The Secret Sauce in Efficient and Precise Static Analysis

The beauty of distributive, summary-based static analyses (and how to master them)

Eric Bodden
Paderborn University & Fraunhofer IEM
Paderborn, Germany
eric.bodden@upb.de

Abstract

In this paper I report on experiences gained from more than five years of extensively designing static code analysis tools—in particular such ones with a focus on security—to scale to real-world projects within an industrial context. Within this time frame, my team and I were able to design static-analysis algorithms that yield both *largely improved precision and performance* compared to previous approaches. I will give a number of insights regarding important design decisions that made this possible.

In particular, I argue that summary-based static-analysis techniques for distributive problems, such as IFDS, IDE and WPDS have been unduly under-appreciated. As my experience shows, those techniques can tremendously benefit both precision and performance, if one uses them in a well-informed way, using carefully designed abstract domains. As one example, I will explain how in previous work on Boomerang we were able to decompose pointer analysis, a static analysis problem that is actually not distributive, into sub-problems that are distributive. This yields an implementation that is both highly precise and efficient.

This breakthrough, along with the use of a demand-driven program-analysis design, has recently allowed us to implement practical static analysis tools such as the crypto-misuse checker CogniCrypt, which can analyze the entire Maven-Central repository with more than 200.000 binaries in under five days, although its analysis is flow-sensitive, field-sensitive, and fully context-sensitive.

Keywords Static analysis, performance, precision, abstract domains, summarization

1 Precision and Performance

Designing static analyses is not an easy task. I myself have been working on and with static analyses since 2003, so a little over 15 years. In hindsight, in the early days I have made lots of beginner's mistakes, but I also perceive that still today many others are struggling as well in finding the optimal design for the particular static analysis they want to build. Because so many people struggle, one can easily get to the point of also drawing wrong conclusions, one of which I find to be the following.

> There seems to be a general perception that static analyses, as they are designed to become more precise, will also become less efficient: "precision costs performance". I have found this to be wrong. If carefully designed, the opposite becomes true: higher precision yields better performance.

This has to do with the fact that an analysis that lacks precision generally suffers from a problem in the literature sometimes described as "overtainting" [20]: due to the imprecision, the analysis will propagate information to unduly large parts of the program, in particular into program parts where that information does not belong. This is true at all levels: An imprecise call-graph analysis will cause information to flow into callees that cannot actually be called at runtime. An imprecise pointer analysis will cause information to be propagated into aliases that cannot occur at runtime, etc.

Michael Hind has previously described this problem as one of abstraction vs. approximation [2]: much previous work on program analysis in general, and as Hind points out on pointer analysis in particular, has focused on lossy approximations rather than on more clever abstractions that carefully determine which information can be ignored *safely without* impeding precision. This paper here is exactly about such abstractions.

A problem with the predominant perception "more precise = slower" is that many people have come to accept the fact that if they find their analysis to not scale, they should lower its precision, for instance by selecting a cheaper-to-compute call-graph or a cheaper-to-compute pointer analysis. In practice, however, I have found that, while this certainly buys time in the early phases of the analysis, it wastes so much

ISSTA Companion/ECOOP Companion'18 , July 16–21, 2018, Amsterdam, Netherlands
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5939-9/18/07...$15.00
https://doi.org/10.1145/3236454.3236500

time and precision later on that one eventually ends up with an analysis that is still inefficient and now also imprecise.

For this reason, in the past few years we have been consequently developing static analyses that attempt to only compromise on precision as a last resort. One important point here is that, in order to be able to conduct a precise and efficient client analysis (such as a taint analysis, typestate analysis, shape analysis, etc.), one must also make sure that the underlying pointer analysis is equally precise. This is because if that is not the case, imprecision from the pointer analysis will end up creeping into the client analysis, not only causing it to report false information but also to waste effort in computing unhelpful information.

Precise pointer analyses are key: In the past years we have been developing highly precise pointer analyses that are flow-sensitive, field-sensitive, and fully context-sensitive. The largely added precision of those analyses has helped us to drastically boost both the precision and performance of the client analyses.

2 Coping with all those "sensitivities"

One important question, of course, is how to design such analyses, both pointer and client analyses, in such a way that they actually maintain efficiency when applied to large code bases despite the many "sensitivities" I mentioned above. Context-sensitive analyses are notorious for creating millions if not billions of contexts for larger code bases, which leads to obvious efficiency problems. Flow-sensitive analyses do not compute a global program state but instead must associate potentially different analysis states with every single program statement, for possibly every single calling context, increasing the efficiency challenge even more. Field-sensitivity largely increases the analysis' abstract domain: where a field-based analysis represents both field accesses `a1.f` and `a2.f` as `A.f` (assuming both `a1` and `a2` have type `A`), and a field-insensitive analysis represents both field accesses `a.f` and `a.g` as the same `a.*`, a field sensitive analysis must keep both abstractions separate. In practice, however, I have found that such highly precise analyses can be designed to also be highly efficient if one incorporates the following ingredients.

The combination of a demand-driven analysis design and an efficient, distributive static-analysis framework can yield analyses that are both highly precise and efficient. Added performance can be achieved through concise procedure summaries.

3 Demand-driven analysis

A demand-driven analysis only analyzes parts of a program, in response to a given query. [22–25] Demand-driven analyses are not the right tool for every application context. Let us assume, for instance, that we are designing an analysis to detect potential data races. In essence, without any further information, such a race could occur at every single field-write, of which you will find thousands and thousands in large applications. In such scenarios, a demand-driven analysis would have to inspect every single such field-write, and start a demand-driven analysis from there. Those are situations in which it may actually well be simpler and faster to instead conduct a whole-program analysis in the first place, that simply computes through the entire program in one large, uniform iteration.

My group, however, has devised static code analyses in particular to uncover security vulnerabilities. As we observed, 20 out of the SANS 25 vulnerabilities (the most commonly observed vulnerabilities, as reported by SANS) [3] can be casted as taint-analysis or typestate-analysis problems. Both types of analysis have in common that they reason about API calls, and relationships between them. A taint analysis typically seeks to identify flows from a given source API to a sink API, which have not passed to an appropriate sanitization API. Typestate analyses on the other hand seek to determine whether one uses APIs incorrectly by issuing calls to objects in incorrect orders or with incorrect arguments.

But both types of analyses share another common trait: they can both benefit from very quickly "homing in" on some few calls to the relevant APIs, and then conducting a precise, demand-driven analysis from there. In practice we have found that even in situations where we have complex rule sets that concern dozens of (for instance security-relevant) classes, these classes are typically only used in small parts of the program. Finding those use sites is cheap, requiring nothing more than a syntactic pattern matching. Moreover, if the demand-driven analysis then bootstrapped from those sites is highly precise, it can restrict its own computation to only small parts of the program, typically just a few methods.

Another advantage of demand-driven analysis is also a positive effect on precision: whole-program analyses have a larger probability of encountering program constructs at which coarse-grain approximations must take place. The imprecision this causes can then propagate to other parts of the analysis. As a demand-driven analysis inspects smaller parts of the program, they are less vulnerable to this problem. [4]

This way of conducting analyses has one limitation that my team and I have learned to accept: since, in essence, the analysis tries to infer as much information as possible locally, around the sites of "interesting calls", one lacks the information about whether or not those calls will actually be reachable at runtime. In essence this means that an analysis designed that way has the tendency to report vulnerabilities also in potentially dead code. Some might perceive this as an issue, but my personal opinion is that vulnerabilities should not exist anywhere, including in dead code, as this code may become alive eventually. Large software development companies are with me on this issue, for instance SAP SE, which follows a "zero vulnerability" policy. [18]

Another caveat of demand-driven analyses is that in some corner cases the analysis for an individual query might—despite all efforts—still compute a long time. In those cases it is common to abort queries after a given time frame or budget has expired. [22–25] In those cases, faced the challenge of whether and how to nonetheless provide the client with some answer to the query. Some so-called refinement-based approaches opt for soundness in those cases, returning more imprecise results of a simpler to compute analysis. [23, 25] Others accept unsoundness by returning incomplete information.

corner cases handling

Demand-driven analysis has the benefit of being efficient, to a large extent, irrespectively of the size of the overall program. This is because, if designed precisely, it can restrict its own computation to the set of local areas in that large program that actually matter to the analysis.

4 Summary-based analysis frameworks

If we accept that demand-driven analysis is useful in some application contexts such as software security, what is the added benefit of summarization and distributivity? Procedure summaries are important for a very simple reason: they avoid the repeated re-analysis of identical code, in particular of the same procedures. For decades, ever since the foundational paper by Sharir and Pnueli [19], researchers have been making inter-procedural static analyses context-sensitive by one of the two following means:

1. In the *call-strings approach*, one basically conducts a regular context-insensitive analysis in the monotone framework [5] but lifts it to a context-sensitive one by annotating data-flow facts with context information, typically k-limited calling-context strings, i.e., sequences of method calls that resemble the call stack. The call-strings approach typically goes without summarization.
2. In the functional approach, one obtains context-sensitivity by computing, at the first time a call to a procedure p is encountered, a reusable summary resembling the effects that a call to p will have in terms of the static analysis. This summary is then applied anew in every further calling context, i.e., at every other call to p.

```
void main() {
    byte[] pri = privateKey();
    byte[] pub = publicKey();
    byte[] priAlias = foo(pri); //context c1
    byte[] pubAlias = foo(pub); //context c2
    ...
    log(pub);
}

byte[] foo(byte[] ba) {
    //some hard-to-analyze code omitted
    return ba;
}
```

Listing 1. Taint analysis benefiting from summarization

To illustrate the benefit of summarization, consider the example in Listing 1. Assume we wish to conduct a taint analysis that ought to warn us when private keys are written to log files. As we can see, the program does handle a private and public key, and logs a key but that key is actually the public one. To make that distinction, however, as both the private and public key are passed to `foo`, one requires a context-sensitive analysis. Such an analysis can be conducted both using the call-strings and the functional approach.

In the call-strings approach, one would analyze the procedure `foo` twice, once for context `c1` and one for context `c2`. While this would yield the correct, precise result "`pri` does not leak to `log`", it wastes computation: irrespective of what parameter object `foo` is called with, it returns that same object. If we assume that `foo` has no further side-effects, but does contain some hard-to-analyze code, then we would end up analyzing `foo` twice, without gaining anything, but wasting precious analysis time. Analyzing a method twice does not cost that much time, but due to method-call nesting, such an approach can easily incur an exponential blowup, which is the reason for why also current papers that describe approaches built on top of the call-strings approach typically report the need to create millions if not billions of contexts for realistic program. Every single such context means the re-analysis of a potentially hard-to-analyze procedure!

Now next let us consider the summary-based approach instead. There are many summary-based approaches, including IFDS [14], IDE [17], WPDS [15] and VASCO [13], but if used correctly, all of them would behave the same on this simplified example. Upon encountering the first call site `c1`, the analysis would analyze `foo` but summarize its effects. For the sake of this example let us assume that the summary is simply:

$$\texttt{ba} \longmapsto \texttt{<ret>}$$

This summary indicates that the procedure returns its argument `ba`, *no matter what that argument actually is.* Importantly, this summary is reusable: upon encountering call site `c2`, the analysis will map also the argument object `pub` to `ba`, and would see that a summary for `ba` has already been computed. It would then simply apply that summary at `c2`, propagating information from `pub` to `pubAlias`, without having to re-analyze `foo`.

By avoiding the repeated re-analysis of callee procedures for different calling contexts, the use of procedure summaries can drastically reduce the complexity of the static analysis.

Note that procedure summaries such as the one above actually represent summary *functions*. Such functions can have representations that are either *extensional* or *intensional*. An extensional summary defines a function by associating inputs with outputs. An extensional definition of the increment-function could look like this:

$$0 \mapsto 1, 1 \mapsto 2, 2 \mapsto 3, \ldots$$

As can easily be seen, extensional summary definitions have at leats two problems: First they are very verbose, making it hard to store them compactly. Second, to be finitely representable at all, they typically must be restricted to a finite subset of inputs.

An intensional definition of the increment function simply looks like this:

$$\lambda x.\ x \mapsto x + 1$$

Note the following differences: Firstly this definition is much more compact than the extensional one. Secondly, it represents all possible inputs to the function. This is because the intensional definitions abstracts from concrete parameters through its input variable x, and explicitly describes the function's effect on x.

When designing procedure summaries, one should strive for intensional (opposed to extensional) summary definitions, as they are compact and represent the entire possible input space.

The summary "`ba` ↦ `<ret>`" for the example from Listing 1 is compact because it is effectively intensional: concrete input variables such as `a1` and `a2` are abstracted away and compactly represented by `foo`'s formal parameter `ba`.

In the past, many fellow researchers and I myself have unconsciously neglected this guideline, resulting in summaries that were effectively extensional. As I will explain in the following, this causes problems with summary reuse.

When applying summary-based techniques such as IFDS, IDE, WPDS and VASCO to static-analysis problems blindly, without the proper understanding of the summarization effects, this can quickly void all advantages that summarization would normally give. One reason for this is that in such cases summaries are frequently extensional.

As an example, let us consider the example in Listing 2. Assume that this time we wish to conduct a flow-sensitive and context-sensitive points-to analysis. Typical points-to analyses use so-called store-based abstractions [6], i.e., model objects through allocation sites. The example contains two such sites: `alloc1` and `alloc2`.

```
14  void main() {
15      A a1 = new A(); //alloc1
16      A a2 = new A(); //alloc2
17      A a1Alias = foo(a1);        //context c1
18      A a2Alias = foo(a2);        //context c2
19  }
20
21  <T> T foo(T o) {
22      //some hard-to-analyze code omitted
23      return o;
24  }
```

Listing 2. Pointer analysis benefiting from summarization

Unfortunately, when propagating allocation-site information through the program blindly, with a summarizing algorithm such as IFDS, IDE, WPDS and VASCO, this can easily destroy all summarization. For the example, assume that before line 17 we are tracking the two pieces of information (`a1`, {`alloc1`}) and (`a2`, {`alloc2`}), denoting that the objects referred to by `a1` and `a2` could have been created by `alloc1` and `alloc2` respectively, i.e., points-to any object created at the respective statement. Now upon encountering the first call to `foo` at `c1`, the analysis would analyze the callee procedure and would, in fact, create a summary, indicating the following mapping:

$$(\texttt{o}, \{\texttt{alloc1}\}) \mapsto (\texttt{<ret>}, \{\texttt{alloc1}\})$$

Note how this summary contains not only information that is local to the callee `foo` (the identifier `o` and the return value `<ret>`) but also information that is context-dependent: the allocation site `alloc1` matters in this particular calling context `c1` and (in this program) no other. This is a bad smell.

Procedure summaries should contain *only* information that has a local meaning to the procedure being summarized. Typically this information relates to variables and pointers that are in scope at this place. Store-based heap abstractions [6] are not a good fit, as they tend to propagate non-local information.

The problem with the above summary is that—despite being a valid summary—it is not very much reusable. It can only be reused in cases where `foo` again is called with *the very same* points-to set. Even in cases where individual elements were added or removed from the points-to set the summaries would not match, and could not be reused. This is also for a good reason, as any changes to the points-to set could change the effect that the call to `foo` has on the computation, which should then actually result in a different summary.

This is also what would happen in the example in Listing 2: At `c2`, the analysis would analyze this time the procedure `foo` with an initial analysis information of (`o`, {`alloc2`}), which does *not* match the left-hand side of the already-created summary—due to the differing points-to sets. Hence, the analysis would re-analyze foo again, essentially resulting in an equivalent, yet still slightly different, second summary:

$$(\texttt{o}, \{\texttt{alloc2}\}) \mapsto (\texttt{<ret>}, \{\texttt{alloc2}\})$$

The important point about summary-based analyses is not to create summaries but to reuse them. Hence, much care must be taken to use abstract domains that are likely to yield reusable summaries.

As alluded to earlier, one key problem with those two summaries is that they are effectively extensional: by naming alloc1 and alloc2 they are explicitly enumerating the procedure's input space (and hence also that of the summary function).

5 Storeless heap models with access paths

The definition of an appropriate abstract domain really is the key to making summary-based analyses useful. In the above, example the use of a store-based heap model, incorporating allocation sites, was obviously not a good idea, since it causes the allocation sites to be propagated throughout the program, to places where they do not belong, where they have no meaning: within procedure foo, the statements alloc1 and alloc2 are not even in scope, they have no meaning there.

The trick is therefore to find different representations, in particular *heap* representations, that allow for *local reasoning*. Luckily, a number of such representations exists, and they all fall into the category of what others have previously called *storeless* heap representations. [6] Essentially, one way or the other, all those representations encode what in the literature is known as *access paths*. An access path has the following form:

```
l.f.g.h
```

Here l is a local variable, and f.g.h is an example of a sequence of field accesses through which one can access the object in question, given l. The length of an access path is given by the length of the sequence of fields. For example, the access path l.f.g.h has length of 3, the access path l with an empty field sequence has length 0.

An important invariant that heap representations based on access-paths should enforce is that all access paths are locally valid: for each access path rooted in a local variable l, this access path should only ever be associated with statements at which l is in scope.

The above property is a litmus test for any flow-sensitive static-analysis implementation: if you find yourself propagating access paths to where they are out of scope then you are very likely doing something wrong.

Now with access paths in mind, let us again consider the example from Listing 2. In recent work we have designed, implemented and evaluated a flow-sensitive pointer analysis called Boomerang, that for the first time uses a heap abstraction solely comprising locally meaningful information, in the form of access paths. In the example, the analysis would, at c1, represent the object pointed to by a1, and therefore also pointed to by the formal parameter o, not by its points-to set but instead solely by the access path o. This then causes the summary-based analysis to instead create the following concise summary:

$$\texttt{o} \mapsto \texttt{<ret>}$$

Note that this summary *is* reusable. At call site c2, the analysis will map a1 to o as well, and this o matches the left-hand side of the summary rule, causing the analysis to skip any further inspection of foo, and to directly apply the summary instead. Note how this summary also comprises method-local information only: both o and <ret> are in scope within foo.

Access paths have one inherent limitation: to obtain a finite domain, their length must be bounded. This can be achieved through simple k-limiting. A better approach, however, is to encode access paths through so-called access graphs. [7] Access graphs are essentially finite-state machines whose induced regular language represents an infinite set of access paths. This allows one to efficiently represent access paths for instance of the form l.(next.prev)*, which is obviously very beneficial when dealing with recursive data structures.

My previous explanation of the example in Listing 2 left open one important question: some client analyses require information not just about pointer relations but actually also about allocation sites, for instance to determine the possible runtime type of a pointer in call-graph analysis. In Boomerang this is achieved by actually creating a new IFDS solver for each allocation site, while the various IFDS solvers all share the same set of procedure summaries. This provides access to the allocation sites where required.

6 Distributive analysis frameworks

An important limiting factor of summary-based static analyses is that not all static analyses *can* be compactly summarized. In general, a callee procedure can apply arbitrarily complex operations to the values the static analysis reasons about, and in arbitrarily complex combinations of those operations. As soon as loops or recursion are involved, it might thus not only be hard but impossible to summarize just any effect that a callee procedure may have. The static-analysis framework VASCO does allow one to conduct a context-sensitive analysis by creating summaries for any static-analysis problem that fits the monotone framework, but those summaries always comprise the entire abstract state reaching the callee procedure: if D is the domain of data-flow facts, then VASCO computes summaries of the form $2^D \rightarrow 2^D$. The problem with this is that as soon as even the tiniest fragment of that

```
void main() {
    Pair<String> p1 =
        makePair(secretKey(),secretKey());
    Pair<String> p2 =
        makePair(secretKey(),publicKey());
}

<T> T makePair(T a, T b) {
    Pair<T> ret = new Pair<T>();
    ret.left = a;
    ret.right = b;
    return ret;
}
```

Listing 3. Example illustrating the advantage of pointwise summaries

abstract state, i.e., an element within the input set, changes from one call to the next, the computed summary cannot be reused any more, hence necessitating a re-analysis of the callee procedure, and the creation of another summary for the newly encountered abstract state.

6.1 The effect of distributive flow functions

Reps et al. have shown with IFDS that one can very efficiently create procedure summaries in cases where the analysis is distributive over the merge operator, i.e., where for all flow functions f and abstract domain values x, y it holds that $f(x) \sqcup f(y) = f(x \sqcup y)$, and where the problem is a subset problem, i.e., where $\sqcup$ is defined as set union. For problems that have this property, one gains the big advantage of being able to produce independently reusable pointwise summaries of the form $D \rightarrow D$:

To understand the effect this has, consider the example in Listing 3, again with a taint analysis tracking the value of the `secretKey()`. At the first call to `makePair`, an approach such as VASCO, that does not assume distributivity, would create a procedure summary such as:

$$\{\texttt{a,b}\} \mapsto \{\texttt{<ret>.left,<ret>.right}\}$$

This summary indicates that after the call both `<ret>.left` and `<ret>.right` are tainted, as {a,b} were tainted before the call. Now at the second call to `makePair`, the abstract inputs to the callee are different: this time only the first parameter is tainted, but not the second parameter `b`. Because the summary is not point-wise, it can only be reused as a whole, which in this case is not applicable. VASCO would hence have to re-analyze the callee, producing a second summary:

$$\{\texttt{a}\} \mapsto \{\texttt{<ret>.left}\}$$

IFDS, IDE and WPDS are frameworks that assume distributive flow functions, but thereby allow the creation of point-wise summaries. In the same example, at the first call to `makePair`, such frameworks would create two independent summaries, as follows:

$$\{\texttt{a}\} \mapsto \{\texttt{<ret>.left}\}$$
$$\{\texttt{b}\} \mapsto \{\texttt{<ret>.right}\}$$

Now at the second call to `makePair`, where only `a` is tainted but not `b`, the analysis can nonetheless reuse the first of the two summaries: it states that as soon as `a` is tainted, so becomes `<ret>.left`—independently of everything else.

Distributive frameworks have the advantage of being able to produce point-wise summaries that are highly reusable, because summaries can be reused on the level of individual elements of the abstract domain.

6.2 Distributive framework = precise solution

But distributive frameworks have another big advantage: optimal precision. The optimal solution to any static analysis problem is the so-called meet-over-all-paths solution, also called MOP. [5] In practice, this solution can *usually* not be computed: Rice has shown that any non-trivial static analysis problem is undecidable, and hence its MOP-solution cannot be computed. [16] This is exactly why practical static analyses approximate the MOP solution through a maximal-fixed-point solution, also called MFP. The MFP solution is hereby guaranteed to be a sound over-approximation:

$$\text{MOP} \sqsubseteq \text{MFP}$$

This relationship is guaranteed by the requirement that within the monotone framework that is used to compute the MFP solution all flow functions must be monotone: for all flow functions f and abstract domain values x, y it holds that:

$$f(x) \sqcup f(y) \sqsubseteq f(x \sqcup y)$$

As already explained above, for any distributive analysis problem the following stronger condition holds:

$$f(x) \sqcup f(y) = f(x \sqcup y)$$

From this it also follows directly that for such problems:

$$\text{MOP} = \text{MFP}$$

This is very important to note:

If one can manage to precisely encode a problem in a distributive framework, then one can solve it with full precision!

6.3 A distributive encoding of pointer analysis

Unfortunately, though, not all problems are distributive. Problems generally become non-distributive when the result of computing a flow function depends on more than one input value. In constant-propagation this occurs at statements such as `a = b+c`, where the analysis can precisely compute the result `a` only when knowing `b` and `c`. Very unfortunately, points-to analysis is also non-distributive: when encountering an assignment such as `a.f = o` then the analysis must associate `o` not just with `a.f` but all its aliases. For this reason, for a long time it has been believed that it is impossible to

```
void main(){
  x = new O();
  y = new O();
  z = new O();
  foo(x,y,z);   //c1
  u = new O();
  v = u;
  foo(u,v,z);   //c2
}

<T> T foo(T a, T b, T c){
   a.f = c;
   return b.f;
}
```

Listing 4. Example illustrating unsoundness of pointwise summaries

precisely encode pointer analysis in a distributive framework. Again in our recent work on Boomerang we showed that this is not entirely so:

> To a large degree, one can precisely encode pointer analysis as a distributive static-analysis problem.

I invite the interested reader to refer to details in our ECOOP publication on Boomerang [22]. For here, suffice it to say that the trick is to encode in a distributive framework—in this case in IFDS—everything *but* the processing of those statements that require a non-distributive treatment, in this case heap assignments. For most of its computation, Boomerang uses IFDS with efficient, pointwise procedure summaries over access graphs. This allows it to not only compute results very efficiently but also with maximal precision. Boomerang is also demand-driven, and itself makes use of this fact: at heap assignments of the form `a.f = o`, Boomerang calls itself to recursively compute all aliases of `a`.[1] Once the result of this query has been computed, it is incorporated back into the IFDS solver, which continues its own efficient distributive analysis. Also note:

> When designing demand-driven summary-based analyses, it is beneficial to define summaries in such a way that they can be also reused across different (but similar queries).

Boomerang follows this idea: it allows one to reuse summaries across partial pointer-assignment chains.

Unsoundness in using summaries

Boomerang’s distributive encoding of points-to relationships comes with a caveat, though, with respect to the soundness of some computed summaries. To illustrate, consider the example code in Listing 4. Here `main` calls `foo` at the two calling contexts `c1` and `c2`. At `c1`, the variables `x`, `y` and `z` point to separate objects, at `c2` `u` and `v`, and therefore formal parameters `a` and `b` in `foo` alias. Now if a summary-based analysis were to use for this example the distributive encoding presented above, this could have the following effect. Suppose the analysis first processes call site `c1`. Since at `c1` we have no aliasing, one would compute for `c` in `foo` this simple summary:

$$\texttt{c} \mapsto \{\texttt{c}, \texttt{a.f}\}$$

In particular, this summary indicates—correctly—that the return value does not depend on `c`. Now suppose the analysis processes call site `c2`. Since it has processed `foo` with the information “c” before, it finds that it should simply reuse the summary we had computed already at `c1`. This is unsound, however: a sound summary for `c` at `c2` would look like this:

$$\texttt{c} \mapsto \{\texttt{c}, \texttt{a.f}, \texttt{b.f}, \texttt{<ret>}\}$$

In particular, it would have to indicate a tainted return value contains a copy of the pointer `c`.

Such unsoundness might be unacceptable to some client analyses, which is why in our implementations we have included the options to disable summaries. Our experiments show that even without summaries the concise domain that the distributive encoding provides lowers the analysis time significantly in comparison to analyses that use more complex encodings—while providing superior precision.

Also, my group has a strong focus on developing security code analyses to find security vulnerabilities, and in this application context, we have learned that developers greatly prefer approaches that indicate actual vulnerabilities rapidly and with high precision over approaches that are sound but suffer from false positives and lack efficiency. In such scenarios, the above distributive encoding hence makes a lot of sense even with summaries, despite a potential for the unsoundness this may entail.

The unsoundness could be avoided by using more complex summaries that forego being reused in contexts in which they would be unsound to use, but at the obvious expense of losing some reuse potential, and with this efficiency. Another way to limit the unsoundness is to share summaries only when they relate to the same abstract objects. This is a technique that we implemented in Boomerang.

7 Demand-driven analysis as a design paradigm

The general idea of demand-driven analyses carries, in my view, a huge potential: in recent work on a static-analysis tool called CogniCrypt [8, 9] (http://www.cognicrypt.org/), we

[1] Heap assignments are just one of several such “points of indirection”. Our original paper on Boomerang [22] lists all the ones that are relevant for Java. Languages with more flexible pointer accesses, such as C, require additional points of indirection.

were able to show how one can chain a number of different demand-driven, distributive computations such that jointly they solve a complex real-world static-analysis problem. CogniCrypt is a tool to identify misuses of cryptography APIs, and it does so through a combination of a demand-driven value analysis, typestate analysis and pointer analysis. CogniCrypt combines those analyses in the following way: First the tool syntactically scans the program's code for method calls of interest. Typically the tool must then analyze the order in which those calls are issues on certain objects, and the parameter value passed to these calls.

CogniCrypt thus uses a demand-driven typestate analysis to identify call sequences, and a demand-driven value analysis to determine possible parameter values. Both are implemented in a distributive way, and both, when encountering a heap access, use Boomerang to efficiently and precisely compute pointer information.

While CogniCrypt resembles one concrete instantiation of this methodology, with IDE^{al} we have presented a general framework to implement such analyses in this fashion. [21] IDE^{al} in particular allows anyone to easily implement field-sensitive static analyses in IDE, without having to worry about aliasing: IDE^{al} automatically takes care of propagating information into aliases and of applying strong updates [10] where possible.

> We have recently succeeded in creating even complex client analyses in a very precise and efficient way by implementing them as a combination of multiple distributive, demand-driven analyses that call each other recursively. This has many advantages over previous approaches that attempt to solve multiple problems within one single analysis using a complex abstract domain with non-distributive flow functions: every single demand-driven analysis has a small abstract domain restricted to "its" analysis problem, often facilitating a distributive encoding and concise summaries.

Implementing analyses that way has many nice properties: First, the abstract domain of each individual analysis (typestate, value, pointer) is concise, and hence can yield highly reusable summaries. Second, each individual implementation is distributive, meaning that those summaries are point-wise, increasing their reuse potential. Third, also because of distributivity, the results obtained are highly precise. As explained at the beginning of this paper, this precision, in turn adds even more efficiency. Combined with the demand-driven nature of the analyses, one arrives at a very efficient solution. Again the key to success here is the use of storeless, i.e., access-path-based abstractions.

Of the three distributive frameworks IFDS, IDE and WPDS, weighted pushdown systems (WPDS) is the most general one. IFDS is essentially restricted to computing summaries in the form of simple mappings. This is sufficient for taint analysis such as the one in Figure 1 because here the domain is very simple and restricted to static program constructs, namely variables and field accesses. More complex analyses such as constant propagation reason about runtime values. Those analyses are better expressed in the IFDS extension IDE. If one were to express constant propagation in IFDS, this would be possible only with extensional summaries, with all the aforementioned problems those incur. In IDE, one can express distributive summary functions such as the increment function I mentioned earlier: $\lambda x.\ x \mapsto x + 1$. The WPDS framework has an identical expressiveness to IDE. Yet, WPDS allows for a more compact representation of the static-analysis data structures and also allows for a relatively simple synchronization of multiple analyses. With WPDS, essentially the entire static analysis can be computed within stack automata. In recent, unpublished work we have migrated `Boomerang` and IDE^{al} from IFDS, respectively IDE to WPDS and have in many cases observed significant speedups, which were obtained by representing both context-matching and field-write/read-matching as WPDS problems.

8 A note on sparse analyses

A large number of recent works has recently presented so-called sparse analyses [1, 11, 12, 25]. The idea behind sparseness is to conduct data-flow analyses not along the regular program's control-flow graph (CFG) but rather along a more compact definition-use graph, also called value-flow graph (VFG). In my eyes, this compactness is also the only major advantage of the technique. Hence, while sparseness certainly allows for a more efficient analysis in general, one must also account for the fact that the creation of the VFG from the CFG does not come for free. So far, it seems that all approaches that have used sparseness, and have benefitted from it, were based on the call-strings approach and/or used store-based heap abstractions, thereby hindering summarization effects as I explained above. Since the approach described in this work follows a fundamentally different philosophy, so far we did not see the need to make our analyses sparse. Also, due to the demand-driven nature of most of the analyses that we build, it is unclear whether the time to build the VFG would not outweigh the time savings the sparse analysis might yield—an interesting open research question.

9 Conclusion

In this work I have shown that—using the right design tricks—one can arrive at a very effective design, providing both high efficiency and precision, in particular for such static analyses that are not exhaustive but instead can safely focus on some relevant parts of a given program. The first essential idea is to execute the analysis in a demand-driven way, the second to use small, concise abstractions within in a distributive framework such as IFDS, IDE or WPDS. As I have shown, one can successfully encode within those frameworks also non-distributive problems such as pointer analyses to some large extent. Concise abstractions yield small domains, which in turn allows one to store highly reusable procedure summaries. Such summaries can speed up the computation even more, but sometimes at the expense of soundness.

Demand-driven analyses become particular powerful when used in combinations: the possibility an analysis can at any point compute additional information on demand, either by calling other demand-driven analyses or by calling itself recursively, opens up many possibilities that one does not easily obtain with traditional algorithms.

Acknowledgements I wrote this paper to present my personal experience of the research in which I was involved over the past few years. Much of the research described here was first and foremost conduct by a bright set of Ph.D. students and PostDocs I had the pleasure of working with, in particular: Johannes Späth, Lisa Nguyen Quang Do, Stefan Krĝer, Johannes Lerch, Karim Ali and Ben Hermann. Thanks a lot to you and also to my long-term collaborator Mira Mezini! Johannes and Ben I also wish to thank in particular for many useful comments that greatly helped me improve my initial drafts of this paper. The paper also received a thorough review and many comments by Uday Khedker. Thanks Uday! The research that lead to the conclusions I describe here, has been funded by the Fraunhofer ATTRACT program, by the DFG through the Emmy Noether Group RUNSECURE and the Collaborative Research Centers CROSSING and On-the-fly Computing, by the Oracle ERO grants 940 and 1579, and by the Heinz Nixdorf Foundation.

References

[1] Ben Hardekopf and Calvin Lin. 2011. Flow-sensitive pointer analysis for millions of lines of code. In *Code Generation and Optimization (CGO), 2011 9th Annual IEEE/ACM International Symposium on*. IEEE, 289–298.

[2] Michael Hind. 2001. Pointer analysis: Haven't we solved this problem yet?. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering (PASTE)*. ACM, 54–61.

[3] SANS Institute. 2018. SANS 25 - Top 25 Most Dangerous Software Errors. https://www.sans.org/top25-software-errors. (2018).

[4] Swati Jaiswal, Uday P. Khedker, and Supratik Chakraborty. 2018. Demand-driven Alias Analysis : Formalizing Bidirectional Analyses for Soundness and Precision. (2018). arXiv:1802.00932

[5] John B Kam and Jeffrey D Ullman. 1977. Monotone data flow analysis frameworks. *Acta Informatica* 7, 3 (1977), 305–317.

[6] Vini Kanvar and Uday P Khedker. 2016. Heap abstractions for static analysis. *ACM Computing Surveys (CSUR)* 49, 2 (2016), 29.

[7] Uday P Khedker, Amitabha Sanyal, and Amey Karkare. 2007. Heap reference analysis using access graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 30, 1 (2007), 1.

[8] Stefan Krüger, Sarah Nadi, Michael Reif, Karim Ali, Mira Mezini, Eric Bodden, Florian Göpfert, Felix Günther, Christian Weinert, Daniel Demmler, and Ram Kamath. 2017. CogniCrypt: Supporting Developers in using Cryptography. In *International Conference on Automated Software Engineering (ASE 2017), Tool Demo Track*. ACM. http://bodden.de/pubs/knr+17cognicrypt.pdf

[9] Stefan Krüger, Johannes Späth, Karim Ali, Eric Bodden, and Mira Mezini. 2018. CrySL: An Extensible Approach to Validating the Correct Usage of Cryptographic APIs. In *European Conference on Object-Oriented Programming (ECOOP)*. To appear.

[10] Ondrej Lhoták and Kwok-Chiang Andrew Chung. 2011. Points-to Analysis with Efficient Strong Updates. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*. ACM, New York, NY, USA, 3–16. https://doi.org/10.1145/1926385.1926389

[11] Lian Li, Cristina Cifuentes, and Nathan Keynes. 2011. Boosting the performance of flow-sensitive points-to analysis using value flow. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 343–353.

[12] Magnus Madsen and Anders Møller. 2014. Sparse dataflow analysis with pointers and reachability. In *International Static Analysis Symposium*. Springer, 201–218.

[13] Rohan Padhye and Uday P Khedker. 2013. Interprocedural data flow analysis in soot using value contexts. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on State Of the Art in Java Program analysis*. ACM, 31–36.

[14] Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 49–61.

[15] Thomas Reps, Stefan Schwoon, Somesh Jha, and David Melski. 2005. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Science of Computer Programming* 58, 1-2 (2005), 206–263.

[16] Henry Gordon Rice. 1953. Classes of recursively enumerable sets and their decision problems. *Trans. Amer. Math. Soc.* 74, 2 (1953), 358–366.

[17] Mooly Sagiv, Thomas Reps, and Susan Horwitz. 1996. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science* 167, 1-2 (1996), 131–170.

[18] SAP 2018. Personal communication at SAP Security Research Seminar. (2018).

[19] Micha Sharir and Amir Pnueli. 1978. Two approaches to interprocedural data flow analysis. (1978).

[20] Asia Slowinska and Herbert Bos. 2009. Pointless tainting?: evaluating the practicality of pointer tainting. In *Proceedings of the 4th ACM European conference on Computer systems*. ACM, 61–74.

[21] Johannes Späth, Karim Ali, and Eric Bodden. 2017. IDEal: Efficient and Precise Alias-aware Dataflow Analysis. In *2017 International Conference on Object-Oriented Programming, Languages and Applications (OOPSLA/SPLASH)*. ACM Press. http://bodden.de/pubs/sab17ideal.pdf

[22] Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. 2016. Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java. In *European Conference on Object-Oriented Programming (ECOOP)*. http://www.bodden.de/pubs/sna+16boomerang.pdf

[23] Manu Sridharan and Rastislav Bodík. 2006. Refinement-based Context-sensitive Points-to Analysis for Java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06)*. ACM, New York, NY, USA, 387–400. https://doi.org/10.1145/1133981.1134027

[24] Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. 2005. Demand-driven Points-to Analysis for Java. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '05)*. ACM, New York, NY, USA, 59–76. https://doi.org/10.1145/1094811.1094817

[25] Yulei Sui and Jingling Xue. 2016. On-demand strong update analysis via value-flow refinement. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 460–473.