Using unreachable code analysis in static analysis tool for finding defects in source code

Reporter:

Rodion Mulyukov <eygz@ispras.ru>

ISP RAS

Svace

- **Svace** is a static analysis tool for finding defects in source code written in C, C++, Java and C#
- The analyzer tends to find as many defects as possible while keeping the rate of false positives low
- Types of defects:
  - Null pointer derefence
  - Division by zero
  - Buffer overflow
  - Use of uninitialized variables
  - Memory and resource leaks
  - Use of tainted data
  - Defects involving multithreading
  - Incorrect usage of library functions

Detecting unreachable code

- Compilers usually find and remove unreachable code for purposes of program optimization

Our goals

- We find unreachable code for:
  - reporting defects
  - improving accuracy of the main analysis:

    unreachable instructions shouldn't affect analysis of reachable instructions

Warnings about unreachable code

- The presence of unreachable code in a program may imply that
  - there's an error in the implementation of the intended algorithm
  - a programmer misunderstood the code he had changed
  - there might be outdated code the programmer forgot to remove

```
if ((ctxt == NULL) || (file == NULL))
    return -1;

if (file == NULL)
    return 0; // unreachable code
```

libxml2-2.7.8

```
for (i = N; i != 0;) {
    int decc = 0;
    int spdc = 0;

    firedec[--i] = decc;
    decc += spdc / 10;

    if (decc > 4)
        spdc -= 1; // unreachable code

    ...
}
```

gst-plugins-good-0.10-31

Types of unreachable code

- We can distinguish the following types of unreachable code:

No path

- no path from the Entry node in control flow graph

```
void foo() {
    ...
    return;
    a = 1; // unreachable code
}
```

```
void bar() {
    ...
    goto label:
    a = 1; // unreachable code
label:
    a = 2;
    ...
}
```

Types of unreachable code

After a termination call

- a function call terminates the procedure, making the next instructions unreachable

```
void foo() {
    ...
    fatal_error(1);
    a = 1; // unreachable code
}
```

```
void bar() {
    ...
    throw_exception();
    a = 1; // unreachable code
}
```

Types of unreachable code

After an invariant comparison

- an unreachable branch of code in conditional instructions

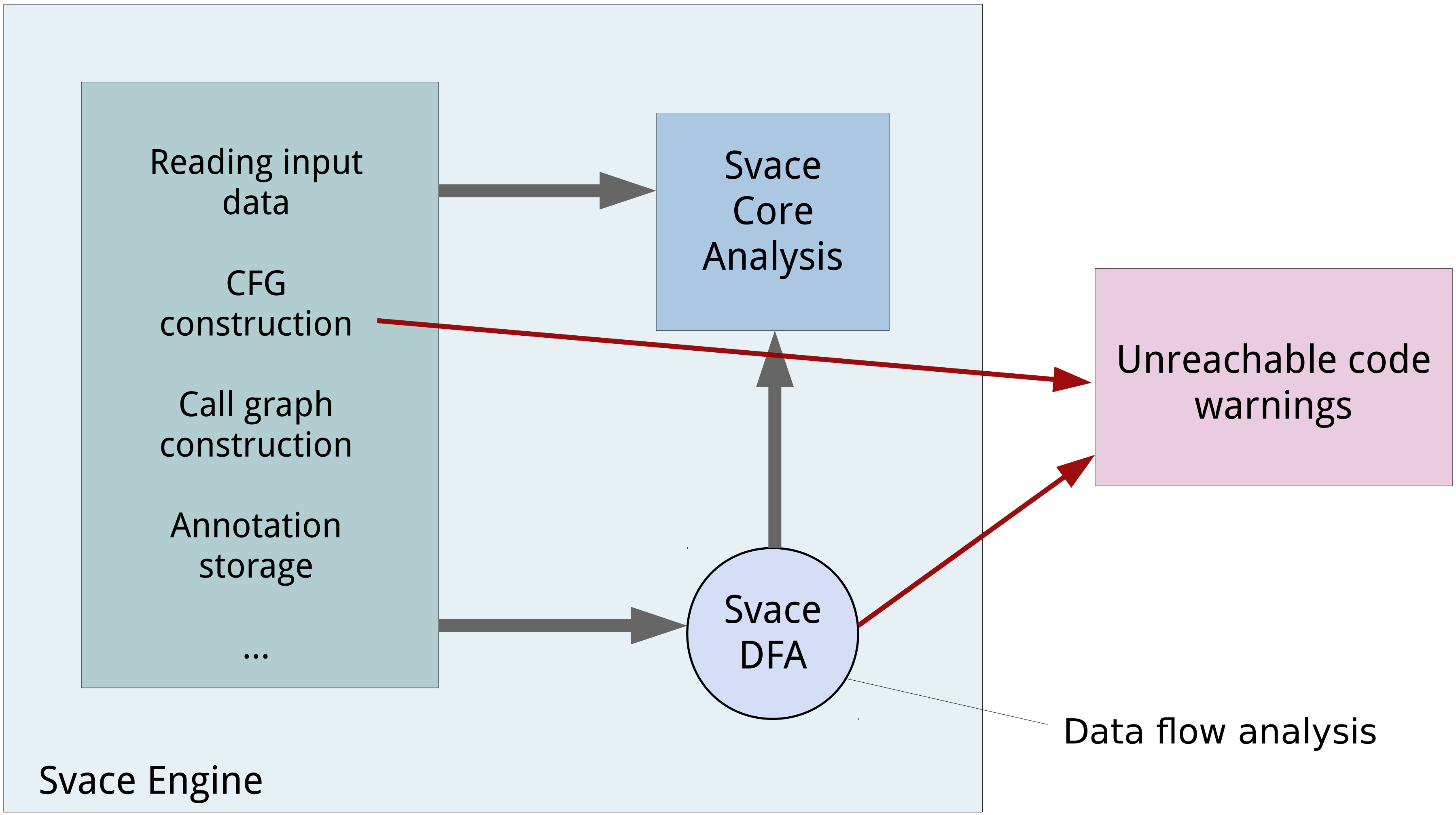
```
void foo() {
    int a = 0;
    int b = 1;
    if (a > b) {
        ... // unreachable code
    }
}
```

```
void bar(char *p) {

    if (!p)
        return;

    ... // p is unchanged

    if (p) {
        ...
    } else {
        ... // unreachable code
    }
}
```

Related Svace features

- Unreachable code detection on CFG construction
- Interprocedural termination call analysis
- Value interval analysis
- Analysis of missing values in value intervals
- Analysis of necessary conditions for program points
- Filtering out “Won't fix” warnings

Unreachable code analysis in the Svace Engine framework

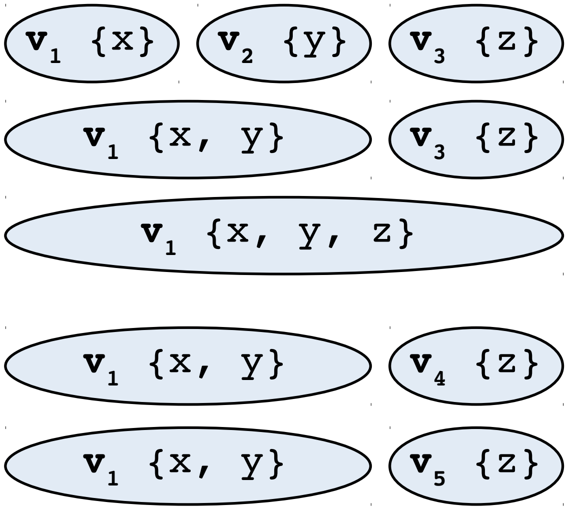


Value numbering

- **Value numbering** technique partitions the set of variables into equivalence classes at each program point
- *Value numbers* are the names of equivalence classes

```
void foo(int a, int b) {
    int x = a + b;
    int y = a + b;
    int z = x;
    if (z > 0) {
        z = 123;
    }
}
```

Program



Equivalence classes

Value numbering

Purposes of value numbering:

- Data flow analysis over the value numbers instead of the variables allows computing statements about the whole equivalence classes at once:

```
x = y;
if (x > 10) {
    if (y <= 10) {
        // unreachable code
    }
}
```

Value interval analysis

- Value interval analysis computes sound value intervals at each program point
- One can detect invariant comparisons using the computed intervals

```
if (…)
    x = 1;
else
    x = 3;

// x ∈ [1, 3]

if (x > 5) {
    ...       // unreachable code
}
```

Widening

- We use simple *widening* technique to make value interval analysis converge:
  - inside strongly connected components when propagating value intervals:

    1) if a bound increases, it's replaced with infinity

    2) new dataflow values are unchanged or weaker than the original ones

```
// n ∈ [-inf, +inf]
for (i = 0; i < n; ++i) {
    // i ∈ [0, 0]      1-st dataflow iteration
    // i ∈ [0, +inf]   2-nd dataflow iteration
}
```

Missing values analysis

- We implemented a simple version of the analysis:
  - At each program point the algorithm finds variables that can't be equal to 0

```
if (x != 0) {
    if  (x == 0) {
        // unreachable code
    }
}
```

Predicate analysis

- Sometimes even the most accurate dataflow values the previous analyses can express don't help:

```c
void foo(int a, int b) {

    if (a > b) {

        if (a <= b) {
            // unreachable code?
        }

    }
}
```

- We can model these situations with formulas

Predicate analysis

- Analysis is applied to a program translated into Static Single Assignment form
- Predicate analysis computes necessary conditions for program points
- The conditions have the form of conjunctions of atomic formulas

```
void foo(int a1, int b1) {

    if (a1 > b1) {
        // a1 > b1

        if (b1 != 3) {
            // a1 > b1 && b1 != 3
        }
        // a1 > b1

    } else {
        // a1 <= b1

    }
}
```

Predicate analysis

- Unreachable code is detected when the condition is FALSE:
  - the conjunction contains two opposite predicates A && !A
  - the conjunction contains a conflicting combination of predicates

```
void foo(int a1, int b1) {

    if (a1 > b1) {        // assume A
        ...
        if (a1 <= b1) {  // assume !A
            // unreachable code
        }
        ...
    }

}
```

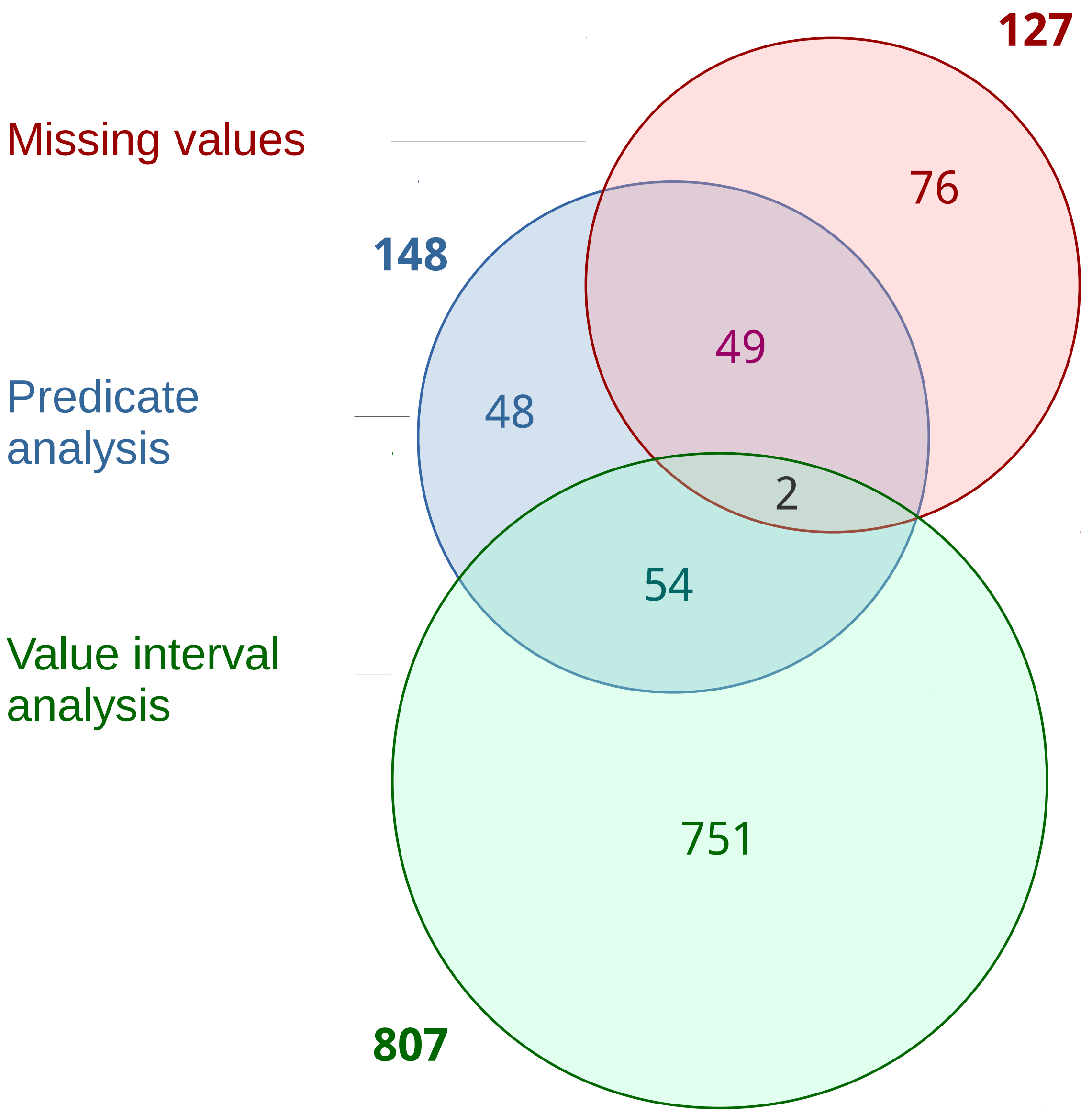
Current results

- Data flow analysis runtime is 3% of the whole Svace runtime

Number of warnings

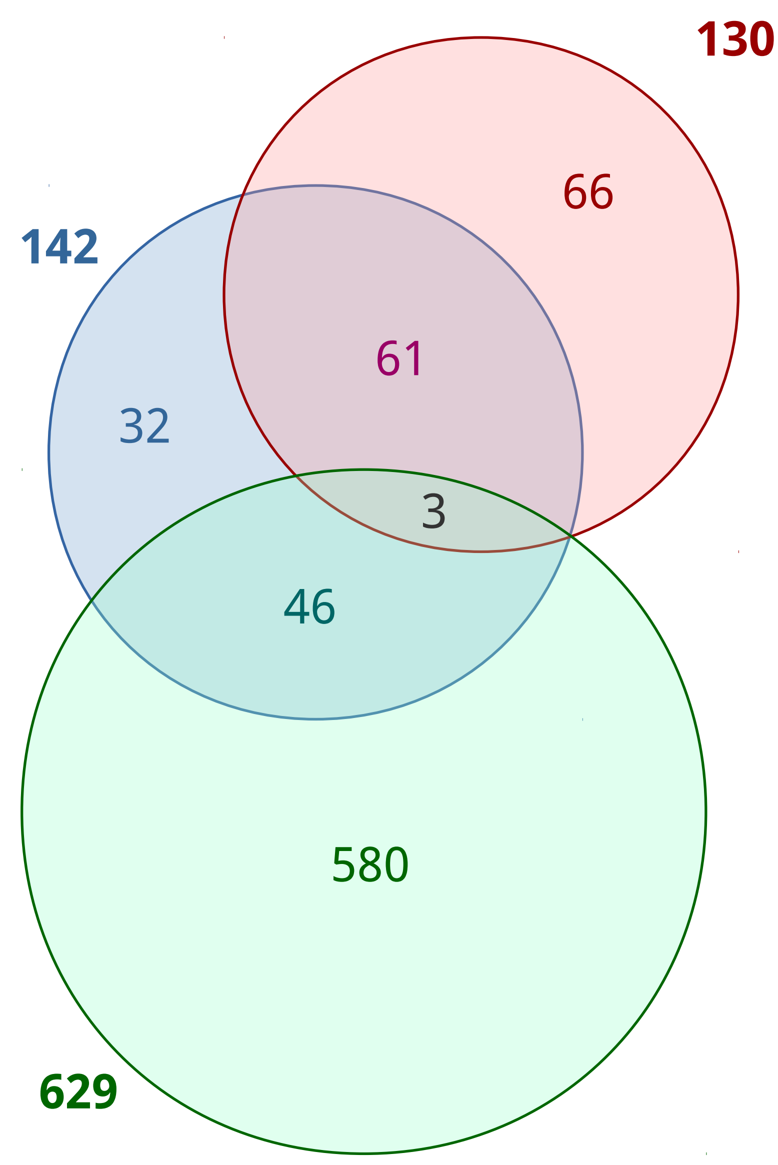
Project	Invariant comparisons	No path	Termination calls
android-5.0.2	980	165	187
tizen-2.3	788	173	60
binutils-2.22	53	14	22
cairo-1.12.14	8	5	0
glib-2.0	26	11	2
gnupg-1.4.11	13	1	5
openssl-1.01	63	2	6
libxml2-2.7.8	34	1	0

Invariant comparisons. Intersection of warnings



Project: Android-5.0.2

Number of warnings: **980**



Project: Tizen-2.3

Number of warnings: **788**

Typical “Won't fix” cases

- Many reports about unreachable code have no practical use:
  - Unreachable code within an expanded macro
  - Invariant comparisons originating from template parameters
  - Invariant comparisons with defined constants
  - Unreachable default case for a switch instruction
  - Invariant comparison “just in case”, for example:

```
free(p);
p = NULL;
goto failure;

... // many lines of code

return;
failure:
    if (p != NULL) {
        // unreachable code “Won't fix”
        free(p);
        p = NULL;
    }
```

ISP RAS

Warnings rating

From a user point of view:

- True positive: 52%
- “Won't fix”: 42%
- False positive: 6%

False positives result from:

  - Unclear warnings messages
  - Reports about compiler generated code

Future work

- Filter out most “Won't fix” cases
- Implement exception handler analysis to find unreachable catch blocks

Thanks for the attention!