Design Your Analysis

A Case Study on Implementation Reusability of Data-Flow Functions

Johannes Lerch Ben Hermann
Technische Universität Darmstadt, Germany
{lastname}@cs.tu-darmstadt.de

Abstract

The development of efficient data flow analyses is a complicated task. As requirements change and special cases have to be considered, implementations may get hard to maintain, test and reuse. We propose to design these analyses regarding the principle of separation of concerns. Therefore, in this paper we present a reference design for data flow analyses in the context of the IFDS/IDE algorithm. We conducted a case study in order to inspect the level of reuse that can be achieved with our design and found it to be helpful for the efficient development of new analyses.

Categories and Subject Descriptors F3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program analysis

General Terms Design

Keywords Separation of concerns, static analysis, design, ifds, ide, flow function

1. Introduction

The development of static analyses is a challenging task. It is – like most software development tasks – an incremental process. New requirements arise and existing requirements change during the development and in most cases it is not obvious how the final result will look like. So, learning from the software engineering field, separation of concerns and reusability are key to efficiently develop these analyses.

Naturally, it is impracticable to start implementing an analysis from scratch. Analysis frameworks such as Soot [9], WALA [4] or OPAL [5] provide basic functionality to be reused by specialized analyses. To aid the analysis developer's task these frameworks provide abstractions and intermediate representations to avoid the need of dealing with low-level problems. On top of that, framework-like algorithms like the IFDS [7] and IDE [8] algorithms provide means to divide large problems into smaller parts enabling developers to focus only on analysis problem specific tasks.

However, the amount of work to develop a specialized analysis still is tremendous. The considerations that have to be made for the analysis have to be encoded in the abstractions of the framework it is build upon. As soon as the analysis grows in its implementation it may become hard to maintain. Something that started as a prototype becomes too complicated to change. Features tend to become intertwined making the reuse of an analysis feature a challenging task, although many aspects of static analyses are identically handled across multiple analyses addressing different problems.

We found this to be the case while developing FlowTwist [6], a taint-flow analysis for the detection of so-called Confused Deputies in Java. In this paper we therefore contribute a design approach that effectively separates different analysis aspects and implementations that are otherwise often interwoven. We focus on IFDS and IDE analysis problems, discussed on examples using the Soot framework and Heros implementation [2] of the IDE algorithm.

In a case study we found this design to be very helpful to reuse analysis components and efficiently take on novel analysis problems.

The remainder of this paper is organized as follows. In Section 2, we present the current design approach takes for IFDS/IDE analyses. Section 3 outlines the proposed design and compares it to the current approach. We discuss the details and advantages of our design in Section 4. We briefly inspect related work in Section 5 and conclude the paper with a summary in Section 6.

2. State-of-the-Art

Data-flow analyses can be approached with the IFDS [7] and IDE [8] algorithms. These algorithms provide a framework to address many analysis problems by expressing them as a graph-reachability problem and therefore take away a large amount of implementation work from the developer of the analysis. Both algorithms require an interprocedural control flow graph, which can be easily provided by most static analysis frameworks, e.g., Soot. To define the actual analysis problem developers provide so-called *flow functions*. Flow functions define the effects that specific edges in the interprocedural control flow graph have on an incoming fact. A fact can be anything that needs to be tracked in the specific analysis, e.g. a value coming from user input. A flow function may generate new facts, pass over existing facts or specify that the incoming fact will not hold at the edges destination, commonly referred to as killing the fact.

For example, consider a simple data flow analysis that tracks values created at a source statement, e.g. a user input, and checks if these values flow to a sink statement, e.g. a database command. When the flow function is called to evaluate an assignment statement where a tracked value is on the right hand side of the assignment, the flow function should return the incoming fact, because the variable of the right hand side is still carrying the tracked value, as well as a new fact representing the variable of the left hand side of the assignment, because it is now carrying that value as well. In the opposite case, when a variable gets overwritten with a constant,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

SOAP'15, June 14, 2015, Portland, OR, USA
© 2015 ACM. 978-1-4503-3585-0/15/06...$15.00
http://dx.doi.org/10.1145/2771284.2771289

```java
public interface FlowFunctions<N, D, M> {
  FlowFunction<D> getNormalFlowFunction(N curr, N succ);
  FlowFunction<D> getCallFlowFunction(N callStmt, M destinationMethod);
  FlowFunction<D> getReturnFlowFunction(N callSite, M calleeMethod, N exitStmt, N returnSite);
  FlowFunction<D> getCallToReturnFlowFunction(N callSite, N returnSite);
}
```

```java
public interface FlowFunction<D> {
  Set<D> computeTargets(D source);
}
```

Figure 1. Flow Function interfaces of the Heros framework to be implemented by specific analyses

the flow function should kill the fact representing the variable on the left hand side of the assignment.

Heros [2] is an open-source implementation of the IFDS and IDE algorithms. In Heros there are two interfaces that have to be implemented in order to specify these flow functions (c.f. Figure 1). The first interface is `FlowFunctions`, which provides instances of the second interface `FlowFunction` that handle specific edges in the interprocedural control flow graph. Edges are categorized by this procedure into normal edges, call edges, return edges, and call-to-return edges. Normal edges represent intraprocedural flow like assignments, conditions and loop. Call edges span from a call site to the first statement of a called method. Return edges reach from an exit point of the method back to the call site the method call originated from. Call-to-return edges span over the call.

This way IFDS and IDE algorithms allow the analysis developer to focus on the effects of each edge in isolation and therefore describe the behavior of the analysis very naturally. Nevertheless, even in this small focus on a single edge the implementation can become very complex with growing requirements to handle more and more specific situations.

In the example above for a complete analysis, its developer has to handle many more features of the language aside from assignments. This includes reading and writing instance fields and static fields, cast expressions, boxing and unboxing of types, exceptions, reflection, mapping of actual parameters to formal parameters at call edges and return values at return edges. Moreover, it might be too costly to precisely analyze the behavior of data structures that are used in the program, therefore the effects of these might be addressed by specification. This is commonly the case for arrays and collections. Depending on the type of analysis, it might also be necessary to handle the construction and concatenation of Strings, for instance using the StringBuilder class of Java. Furthermore, when not analyzing the complete program including its used libraries and the runtime library, flow functions have to account for the effect on facts passed over calls to library functions. Similarly, there might be definitions summarizing the effects of functions implemented in other languages, e.g. native code. Besides handling all these language features, flow functions also need to be aware of special function calls in the context of the analysis. For instance, an analysis tracking user input might kill facts when input data is treated with a special sanitization function.

During the development of FlowTwist [6], we went through the experience that implementations respecting all of those concerns in a single flow function become hard to maintain, difficult to test and impossible to reuse. In addition to our own experience, we looked at implementations of other developers finding that this seems to be a common result, e.g., FlowDroid [1] a data and information flow analysis for Android based on Heros contains more than 400 lines of code for handling normal flow functions alone, not counting code of called helpers. We thus propose to separate the different concerns of a flow function better and present our proposal to do this in the next section.

3. Design

We want to achieve that different concerns can be expressed in different isolated classes. Ideally, each concern being one implementation of the `FlowFunction` interface. But, it is not possible to easily separate all concerns as some concerns depend on others. For example, in a taint analysis one concern of a flow function is the propagation of facts through assignments so that facts holding at the right hand side get propagated to the left hand side. At the same time, we might have another concern that we never want to track values of primitive data types. Therefore, facts are killed if they are propagated through a cast expression or being unboxed. However, there is a dependency between these two concerns, because we do not want to propagate any fact at an assignment if a second concern claims the fact should be killed instead.

In order to separate concerns of a flow function, we introduce a new interface `Propagator` shown in Figure 2. Each implementation of that interface reflects one specific concern. Bundled together these `Propagator` implementations will behave like an implementation of the Heros interface `FlowFunction`. Thus, our approach represents a functional equivalent to the `FlowFunction` interface while at the same time separating concerns from each other. The new data structure `KillGenInfo` is used as the return type instead of a set of facts. In Heros a concern that does not affect the source fact has to return the source fact and does not return the source fact if it wants to signal the fact should be killed. We opt for a more explicit API explicitly communicating a kill of a fact. Therefore, we use the `KillGenInfo` data structure. The data structure is a named pair holding a boolean flag indicating if the source fact should be killed and a set of facts that should be generated, which does not require to include the given source fact.

To deal with dependencies between concerns, we group instances of the `Propagator` interface into multiple phases, whereas each phase can contain multiple `Propagators` as shown in Figure 3. Processing of `Propagators` of the same phase are not allowed to affect each other, while the processing of each phase depends on the result of its preceding phase. Actually, if any `Propagator` of a phase decides that the source fact should be killed, the succeeding phases will not be processed at all. But, `Propagators` of the same phase will still be processed independently, meaning `Propagators` of a single phase may be processed in arbitrary orders or even in parallel. The input fact is the same source fact for all phases and does not depend on the generated facts of preceding phases. Each `Propagator` may generate new facts. At the end of the process the union set over all facts propagated by all phases is built. Therefore, the processing in phases is *not* a processing pipeline, i.e., the input of a phase is not the output of the preceding phase. However, the execution of a succeeding phase depends on the preceding phase.

To adapt Heros-style `FlowFunctions` to our phases-propagators design, we developed the processing of phases as implementations of the `FlowFunction` interface as shown in Figure 4. Phases are defined as a two-dimensional array of type `Propagator`. The first

```
public interface Propagator<N, D, M> {
  boolean canHandle(D fact);
  KillGenInfo<D> propagateNormalFlow(D source, N curr, N succ);
  KillGenInfo<D> propagateCallFlow(D source, N callStmt, M destinationMethod);
  KillGenInfo<D> propagateReturnFlow(D source, N callSite, M calleeMethod, N exitStmt, N returnSite);
  KillGenInfo<D> propagateCallToReturnFlow(D source, N callSite);
}
```

Figure 2. `Propagator` Interface

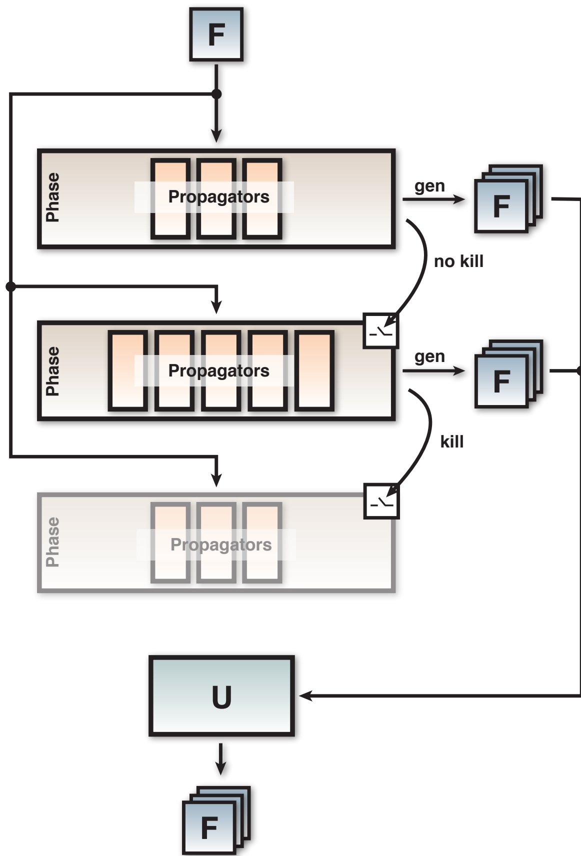

Figure 3. Overview on fact handling

```
Set<D> computeTargets(D source) {
  boolean killed = false;
  Set<D> gens = new HashSet<D>();
  for(Propagator<D>[] phase : phases) {
    for(Propagator<D> propagator : phase) {
      if(propagator.canHandle(source)) {
        KillGenInfo kgi = propagate*(source, ...);
        killed |= kgi.kill;
        gens.addAll(kgi.gens);
      }
    }
    if(killed)
      break;
  }
  return gens;
}
```

Figure 4. Processing of Phases

```
phases = new Propagator[][] {
  {
    new PrimitiveTypesKiller(),
    new PermissionCheckPropagator(),
    /* ... */
  },
  {
    new AssignmentPropagator(),
    new FieldAccessPropagator(),
    new StringBuilderPropagator(),
    /* ... */
  },
  {
    new SinkHandler(),
    /* ... */
  }
};
```

Figure 5. Phase Configuration

dimension reflects each phase while the second dimension contains the `Propagator` instances of each respective phase. The inner loop collects generated facts by the `Propagator` instances of a phase. The outer loop cycles over all phases until all of them are processed or some `Propagator` returns that the source fact should be killed.

An example configuration for a data flow analysis where the phases are implemented in form of a two-dimensional array is shown in Figure 5. In the presented example, the first phase contains `Propagators` reflecting the sanitization of data flows, i.e. it may kill facts under certain conditions. If a fact survives that phase, the second phase handles data flow propagations, i.e., assignments, field processing, calls, etc. The last phase contains a `Propagator` that is generating a report if a fact has reached a sink, i.e. the analysis found a data flow from a source statement to a sink statement.

In this example, it becomes obvious that the separation of concerns is helpful. Some of the `Propagator` implementations are very general ones that can be reused across many different analyses. For instance, the `Propagators` of the second phase can be used for any data flow analysis, while the `Propagators` in the first and last phase may be different for other analyses. It might be even the case that more (or less) phases are required for different analyses. Note that this can be done as easy as adding an element to the array. In conclusion, our design easily separates different analysis concerns, but centralizes their combination to this end that a specific analysis can be formed by constructing a phase array from existing `Propagator` implementations. Thus it allows reusing `Propagator` implementations.

4. Discussion

We initially became aware of the problem that implementations of flow functions become hard to maintain and hard to test when we started to implement FlowTwist [6], an analysis to find confused deputy problems in the Java Class Library. At some point we changed its design to the design presented here. In the following we discuss the experiences we gathered after that design change on an analysis on which we continued development for over two years already.

FlowTwist is a taint analysis that handles data flows of arbitrary types, except primitive data types, which are always filtered. This means that the propagated fact represent if variables are tainted by unsafe input data. Hence, we use the terms *taint* and *fact* as synonyms here. The analysis contains an implementation of the `Propagator` interface that handles assignments and instance fields. Static fields have to be handled differently than instance fields, therefore an additional `Propagator` is used. In FlowTwist we have to track Strings, therefore we used two separate `Propagator` implementations to handle `StringBuilder` and other operations on String, for example `String.valueOf`. The `Propagator` implementation for `StringBuilder` detects if taints are passed into a `StringBuilder` instance and generates facts now tracking the respective `StringBuilder` instance. If a `String` is generated from a tracked `StringBuilder` instance the generated value is being tracked. Using the suggested design, this rather special treatment of a specific type is encapsulated in a separate class. In a straightforward implementation this treatment would be scattered all over a flow function implementation as it requires to detect different interactions, i.e. arguments passed to `StringBuilder` objects (via call edges) as well as retrieving values from `StringBuilder` objects (via return edges).

FlowTwist also uses `Propagator` implementations very specific to its analysis problem, like the handling of source statements, i.e., initially mapping zero facts to taint facts, implementations for killing facts passed over permission checks, and lastly implementations to detect taint facts reaching sink statements.

For experiments and to evaluate multiple approaches, FlowTwist consists of multiple variations of the analysis. One configuration represents the full-featured analysis. Another smaller configuration is used in an experiment, in which less statements of interest are considered to perform experiments on the scalability. This variation does not affect how assignments have to be processed, but requires slight changes in the source and sink handling. In a straightforward implementation this variation would have to be encoded by multiple conditions checking which variation is currently used which will be scattered across the whole flow function implementation. By using the suggested design, there are just two slightly different versions of the phase configuration like the one shown in Figure 5.

Another more challenging variation configures the analysis so that it is performed only in a forward direction through the interprocedural control flow graph, while in the full-featured analysis it starts in the middle of the program performing a backward and a forward analysis. Performing the analysis backward through the interprocedural control flow graph requires a different processing of assignments, calls, and returns. While starting the analysis in the middle of the program requires a different handling of source and sink statements as well. Here again, we were able to reuse `Propagator` implementations across the variations and define each variation as a two-dimensional phases configuration array.

The reuse case within one analysis problem through variations might be a special case for the FlowTwist project. Therefore, we conducted a small case study and tried to reuse implementations for a different analysis problem. Namely, we implemented an analysis to detect SQL-Injection, Command-Injection, and XSS vulnerabilities in web applications. These problems are data flow problems in nature and therefore require handling of assignments, calls, returns, etc. As we separated the implementation for these concerns from the part specific to the analysis problem, we were able to reuse these `Propagator` implementations. Actually, it turned out that the only `Propagator` implementations we have to exchange by other implementations are the ones concerned with source, sanitization, and sink handling. This means that we were able to handle a very different analysis problem just by providing a new phase configuration. Even better, the code bases for both problems will not divert as they can be naturally kept in the same project as there is only one single place where dependencies between these are configured: the phase configuration.

While we implemented the suggested design as part of an analysis using Heros, it would also be possible to integrate the phase processing in Heros itself and provide the `Propagator` interface to the specific analysis implementations. There is one drawback that has to be considered when doing this. In the current state, Heros allows caching of `FlowFunction` implementations. The cache is used to determine if for the same edge the analysis was already asked to create a `FlowFunction` instance. Potentially, this can be used to precompute all flow functions for the whole program. Nevertheless, for a concrete fact the flow function has to be evaluated eventually. In our experience, this evaluation is the expensive processing part compared to the creation of a `FlowFunction` instance, because in most scenarios it is not possible to determine that a flow function for a specific edge will always generate some specific fact or always kill the incoming fact. Most of the times, such decisions depend on the concrete source fact passed into the flow function. In summary, at the cost of a cache lookup a class instantiation is saved on cache hit, i.e., an analysis with many cache misses might even be faster without the cache. Therefore, we recommend to integrate our proposed design allowing to separate concerns.

5. Related Work

As the requirements for static analyses steadily expand, their implementation is exposed to the risk of getting hard to maintain and to test. On the other side, developers feel the need to reuse parts of already implemented analyses in the creation of new ones. This puts a great pressure on the design of analysis implementations, which is only recent a subject in scientific literature.

With TS4J [3] Eric Bodden presents a design approach using Evans and Fowler's *fluent interfaces*. This design is used to express properties for a typestate analysis. By also implementing behavior into the fluent interfaces rather than just using it for configuration of the analysis, this approach is quite similar to our approach as analysis abstractions are also modified/created by the configuration objects.

Eichberg et al. present a design to facilitate the implementation of static analyses in a product line approach [5]. Using their OPAL framework analysis components can be combined in many different ways to adapt them to the current problem. This also separates the concerns of an analysis very efficiently and leads to an easier implementation of new analyses.

6. Conclusion

We found and discussed in detail that implementations of data-flow functions have to address many different concerns. Therefore, straightforward implementations tend to intertwine different concerns making them hard to reuse, test, and maintain. We suggested a design that separates concerns successfully and discussed in the context of a taint analysis how the design can be implemented and used to achieve maintainability and testability. In a case study we confirmed the reusability of implementations addressing concerns that are used across multiple targeted analysis problems. To provide the suggested design to everyone without the need of implementing it again and again, we close with the recommendation to integrate our design directly into the Heros framework.

Acknowledgments

This work was supported by the BMBF within EC SPRIDE, by the BMBF within the Software Campus initiative (01IS12054) and by the Hessian LOEWE excellence initiative within CASED.

References

[1] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 259–269, New York, NY, USA, 2014. ACM.

[2] E. Bodden. Inter-procedural data-flow analysis with ifds/ide and soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis*, SOAP '12, pages 3–8, New York, NY, USA, July 2012. ACM.

[3] E. Bodden. Ts4j: A fluent interface for defining and computing typestate analyses. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*, SOAP '14, pages 1–6, New York, NY, USA, 2014. ACM.

[4] J. Dolby, S. J. Fink, and M. Sridharan. T.j. watson libraries for analysis (wala). URL `http://wala.sf.net/`.

[5] M. Eichberg and B. Hermann. A software product line for static analyses: the OPAL framework. In *SOAP '14: Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*, pages 1–6, New York, NY, USA, June 2014. ACM.

[6] J. Lerch, B. Hermann, E. Bodden, and M. Mezini. FlowTwist: efficient context-sensitive inside-out taint analysis for large codebases. In *FSE 2014: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 98–108, New York, NY, USA, Nov. 2014. ACM.

[7] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 49–61, New York, NY, USA, 1995. ACM.

[8] M. Sagiv, T. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. In *Selected Papers from the 6th International Joint Conference on Theory and Practice of Software Development*, TAPSOFT '95, pages 131–170, Amsterdam, The Netherlands, 1996. Elsevier Science Publishers B. V.

[9] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot: A java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, pages 214–224. IBM Corp., 2010.