Wolfgang De Meuter[1][c] and Coen De Roover[1][d]
[1]*Software Languages Lab, Vrije Universiteit Brussel, Pleinlaan 2, Brussels, Belgium*
{*jens.van.der.plas, jens.nicolay, wolfgang.de.meuter, coen.de.roover*}*@vub.be*

Keywords: Information Flow Control, Data Flow Analysis, Taint Analysis, Static Analysis, Modular Analysis

Abstract: Information Flow Control is important for securing applications, primarily to preserve the confidentiality and integrity of applications and the data they process. Statically determining the flows of information for security purposes helps to secure applications early in the development pipeline. However, a sound and precise static analysis is difficult to scale. Modular static analysis is a technique for improving the scalability of static analysis. In this paper, we present an approach for constructing a modular static analysis for performing Information Flow Control for higher-order, imperative programs. A modular analysis requires information about data dependencies between modules. These dependencies arise as a result of information flows between modules, and therefore we piggy-back an Information Flow Control analysis on top of an existing modular analysis. Additionally, the resulting modular Information Flow Control analysis retains the benefits of its modular character. We validate our approach by performing an Information Flow Control analysis on 9 synthetic benchmark programs that contain both explicit and implicit information flows.

1 INTRODUCTION

Information Flow Control (IFC) is the practice of detecting and preventing undesirable flows of information in an application to preserve certain security properties of the application and the systems it runs on. IFC can be used to preserve confidentiality, integrity and availability, by disallowing secret or sensitive information to flow to public 'sinks' and by disallowing untrusted data to end up at sensitive sinks like a query evaluator. Unwanted flows can be detected by a static information flow analysis (e.g., (Zanotti, 2002; De Bleser et al., 2017)) that tracks information as it moves between sources and sinks. The most well-known static IFC analysis is taint analysis.

Static information flow analysis can be applied early in a software development pipeline, but analysing non-trivial applications is challenging with respect to scalability and precision. To improve scalability, modular analysis can be performed, where instantiations of modules (or 'components') are analyzed separately. In non-trivial applications, components can be inter-dependent. For example, when treating functions as components, a function can call another, or functions can access and modify shared resources. The component dependency graph that arises results of data flow between the different components.

In this paper, we exploit the insight that modular analyses depend on data (or information) flow to handle inter-component dependencies, and that this can be the basis for a modular information flow analysis. We adapt and extend ModF (Nicolay et al., 2019), a generic modular static analysis for higher-order, imperative programs, into an information flow analysis usable for IFC. ModF is function-modular, its components correspond to function calls, and soundly manages inter-component dependencies by tracking interactions of components with the store (or heap).

IFC also requires tracking dependencies, not only *between* components – like modular analysis – but also *inside* components. In this paper, we propose to reuse the dependency tracking mechanism of a modular analysis that we extend to track data flows within components and through the program under analysis. We explain our approach by extending ModF to a modular information flow analysis we call ModInf. ModF is generic and simple enough to represent other modular analysis modelling inter-component dependencies based on heap or other effects.

[a] https://orcid.org/0000-0002-7475-576X
[b] https://orcid.org/0000-0003-4653-5820
[c] https://orcid.org/0000-0002-5229-5627
[d] https://orcid.org/0000-0002-1710-1268

MODINF is capable of detecting information flow as a result of both data dependence (explicit information flow) and control dependence (implicit information flow). We implemented a prototype of MODINF and validated it on benchmark programs containing a mix of language features (assignment, higher-order functions,...), and different explicit and implicit flows.

2 BACKGROUND

MODINF, is situated at the confluence of IFC and (modular) static analysis, which we introduce here.

2.1 Information Flow Control

The goal of Information Flow Control (IFC) (Hedin and Sabelfeld, 2012; Scull Pupo et al., 2018; Russo and Sabelfeld, 2010) is to detect and prevent flows of information that decrease the security of an application or the systems it runs on, e.g., to preserve properties such as confidentiality, integrity, and availability.

Information enters an application at one or more *sources* and leaves the application at various *sinks*. Conceptually, the information that appears at sources is tagged with a particular value, depending on the type of source and the security properties to be verified. When preserving confidentiality, sources tag their values e.g., with a label indicating its confidentiality level. Tags are drawn from a join semilattice (Denning, 1976); when multiple tags have to be joined, a unique least upper bound is always defined. The simplest example of such a lattice only has two elements: H, denoting highly-sensitive information, and L, denoting the sensitivity of the information is low. This lattice has the partial order $L \sqsubseteq H$; the join operation is defined as $L \sqcup L = L$ and H otherwise.

Tags are propagated as the program manipulates information. There are two dimensions along which this happens: data dependence (giving rise to explicit information flow) and control dependence (giving rise to implicit information flow).

2.1.1 Explicit Information Flow

When new information is derived from previous information, this information is *data-dependent* on the existing information. In this case, the newly derived information has as label the unique least upper bound of the set of tags belonging to the existing information. This flow is called an 'explicit' information flow.

Consider e.g., the following simple Scheme program that reads two values from user input and prints their sum. The values of `x` and `y` thus come from sources and can be labeled accordingly. The value of `z` carries as label the join of the labels of `x` and `y`.

```
(define x (read))
(define y (read))
(define z (+ x y))
(display z)
```

2.1.2 Implicit Information Flow

When the derivation of new information is dependent on a condition (also information), then the new information is *control-dependent* on that condition, and it is tagged with the label of the condition. This flow is called an 'implicit' information flow.

The following code exemplifies such a flow. Based on user input, the value of the variable `result` may change. Thus, although no information flows directly from `input` to `result`, the value of the latter still depends on the value of the former.

```
(define result #f)
(define input (read))
(if (> input 0) (set! result #t))
```

2.1.3 Declassification and Endorsement

Joining different tags of values due to data and control dependencies results in monotonically-increasing tags: derived information can never be less sensitive, more trustworthy, etc. than any of the information from which it was derived. However, at some point programs do have to release some information, and to do so safely specific operations can be applied that result in a 'decrease' of a tag (Chong and Myers, 2004).

Take the example of preserving the confidentiality of information using tags that indicate the information secrecy level. Before storing some data in a database, an application could first encrypt this data, resulting in encrypted data that has a lower secrecy level than the original data. In the context of preserving confidentiality, such an encryption function is called a 'declassifier'. Similarly, when protecting the integrity or availability of systems and applications, 'sanitizers' or 'endorsers' in information flows have the ability to decrease the untrustworthiness of information.

2.2 Modular Static Analysis

Static analysis is used to determine semantic properties of programs without actually executing those programs (Cousot and Cousot, 1977). Typically, static analyses over-approximate the actual run-time behaviour to be sound and terminate within a reasonable time, where 'reasonable' depends on the context. This, however, means that results may be imprecise:

the analysis models all possible run-time behaviour, but potentially also behaviour that can never occur.

The relation between speed, precision, and soundness is complex (Andreasen et al., 2017): small changes in any of these aspects may result in large, unpredictable changes in results and the ability of the analysis to produce useful answers. The main challenge is to produce useful answers in a fair amount of time. Much research has been focused on techniques for increasing static analysis performance without negatively impacting other criteria such as precision too much or at all (e.g., (Might and Shivers, 2006)).

Modularization is one technique to improve analysis performance (Cousot and Cousot, 2002; Nicolay et al., 2019). A modular analysis analyses parts of the program separately, and composes the results to obtain information about the entire program. These parts are referred to as *modules*. At runtime, multiple instantiations of these modules may exist, which the analysis may distinguish as well. For example, a given function can be called multiple times. The reifications within the analysis of these instantiations, called *components*, are analysed in isolation. Modules can vary from coarse-grained (e.g., a thread definition (Stiévenart et al., 2019)) to fine-grained (e.g., a function definition (Nicolay et al., 2019)). The corresponding components are then a thread and a function call. A component contains the corresponding module and a context that allows more components to be distinguished, thereby increasing analysis precision.

Modular analysis can be more efficient than a whole-program counterpart for various reasons: memory consumption is reduced, components can be analysed in parallel (Van Es et al., 2020; Stiévenart et al., 2021), and components will not be reanalyzed due to non-relevant changes (Van der Plas et al., 2020; Van der Plas et al., 2023).

Ideally, components do not depend on each other. Yet, in all but the most trivial cases, components *do* depend on information computed by other components. A function may e.g., use the return value of another. Thus, some ordering must be obeyed so that a component is only analyzed after all its dependencies have been analyzed. Computing inter-component dependence can happen before or during the analysis (Nicolay et al., 2019). In case of a cyclic dependencies or self-dependence (e.g., due to recursion), no such ordering may exist; components may be analysed multiple times to take into account information computed by the analyses of the components it depends on. In all cases, component dependencies can be tracked using heap access: when a component writes to a memory location, any component reading this location depends on it.

2.3 Effect-driven Modular Analysis

MODF (Nicolay et al., 2019) is a type of modular analysis, expressed as an abstract interpretation (Cousot and Cousot, 1977), computing essential control and value flow properties that found many other non-trivial static analyses. Its fixed-point computation consists of two alternating phases. An *intra-component analysis* analyses a component in isolation. Doing so, it infers *effects* representing its interactions with the global store σ and the discovery of other components. (MODF uses global store widening (Shivers, 1991), i.e., a single value store is shared among all components.) An *inter-component analysis* uses these effects to decide which components to analyse using a worklist. For example, if a component reads at an address in σ (indicated by a *read effect*), it becomes dependent on it. If this address is later written to and the value at the address changes (indicated by a *write effect*), all dependent components must be reanalysed. Thus, the inter-component analysis can then add all dependent components to its worklist.

MODF is function-modular: modules are functions and components represent function calls. When a function call is encountered, the analysis does not step into this function but generates a *call effect* for the component representing the call in the analysis. If this component has not yet been analysed, the inter-component analysis schedules it for analysis. The analysis then retrieves the return value of the component from σ (or $\perp$ if it had not yet been analysed).

MODF reaches a fixed point as soon as the worklist is empty. It can be used with different representations of abstract values and different context sensitivities. The latter allows multiple components to be used for a single function, increasing the precision of the analysis. There can e.g., be a different component for every calling location of a function. Effect-driven analyses can also be created for other module granularities such as threads (Stiévenart et al., 2019).

2.3.1 Example of a MODF Analysis

We exemplify MODF using the program in Listing 1, visualising its analysis in Figure 1. When representing values by their type and using no context sensitivity, the program in Listing 1 is analysed as follows:

1. The Main component, representing the program entry point, is analysed. Upon the definition of `num`, a write effect is generated. When the call to `plus-n` is encountered, a new component is created and added to the worklist. As no return value for `plus-n` was computed yet, $\perp$ is retrieved from σ and a read effect is registered on this return value. Finally, a return value is written to σ.

Listing 1: Example program.

```
(define num 0)
(define (plus-n n) (set! num (+ n num)) (print))
(define (print) (display (string-append "current:"
                         (number->string num))))
(plus-n 10)
```

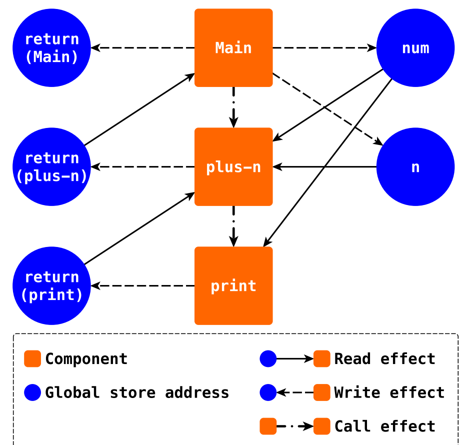

Figure 1: MODF analysis of the program in Listing 1.

2. The plus-n component is analysed. Reading `n` and `num` generates read effects. Writing `num` generates no write effect as the value of `num` in σ remains unchanged (`Int`). As a call to `print` is encountered, a new component is created and added to the worklist. After retrieving its return value ($\perp$) from σ, the return value of plus-n is written to σ.
3. The print component is analysed. Reading `num` generates a read effect. Then, the return value of print (`void`) is written to σ. As a write effect is generated, plus-n is added to the worklist again.
4. The analysis continues until the worklist is empty, indicating that a fixed-point has been reached.

3 APPROACH

We exploit the inter-component dependencies of a modular analysis, which model the inter-component information flow of the application under analysis, as a basis for an IFC analysis. We extend the modular analysis to also compute intra-component information flow information, thus obtaining the complete information flow, containing both explicit *and* implicit flows, that can be used to perform taint analysis.

The benefit of our approach is that it does not require the design of an analysis with a specific taint lattice. Instead, we use the data flow information that already exists within a modular analysis and complete it with the intra-component data flow. In particular, we reuse the mechanism of the modular analysis for tracking inter-component dependencies based on access to shared resources, and extend it so that intra-component dependencies are tracked as well. Typically, this means that only modest changes to the analysis are required. The resulting taint analysis also inherits (and benefits from) the modularity, context sensitivity, lattice implementations, or any other property or mechanism of the underlying analysis from which it was derived. Our approach thus does not depend on these properties or mechanisms, nor does it require additional effort in designing and implementing them.

MODF represents the essence of a modular analysis: is does not prescribe how the intra-component analysis must be performed, but only specifies two constraints which enbody two essential ingredients of any modular analysis: finite per-component analysis and tracking of inter-component dependencies using effects occurring on shared resources. Thus, while we base our work on MODF, our approach can be instantiated with any modular analysis that contains these two ingredients and exposes information about them.

We assume the existence of three constructs to provide information about taint in a program:

- `(source x)` returns the value of the variable `x` and marks this as originating from a source;
- `(sink x)` indicates that the value of the variable `x` has reached a sink, and also returns this value;
- `(sanitize x)` returns the value of the variable `x` but indicates that this value has been sanitized.

In a real-world setting, existing (library) functions for interacting with the outside world would play the role of sources, sinks, and sanitizers. We abstract them here to keep our approach general, and also for practical (testing) purposes (Section 6).

4 FROM MODF TO MODINF

We chose MODF as the analysis to instantiate our approach with, because it is a straightforward, generic and modular flow analysis that models inter-component dependencies. The resulting modular IFC analysis is called MODINF.

In addition to *inter-component* data flow, MODINF also has to track *intra-component* data flow. It does so by extending the inter-component data flow tracking offered by MODF. MODF infers inter-component data flow by detecting how values flow between addresses in the analysis store σ. The store maps addresses to values, and represents the heap of

the analysis. Consequently, MODINF infers intra-component data flow during the intra-component analysis using the same mechanism of monitoring store operations. In short, whenever the analysis of a component reads a value from a certain address in the store, this value is labeled with the address. This means that addresses piggy-back on top of values as they flow through the analyzed program. In the next sections, we describe how MODINF handles both types of information flow (explicit and implicit) in more detail.

4.1 Explicit Information Flow

Explicit information flow arises due to value flow in the program. When a program is executed, values are propagated through its operations, and values are read from and stored in the heap. Similarly, during an analysis, abstract values are propagated through abstract operations, and abstract values are read from and stored in the store.

During the analysis of a component, when values are propagated, the labels of these values need to be propagated together with the values flowing through in the analysis. Doing so allows the analysis to keep track of how values flow between addresses in the analysis store (recall that the labels indicate the addresses in the store from which a value originates).

When a value is used in a computation, its labels are added to the result of the computation; when multiple values are used in the computation, the result carries the labels of all the argument values. After all, when one value originates from address *a* and another value originates from an address *b*, then the result from an operation on both values originates from information in both *a* and *b*. Similarly, when e.g., a pointer is dereferenced, the resulting value inherits the labels of the pointer, as the resulting value depends on the value of the pointer. Thus, the labels attached to a value indicate all addresses in the store the value is influenced by/originating from.

4.2 Implicit Information Flow

Implicit information flows arrive from conditions in the program, such as branching in an `if` statement or dynamic function calls: based on a condition, a piece of code may or may not be executed, or the function to be called depends on a value. A taint analysis must take these implicit flows into account, because the conditional branching may depend on a tainted value, and the choice of function to be executed as well.

To handle these implicit flows, extra flow information is needed. In general, this information originates at the condition on which is branched (e.g., the predicate in an `if` statement): all data that impacts a condition also impacts its branches and result. Thus, when a value in a branch is written to the store, the analysis does not only take its labels into account but also the labels of the condition. These labels are also added to the result value of the conditional computation, as this is dependent on the value of the condition as well.

Consider e.g., the following program. Explicit data flow in this program arises from the parameter `n`, whose value flows to the result of `maybe-inc`. As `cond` also influences the result, the labels attached to its value are added to the return value of `maybe-inc`.

```
(define (maybe-inc n cond) (if cond (+ n 1) n))
```

A similar system is needed for dynamic function calls. To propagate implicit information flow across function boundaries, which are also component boundaries, upon a function call, the current implicit information flow labels, as well as the explicit flow labels of the function value, are ‘attached’ to the called component. When the component is analysed, these labels are considered as well when a value is written to the analysis store. However, this implies that when new labels are encountered upon a function call, the corresponding component needs to be reanalysed (so the newly added labels can be propagated during its analysis)[1]. Consider e.g., the program in Listing 2. During the analysis of `a`, the analysis infers that the argument of `f` is a boolean and discovers no implicit flows to be added to `f`. Thus, during the analysis of `f`, no implicit flows are taken into account. If then, however, `b` is analysed, an implicit flow for `f` is found. However, since the abstract value for `x` remains the same, a modular analysis would not reanalyse `f`, thus ignoring the implicit flows for `f`.

Listing 2: Reanalysis of `f` is needed to propagate labels.

```
(define (f x) (sink x))
(define (a) (f #t))
(define (b) (define v #t)
            (define v-s (source v))
            (if v-s (f #t)))
(a)
(b)
```

When conditional flows are nested (e.g., nested `if` statements), implicit flow information arising from all conditions must be taken into account as they all contribute to the path that is taken by the program under analysis. Also, control flow depends on value flow and vice versa. The value of a predicate (value flow)

[1]The authors assume that implicit flows over function boundaries can also be added after the analysis by using the inferred write and call effects, thereby avoiding these extra component analyses, but have not explored this path further.

in a conditional statement such as `if` determines e.g., which branch of the conditional is executed (control flow). If a tainted value induces control flow, then any value flow that happens as a result of this control flow must also be tainted. We say that the branch or function body is executed in a *tainted context*.

4.3 Interactions with the Global Store σ

When a value is propagated by the analysis, so are its labels. Keeping these labels attached to the values at all times is unnecessary, however, and would cause the sets of labels of each value to keep growing. These sets would also give little information on how values actually flow through the program, but only contain all addresses that may have influenced the value.

Instead of keeping labels attached at all times, we remove labels from values prior to writing to σ. As such, the store only contains values but no labels. For every address in σ, we keep track of the labels that were attached to the values written, and we merge the labels corresponding to the explicit and implicit information flow together at the time of the store write. We thus obtain a kind of data flow graph, where the edges are directed backwards. As such, the analysis tracks how values flow between addresses in σ, and thus also how values flow within the analyzed program.

4.4 Taint Derivation

The data flow information, computed as just explained, can be used as a basis for taint analysis.

When a value is marked as originating from a source, it is labeled with a specific label which can e.g., carry information indicating the type of taint. A sanitizer causes the value flow to be threaded through σ using a specific sanitization address to allow the traversal (described next) to stop looking for tainted values as sanitization removes taint. When encountering a sink, the analysis also threads the value flow through σ, using a specific sink address, allowing the traversal to find sinks as starting points for tracing.

At the end of the analysis, the data flow information can be used to detect harmful flows by tracing the data flow backwards starting from sink addresses. A sanitization address causes the trace to be abandoned as no tainted flow can originate from it. When a source label is found, however, there exists a non-sanitized flow from the corresponding source to at least one sink. Hence, a security risk may exist in the application. Our analysis then reports a tuple containing the source and sink (but could e.g., also report the entire flow path).

5 IMPLEMENTATION

We have implemented MODINF in MAF (Van Es et al., 2020), a framework for the construction of modular analyses. Our implementation can analyse Scheme programs that are enriched with the `source`, `sink`, and `sanitize` constructs presented earlier. Scheme, a dynamically-typed higher-order language, is very difficult to analyse since control flow and data flow are intertwined. The concepts introduced in this work can therefore be transferred to other highly-dynamic languages, like JavaScript, Java, and C++.

Only minor changes were needed to extend an existing modular analysis in MAF with intra-component data-flow information. We extended the representation of abstract values so that labels can be piggy-backed. When values are joined, the union of the sets carried along those values is computed. When an operation is applied to values, the result is labeled with the union of the label sets of the arguments. When an abstract pointer is dereferenced, the obtained value is labeled with the labels of the pointer.

To propagate implicit flow information across function boundaries, we store for every component a set of all implicit taints that were present during any call to that component. The analysis of a component considers this set as being part of the implicit flows and adds it to the implicit flows caused by conditionals upon every write to the analysis store.

6 VALIDATION

We performed a preliminary validation of our work using 9 hand-crafted programs that reflect the various ways in which taints may flow through a program. The goal of this validation is to ensure that our analysis finds all vulnerable flows, i.e., that it is sound, in the presence of various complex value flows. To facilitate our validation, we constructed the smallest possible programs that contain complex flows. Exploring properties of MODINF such as performance and scalability is an interesting avenue of future work.

Concretely, we considered the following 9 small programs (each between 5 and 10 LOC):

bad-flow-retrigger-needed Shown in Listing 2. Contains a flow from a source to a sink that, given the worklist algorithm used by this validation (described later), can only be found if a component is reanalysed after new implicit flows are found.

implicit-flow Contains conditional branching based on a tainted value.

sanitization-in-tainted-context Shown in Listing 3.

Sanitizes a value and feeds it to a sink in a tainted context, for which the analysis should still consider this as a harmful flow.

sanitized-flow A flow originating from a source passes a sanitizer and flows to a sink. Hence, this program should be considered safe.

side-effecting-function Calls a function in a tainted context. The called function changes the value of a variable which flows to a sink afterwards.

simple-flow A tainted value flows to a sink.

sink-in-tainted-context A tainted value is passed through a sanitizer to a sink. The flow of this value to the sink depends on the original tainted value. Hence, there is a harmful (implicit) flow.

tainted-function-choice-1 A side-effecting function to be executed is selected from a list based on a tainted value. The side-effect influences the value flowing to a sink.

tainted-function-choice-2 Shown in Listing 4. The return value of a function call flows to a sink. However, the function might have been overridden (depending on a tainted value).

Listing 3: The `sanitization-in-tainted-context` benchmark.

```
(define x #t)
(define x-s (source x))
(if x-s (let ((san (sanitize x-s))) (sink san)))
```

Listing 4: The `tainted-function-choice-2` benchmark.

```
(define a #t)
(define a2 (source a)) ;Value comes from a source.
(define (b x) x)
(define (set-b) (set! b (lambda (x) #f)))
(if a2 (set-b)) ;b depends on a2.
(define res (b 10))
(sink res) ;Result of (b 10) flows to a sink.
```

We instantiated MODINF with a type domain (representing values by their type except booleans which are represented concretely when possible; pointers, closures and primitive functions are represented using sets), without context sensitivity and a last-in-first-out worklist algorithm. (Other lattice representations, context sensitivities and worklist algorithms are possible as well.) Our analysis was able to detect all harmful flows within the programs. Thus, our preliminary evaluation shows that our analysis does not have false negatives on several small handcrafted programs containing challenging value flow. We also did not find any false positives in the results of the analysis, though this may be caused by the limited size of the programs used. Therefore, future work should evaluate the precision of our analysis using bigger programs, that are already supported by our implementation, in which false positives may arise.

7 RELATED WORK

To the best of our knowledge, there are no related approaches that describe the relation or transition between modular static analysis and static IFC analysis.

Some related static analysis approaches also use the store to determine dependence, but for other purposes. (Nicolay et al., 2011) attempts to parallelize binding expressions by computing dependencies between these expressions based on address reads and writes. (Stiévenart et al., 2015) detects concurrency bugs based on conflicts involving shared addresses. (Nicolay et al., 2017) investigates function purity based on reading and writing of store addresses visible from the point of view of all callers on the stack.

The majority of static IFC analysis approaches and implementations (see e.g. (Pauck et al., 2018) for an overview) are not modular by design. MODINF differs from most of the existing work as it does not use a state space in which a value is tagged with a taint label from a lattice. Instead, MODINF tracks address dependencies as values are read and written in the store, without propagating taint labels explicitly.

Modular (static) IFC analyses are far and few between. A notable example of a modular static taint analysis is LGTM[2], a code analysis platform developed by GitHub that is capable of performing taint analysis of modules in JavaScript applications. It achieves modularity by either not stepping into other modules, or relying on a manually provided specification of taint flows, requiring a trade-off between accuracy and effort (Staicu et al., 2020). In contrast, our approach does not rely on manual specifications.

8 CONCLUSION AND FUTURE WORK

In this paper, we introduced MODINF, a novel way to information flow analysis that leverages the inter-component data flow information inferred by a modular analysis. We extended this data flow information with intra-component data flow information and differentiate between 'explicit' information flow, indicating data dependence, and 'implicit' information flow, indicating control dependence. Using this flow information, we obtain an information flow analysis

[2]https://lgtm.com/, soon to be integrated in GitHub code scanning.

that can detect harmful flows in computer programs. We validated our approach using 9 hand-crafted programs with complex data flows and verified that all harmful flows were discovered by the analysis.

Our work shows that an information flow analysis can be obtained by making only minor changes to a modular analysis. The resulting analysis is modular, meaning that it scales well to large programs, and independent of a particular lattice or context sensitivity.

Future work may consider distinguishing different types of taint tags, e.g., to reflect levels of information sensitivity where low-sensitive data may e.g., be allowed at some sinks in a program. Another improvement would be to extend our validation and evaluation using larger programs, allowing e.g., to evaluate the precision of the analysis (i.e., the number of false positives found by the analysis). Our implementation already supports these bigger programs.

ACKNOWLEDGEMENTS

This work was partially supported by the Research Foundation – Flanders (FWO) (grant No. 11F4822N) and by the *Cybersecurity Initiative Flanders*.

REFERENCES

Andreasen, E. S., Møller, A., and Nielsen, B. B. (2017). Systematic approaches for increasing soundness and precision of static analyzers. In *SOAP 2017, Proc.*, pages 31–36.

Chong, S. and Myers, A. C. (2004). Security policies for downgrading. In *CCS 2004, Proc.*, pages 198–209.

Cousot, P. and Cousot, R. (1977). Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL 1977, Proc.*, pages 238–252.

Cousot, P. and Cousot, R. (2002). Modular Static Program Analysis. In *CC 2002, Proc.*, pages 159–179. Springer.

De Bleser, J., Stiévenart, Q., Nicolay, J., and De Roover, C. (2017). Static Taint Analysis of Event-driven Scheme Programs. In *ELS*, pages 80–87.

Denning, D. E. (1976). A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243.

Hedin, D. and Sabelfeld, A. (2012). A perspective on information-flow control. In *Software safety and security*, pages 319–347. IOS Press.

Might, M. and Shivers, O. (2006). Improving flow analyses via Γcfa: Abstract garbage collection and counting. In *ICFP 2020, Proc.*, pages 13–25.

Nicolay, J., De Roover, C., De Meuter, W., and Jonckers, V. (2011). Automatic Parallelization of Side-Effecting Higher-Order Scheme Programs. In *SCAM 2011, Proc.*, pages 185–194. IEEE.

Nicolay, J., Stiévenart, Q., De Meuter, W., and De Roover, C. (2017). Purity analysis for JavaScript through abstract interpretation. *Journal of Software: Evolution and Process*, 29(12).

Nicolay, J., Stiévenart, Q., De Meuter, W., and De Roover, C. (2019). Effect-driven Flow Analysis. In *VMCAI 2019, Proc.*, pages 247–274. Springer.

Pauck, F., Bodden, E., and Wehrheim, H. (2018). Do android taint analysis tools keep their promises? In *ESEC/FSE 2018, Proc.*, pages 331–341.

Russo, A. and Sabelfeld, A. (2010). Dynamic vs. static flow-sensitive security analysis. In *CSF 2010*, pages 186–199. IEEE.

Scull Pupo, A. L., Christophe, L., Nicolay, J., Roover, C. d., and Gonzalez Boix, E. (2018). Practical information flow control for web applications. In *RV 2018, Proc.*, pages 372–388. Springer.

Shivers, O. (1991). *Control-Flow Analysis of Higher-Order Languages*. Doctoral dissertation, Carnegie Mellon University, Pittsburgh, PA, USA.

Staicu, C.-A., Torp, M. T., Schäfer, M., Møller, A., and Pradel, M. (2020). Extracting taint specifications for javascript libraries. In *ICSE 2020, Proc.*, pages 198–209.

Stiévenart, Q., Nicolay, J., De Meuter, W., and De Roover, C. (2015). Detecting Concurrency Bugs in Higher-Order Programs through Abstract Interpretation. In *PPDP 2015, Proc.*, pages 232–243.

Stiévenart, Q., Nicolay, J., De Meuter, W., and De Roover, C. (2019). A general method for rendering static analyses for diverse concurrency models modular. *Journal of Systems and Software*, 147:17–45.

Stiévenart, Q., Van Es, N., Van der Plas, J., and De Roover, C. (2021). A parallel worklist algorithm and its exploration heuristics for static modular analyses. *Journal of Systems and Software*, 181:111042.

Van der Plas, J., Stiévenart, Q., and De Roover, C. (2023). Result Invalidation for Incremental Modular Analyses. In Dragoi, C., Emmi, M., and Wang, J., editors, *VMCAI 2023, Proc.*, volume 13881 of *Lecture Notes in Computer Science*, pages 296–319. Springer.

Van der Plas, J., Stiévenart, Q., Van Es, N., and De Roover, C. (2020). Incremental Flow Analysis through Computational Dependency Reification. In *SCAM 2020, Proc.*, pages 25–36. IEEE Computer Society.

Van Es, N., Stiévenart, Q., Van der Plas, J., and De Roover, C. (2020). A Parallel Worklist Algorithm for Modular Analyses. In *SCAM 2020, Proc.*, pages 1–12. IEEE.

Van Es, N., Van der Plas, J., Stiévenart, Q., and De Roover, C. (2020). MAF: A Framework for Modular Static Analysis of Higher-Order Languages. In *SCAM 2020, Proc.* IEEE Computer Society.

Zanotti, M. (2002). Security typings by abstract interpretation. In *International Static Analysis Symposium*, pages 360–375. Springer.