The Secret Sauce in Efficient and Precise Static Analysis
The Beauty of Distributive, Summary-Based Static Analyses (and how to master them)

Eric Bodden
No. 1 goal of this talk

Refute these myths:

As static analysis gets more precise, it becomes less efficient.

Context-sensitive and flow-sensitive analysis is impossible to scale, thus one needs to approximate.
Common mistake in static analysis

Client analysis
Highly precise: context-sensitive, field-sensitive, flow-sensitive, …

Pointer analysis
Very imprecise: Andersen-style, flow-insensitive, context-insensitive
Common mistake in static analysis

Client analysis
Highly precise: context-sensitive, field-sensitive, flow-sensitive, …

Imprecision

Pointer analysis
Very imprecise: Andersen-style, flow-insensitive, context-insensitive
Example: false positive due to flow-insensitive pointer analysis

```java
A a = new A();
A b = new A();

a.x = secret();
leak(b.x);

a = b;
```

*alias created afterwards, yet causes false positive*
Lessons

1. Precise pointer analyses are key!
But precise analyses are hard to scale, no?
They are, as whole-program analysis, but often we can get away with a Demand-driven Analysis.
Can an attacker manipulate the string?

Integrity problem

Can the attacker get hold of the returned class?

Confidentiality problem

Taint Analysis!

```java
Class<?> findClass(String paramString) {
    return Class.forName(paramString);
}
```
Class<?> findClass(String paramString) {
    return Class.forName(paramString);
}
Bottom-up analysis

Class<?> findClass(String paramString) {
    return Class.forName(paramString);
}
Class<?> findClass(String paramString) {
    return Class.forName(paramString);
}
```java
Class<?> findClass(String paramString) {
    return Class.forName(paramString);
}
```

aka. “unbalanced return”

no calling context!

[ASE’15]
Class<?> findClass(String paramString) {
    return Class.forName(paramString);
}
Class<?> findClass(String paramString) {
    return Class.forName(paramString);
}

Balanced return only to correct call site

[ASE’15]
Properties of a demand-driven analysis

• Computed responses to on-demand queries, e.g.: \texttt{pointsTo(a.f.g)}

• Analysis starts anywhere, “in the middle of the program”

• It typically conducts some combination of backward and forward analysis

• It necessarily has to deal with unbalanced returns
Effects of a demand-driven analysis

• The analysis becomes localised
• Precision therefore may increase
• But: it may report problems in dead code
• Can do recursive queries
On-demand analysis

- Compute **access paths** only when needed
- Compute **pointer information** only when needed
- Compute **string information** only when needed
- Compute **call edges** only when needed

[ASE 2015]
[ECOOP 2016]
[ECOOP 2018]
ongoing…

can query one another!
Lessons

1. Precise pointer analyses are key!

2. Demand-driven analysis can aid both performance and precision
Gaining further efficiency through procedure summaries
Taint analysis tracks both keys

```c
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;
}
```
which follows a "zero vulnerability" policy. 

\[\text{main()}\] 

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

```c
byte[] foo(byte[] ba) {
    //some hard-to-analyze code omitted
    return ba;
}
```
void main() {
    byte[] pri = privateKey();
    byte[] pub = publicKey();
    byte[] priAlias = foo(pri);  //context c1
    byte[] pubAlias = foo(pub);  //context c2
    ...
    log();
}

byte[] foo(byte[] ba) {  
    //some hard-to-analyze code omitted
    return ba;
}
Which follows a “zero vulnerability” policy. 

Ten to log one of the two following means:
- Making inter-procedural static analyses context-sensitive by summarization.
- Avoiding the repeated re-analysis of identical code, in particular, procedure summaries are important for a very simple reason: they avoid the repeated re-analysis of identical code, in particular, procedure summaries are important for a very simple reason: they avoid the repeated re-analysis of identical code.

4 Summary-based analysis frameworks

Others accept unsoundness by returning incomplete information. Some approaches opt for soundness in those cases, returning more imprecise results of a simpler to compute analysis. [1] Yet, in some cases, the client might require the most precise information. To illustrate the benefit of summarization, consider the following code:

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

In the functional approach, one obtains context-sensitivity by annotating data-structures that a call to a procedure has already been encountered, a reusable summary resembling the first call site will have in terms of the static context.

In the call-strings approach, one would analyze the procedure twice, without gaining anything, but summarize its effects that a call to a procedure has already been encountered, a reusable summary resembling the first call site will have in terms of the static context.

Demand-driven analysis has the benefit of being efficient and precise. Static analysis is conducted upon the first time a call to a procedure is encountered, a reusable summary resembling the first call site will have in terms of the static context.

In those cases, faced the challenge of summarization, consider the following code snippet:

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

While this would yield the correct, precise result "<ret>", it wastes computation: irrespective of whether and how to nonetheless provide the client with some answer to the query. Some so-called re-analysis approaches built on top of the call-strings approach typically report the need to create millions if not billions of contexts—still compute a long time. In those cases it would be worthwhile to employ the functional approach instead only once!

1. Call-Strings summary-based analysis
2. Functional summary-based analysis
3. Demand-driven analysis

There are many summary-based approaches, including IFDS [5], IDE [13], WPDS [12], which follows a "zero vulnerability" policy. 

For decades, ever since the foundation of cryptography, companies are with me on this issue, for instance SAP SE, which follows a "zero vulnerability" policy. 

Large software development is common to abort queries after a given time frame or budget despite all efforts—still compute a long time. In those cases it would be worthwhile to employ the functional approach instead only once!

In those cases, faced the challenge of summarization, consider the following code snippet:

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

While this would yield the correct, precise result "<ret>", it wastes computation: irrespective of whether and how to nonetheless provide the client with some answer to the query. Some so-called re-analysis approaches built on top of the call-strings approach typically report the need to create millions if not billions of contexts—still compute a long time. In those cases it would be worthwhile to employ the functional approach instead only once!
Extensional vs. intensional summaries

extensional definition of the \textit{increment} function:

\[ 0 \mapsto 1, 1 \mapsto 2, 2 \mapsto 3, \ldots \]

intensional definition:

\[ \lambda x. x \mapsto x + 1 \]

\textit{much more compact!}
Precise pointer analyses are key!

Demand-driven analysis can aid both performance and precision

Summaries avoid repeated re-analysis of callee procedures

One should strive for intensional summary definitions
How to compute summaries?
assume forward analysis...

Procedure $r$

- $\Phi_r(u_1) \equiv \phi_{id}$
- $\Phi_r(u_2) \equiv f_1$
- $\Phi_r(u_3) \equiv f_1$
- $\Phi_r(u_4) \equiv f_1$
- $\Phi_r(u_5) \equiv f_2 \circ f_1$
- $\Phi_r(u_6) \equiv f_3 \circ f_1$
- $\Phi_r(u_7) \equiv f_2 \circ f_1 \sqcap f_3 \circ f_1$
- $\Phi_r(u_8) \equiv f_4 \circ (f_2 \circ f_1 \sqcap f_3 \circ f_1)$

May 2011

Uday Khedker

Slide uses illustration by Uday Khedker
Generally assume that flow functions are **monotone**:

\[ f(x) \cap f(y) \subseteq f(x \cap y) \]

To be able to compute intensional summaries, they **need to be distributive**:

\[ f(x) \cap f(y) = f(x \cap y) \]
IFDS, IDE and WPDS compute summaries automatically, but care must be taken when choosing the appropriate heap abstraction!
Assume that this time we wish to conduct a

As can easily be seen, extensional summary de

variables such as

objects through allocation sites. The example contains two

and context-sensitive points-to analysis. Typical points-to

following, this causes problems with summary reuse. 

Note the following di

An intensional de

represents all possible inputs to the function. This is because

Note that procedure summaries such as the one above

An extensional summary de

can drastically reduce the complexity of the static analysis. 

By avoiding the repeated re-analysis of callee procedures for

An intensional de

different calling contexts, the use of procedure summaries

See different call to

An intensional de

rienctions: Firstly this de

An intensional de

dition is much
ter, as they tend to propagate non-local infor-

This is also what would happen in the example in Listing 2:

This is also what would happen in the example in Listing 2:

The problem with the above summary is that—despite

The summary "

instance matters in this particular calling context

One reason for this is that in such cases

When designing procedure summaries, one should strive for

When designing procedure summaries, one should strive for

extensively extended to static-analysis problems blindly,

increment-function could look like this:

increment-function could look like this:

the intensional de

The summary "

the function's e

so that

As an example, let us consider the example in Listing 2.

As an example, let us consider the example in Listing 2.

In the past, many fellow researchers and I myself have un-

The summary "

is a formal parameter

The summary "

is a formal parameter

is a local meaning to the procedure being summarized. Typ-

is a formal parameter

is a local meaning to the procedure being summarized. Typ-

The summary "

is a formal parameter

is a local meaning to the procedure being summarized. Typ-

Note how this summary contains not only information that

Note how this summary contains not only information that

outputs:

Note how this summary contains not only information that

outputs:

outputs:

outputs:

Note how this summary contains not only information that

outputs:

Note how this summary contains not only information that

outputs:

Note how this summary contains not only information that

outputs:

outputs:

Note how this summary contains not only information that

outputs:

outputs:

outputs:

Note how this summary contains not only information that

outputs:

outputs:

Note how this summary contains not only information that

outputs:

outputs:

outputs:

Note how this summary contains not only information that

outputs:

outputs:

outputs:

Note how this summary contains not only information that

outputs:

outputs:

Note how this summary contains not only information that

outputs:

outputs:

Note how this summary contains not only information that

outputs:

outputs:

Note how this summary contains not only information that

outputs:

outputs:

Note how this summary contains not only information that

outputs:

outputs:

Note how this summary contains not only information that

outputs:

outputs:

Note how this summary contains not only information that

outputs:

outputs:

Note how this summary contains not only information that

outputs:

outputs:

This is because

increase-function could look like this:

increase-function could look like this:

the intensional de

represents all possible inputs to the function. This is because

Note that procedure summaries such as the one above

This is also what would happen in the example in Listing 2:

This is also what would happen in the example in Listing 2:

The problem with the above summary is that—despite

The problem with the above summary is that—despite

The problem with the above summary is that—despite

Note how this summary contains not only information that

Note how this summary contains not only information that

Note how this summary contains not only information that

Note how this summary contains not only information that

Note how this summary contains not only information that

Note how this summary contains not only information that

Note how this summary contains not only information that

outputs:

Note how this summary contains not only information that

outputs:

outputs:

Note how this summary contains not only information that

outputs:

outputs:

Note how this summary contains not only information that

outputs:

outputs:

Note how this summary contains not only information that

outputs:

outputs:

Note how this summary contains not only information that

outputs:

outputs:

Note how this summary contains not only information that

outputs:

outputs:

Note how this summary contains not only information that

outputs:

outputs:

Note how this summary contains not only information that

outputs:

outputs:

Note how this summary contains not only information that

outputs:

outputs:

Note how this summary contains not only information that

outputs:

outputs:

Note how this summary contains not only information that

outputs:

outputs:

Note how this summary contains not only information that

outputs:

outputs:

Note how this summary contains not only information that

outputs:

outputs:
Assume that this time we wish to conduct a
extensional summary of procedure summaries. As can easily be seen, extensional summaries
are not a good choice. They are very verbose, making it hard to store them compactly. Second, to be
represented at all, they typically must be restricted to a
finite set of information, which should then actually result in a di
terent calling context, the use of procedure summaries
would normally give. One reason for this is that in such cases
summaries are frequently extensional.

This is also what would happen in the example in Listing 2:

```c
<T> T foo(T o) {
    //some hard-to-analyze code omitted
    return o;
}
```

One problem: summary for `alloc1` is not reusable for `alloc2`

```c
void main() {
    A a1 = new A(); //alloc1
    A a2 = new A(); //alloc2
}
```

Note how this summary contains not only information that
was e
consciously neglected this guideline, resulting in summaries
that were e
ct on
variables such as
objects through allocation sites. The example contains two
and context-sensitive points-to analysis. Typical points-to
functions have representations that are either
intensional (opposed to extensional) summary de
nitions abstracts from concrete parame-
ters through its input variable
ers through its input variable
functions

```c
<T> T foo(T o) {
    //some hard-to-analyze code omitted
    return o;
}
```

This is also what would happen in the example in Listing 2:

```c
void main() {
    A a1 = new A(); //alloc1
    A a2 = new A(); //alloc2
    //context c1
    //context c2
}
```

```c
<T> T foo(T o) {
    //some hard-to-analyze code omitted
    return o;
}
```

This is also what would happen in the example in Listing 2:
Instead use access paths

\[ l \ldots k \text{ field accesses} \]

\[ l.f.g.h \]

local variable
### First advantage: much smaller abstract domain!

```c
void main() {
    A a1 = new A(); // alloc1
}
<T> T foo(T o) {
    // some hard-to-analyze code omitted
    return o;
}
<ret>
```
Assume that this time we wish to conduct a

As can easily be seen, extensional summary de

variables such as

objects through allocation sites. The example contains two

analyses use so-called store-based abstractions 

following, this causes problems with summary reuse.

that were e

consciously neglected this guideline, resulting in summaries

represented by

is compact because it is e

 ters through its input variable

resents all possible inputs to the function. This is because

more compact than the extensional one. Secondly, it rep-

Note the following di

as they are compact and represent the entire possible input

functions

Such functions can

can drastically reduce the complexity of the static analysis.

An intensional de

An extensional summary de

nes a function by asso-

rithm such as IFDS, IDE, WPDS and VASCO, this can easily

Listing 2.

2nd advantage: analyzing foo only once!

void main() {
    A a1 = new A(); //alloc1
    A a2;
    A a1;
    A a2;
}

<T> T foo(T o) {
    //some hard-to-analyze code omitted
    return o;
}
Lessons

1. **Precise pointer analyses** are key!
2. **Demand-driven analysis** can aid both performance and precision
3. **Summaries** avoid repeated re-analysis of callee procedures
4. One should strive for **intensional summary definitions**
5. The point of procedure summaries is not to be able to create them but to be able to **reuse** them!
6. Alloc-site-based heap abstraction **hinder summary reuse**
7. Access paths **foster reusable summaries**
Remember:

To be able to compute intensional summaries, they need to be distributive:

\[ f(x) \cap f(y) = f(x \cap y) \]

But: distributive frameworks yield perfect precision!
Then it holds:

### Important: The MFP soundly solution approximates the MOP solution:

1. **Meet over all paths**
   \[ \forall s. \ MOP(s) \subseteq MFP(s) \]
   \[ f(x) \cap f(y) \subseteq f(x \cap y) \]

2. **Maximal fixed point**
   \[ \forall s. \ MOP(s) = MFP(s) \]
   \[ f(x) \cap f(y) = f(x \cap y) \]
\[ \forall s. \ MOP(s) \subseteq MFP(s) \]

\[ f(x) \cap f(y) \subseteq f(x \cap y) \]

merging earlier gives an over-approximation

\[ f(x) \cap f(y) = f(x \cap y) \]
\[ \forall s. \ MOP(s) \subseteq MFP(s) \]

it does not matter when we merge!

\[ \forall s. \ MOP(s) = MFP(s) \]

\[ f(x) \cap f(y) = f(x \cap y) \]
which in this case is not applicable. VASCO would hence have to re-analyze the callee, producing a second summary: summaries, as follows:

In the same example, at the first call to `makePair(secretKey(),secretKey())`, the abstract inputs were tainted before the callee was entered, so the result is point-wise. In this case, one can use the value of the points-to analysis to compute a reusable summary:

```
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;
}
```
two taints

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

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

one taint
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<>();
    ret.left = a;
    ret.right = b;
    return ret;
}

Example illustrating the advantage of
functions, such as VASCO, that does not assume distributivity, would be able to produce independently reusable pointwise summaries that have this property, one gains the big advantage of being able to produce summaries, as follows:

In the same example, at the first call to
makePair(secretKey(),secretKey()),
the abstract state, i.e., an element within the input set, changes from one call to the next, the computed summary cannot be reused as a whole, it can only be reused as a point-wise summary. In the same example, at the first call to
makePair(secretKey(),publicKey()),
the newly encountered abstract state. So the callee, producing a second summary:

Unfortunately, though, not all problems are distributive. Problems generally become non-distributive when the result of a call is tainted, but not the second parameter at first context

IFDS, IDE and WPDS are frameworks that assume distributivity: this time only the first parameter
were tainted before the call. Now at the second call to
makePair(secretKey(),secretKey()),
the analysis can nonetheless reuse the summary: it states that as soon as the result
was tainted, so becomes point-wise summaries. In constant-propagation this occurs at statements —independently of everything else.

This is very important to note: computing a fixed-point solution, also called MFP. The MFP solution is hereby guaranteed to be a sound over-approximation:

The optimal solution to any static analysis problem is the so-called meet-over-all-paths solution, also called MOP. In practice, this solution can be computed. \[ \text{MOP} \]

\[ \text{MFP} \]

But distributive frameworks have another big advantage: for a long time it has been believed that it is impossible to associate elements of the abstract domain.

This relationship is guaranteed by the requirement that distributive frameworks must be able to produce point-wise summaries that are highly reusable, such as VASCO, that does not assume distributivity, would be able to produce independently reusable pointwise summaries that have this property, one gains the big advantage of being able to produce summaries, as follows:

In the same example, at the first call to
makePair(secretKey(),secretKey()),
the abstract state, i.e., an element within the input set, changes from one call to the next, the computed summary cannot be reused as a whole, it can only be reused as a point-wise summary. In the same example, at the first call to
makePair(secretKey(),publicKey()),
the newly encountered abstract state. So the callee, producing a second summary:

Unfortunately, though, not all problems are distributive. Problems generally become non-distributive when the result of a call is tainted, but not the second parameter at first context

IFDS, IDE and WPDS are frameworks that assume distributivity: this time only the first parameter
were tainted before the call. Now at the second call to
makePair(secretKey(),secretKey()),
the analysis can nonetheless reuse the summary: it states that as soon as the result
was tainted, so becomes point-wise summaries. In constant-propagation this occurs at statements —independently of everything else.

This is very important to note: computing a fixed-point solution, also called MFP. The MFP solution is hereby guaranteed to be a sound over-approximation:

The optimal solution to any static analysis problem is the so-called meet-over-all-paths solution, also called MOP. In practice, this solution can be computed. \[ \text{MOP} \]

\[ \text{MFP} \]

But distributive frameworks have another big advantage: for a long time it has been believed that it is impossible to associate elements of the abstract domain.
which in this case is not applicable. VASCO would hence generate summaries, as follows:

To produce independently reusable pointwise summaries, one can very efficiently encode the analysis problem the following stronger condition holds:

Unfortunately, though, not all problems are distributive. Problems generally become non-distributive when the result of an assignment such as

As already explained above, for any distributive analysis

This is very important to note: for a long time it has been believed that it is impossible to create procedure summaries in cases where the analysis is non-distributive. Points-to analysis is also non-distributive: when encountering the result of a computation such as

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.

At second context

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

can reuse this partial summary!
sound only due to distributivity!

```java
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;
}
```

can reuse this partial summary!
Lessons

1. **Precise pointer analyses** are key!
2. **Demand-driven analysis** can aid both performance and precision
3. **Summaries** avoid repeated re-analysis of callee procedures
4. One should strive for **intensional summary definitions**
5. The point of procedure summaries is not to be able to create them but to be able to **reuse** them!
6. Alloc-site-based heap abstraction **hinder summary reuse**
7. Access paths **foster reusable summaries**
8. Intensional summaries **require distributivity**
9. Distributivity yields **perfect precision**
10. Distributive summaries can be **more effectively reused**
My 10 laws of precise and efficient analysis

1. **Precise pointer analyses** are key!
2. **Demand-driven analysis** can aid both performance and precision
3. **Summaries** avoid repeated re-analysis of callee procedures
4. One should strive for **intensional summary definitions**
5. The point of procedure summaries is not to be able to create them but to be able to **reuse** them!
6. Alloc-site-based heap abstraction **hinder summary reuse**
7. Access paths **foster reusable summaries**
8. Intensional summaries **require distributivity**
9. Distributivity yields **perfect precision**
10. Distributive summaries can be more **effectively reused**

Valid until disproven! ;-)
We are hiring!

PostDocs
and Ph.D. students