Context Generation from Formal Specifications for C Analysis Tools

Michele Alberti[1] Julien Signoles[2]

[1]TrustInSoft

[2]CEA LIST, Software Reliability and Security Laboratory

LOPSTR 2017, Namur, Belgium

Code Analysis Tools

- Effective enough for real-world code;
- Based on different approaches:
  - Abstract interpretation;
  - Symbolic execution;
  - Testing.
- Work best on whole programs;
- Start from the main entry point of the program.

Single Function Analysis

Common scenario: Analysis of single functions.

- Third-party software (*e.g.* libraries);
- Core/Critical functions only.

Q: How to analyze single functions?

```
int foo (int *a, size_t size) {
  /* some interesting computation */
}
```

Single Function Analysis

Common scenario: Analysis of single functions.

- Third-party software (*e.g.* libraries);
- Core/Critical functions only.

Q: How to analyze single functions?

```
int foo (int *a, size_t size) {
  /* some interesting computation */
}
```

A: Set the function in question as the entry point (here, `foo`).

Single Function Analysis

Common scenario: Analysis of single functions.

- Third-party software (*e.g.* libraries);
- Core/Critical functions only.

Q: How to analyze single functions?

```
int foo (int *a, size_t size) {
  /* some interesting computation */
}
```

A: Set the function in question as the entry point (here, `foo`).

Outcome

Mostly imprecise and useless analysis results.

Function Context

Bottom Line

Analyzing single functions requires an appropriate context.

Function context:

- Initialization of function parameters and globals;
- Actual entry point to start the analysis from.

Function Context

Bottom Line

Analyzing single functions requires an appropriate context.

Function context:

- Initialization of function parameters and globals;
- Actual entry point to start the analysis from.

Common approaches:

- Write the context by-hand: Error-prone;

Function Context

Bottom Line

Analyzing single functions requires an appropriate context.

Function context:

- Initialization of function parameters and globals;
- Actual entry point to start the analysis from.

Common approaches:

- Write the context by-hand: Error-prone;
- Make analysis tools support a specification language: Arduous (if ever possible) and to be done for every tool.

This Work

Idea

Automatic contexts generation from formal specifications.

Contributions:

- System of inference rules for computing symbolic ranges;
- Precise and sound formalization;
- Prototype implementation as Frama-C plug-in.

Outline

Frama-C

- Suite of tools for the source code analysis of C programs;
- Extensible and collaborative platform:
  - Modular plug-in architecture based on a common kernel;
  - Combination of analysis to provide more precise results.
- Main available analysis:
  - Variable variation domains via abstract interpretation;
  - Deductive verification via weakest precondition calculus.
- Frama-C is open source software.

ACSL: ANSI/ISO C Specification Language

- Behavioral specification language for C programs;
- Specifications via code annotations of the form `/*@ ... */`;
- Function contracts given by pre- and postconditions:

```
/*@ requires \valid(a);
  @ requires 0 <= size <= 32;
  @ requires size % 16 == 0;
  @ ensures \forall integer i;
            0 <= i < size ==> *(a+i) == 0;
 */
int foo (int *a, size_t size) { ... }
```

ACSL: ANSI/ISO C Specification Language

- Behavioral specification language for C programs;
- Specifications via code annotations of the form `/*@ ... */`;
- Function contracts given by pre- and postconditions:

```
/*@ requires \valid(a);
  @ requires 0 <= size <= 32;
  @ requires size % 16 == 0;
  @ ensures \forall integer i;
            0 <= i < size ==> *(a+i) == 0;
 */
int foo (int *a, size_t size) { ... }
```

- This work considers preconditions only (*i.e.* **`requires`**).

Core Specification Language

$$
\begin{array}{rll}
P ::= & T\ \{\equiv, \leq, <\}\ T & \text{term comparison} \\
| & \texttt{defined}(M) & M \text{ is defined} \\
| & P \wedge P \mid P \vee P \mid \neg P & \text{logic formula} \\
T ::= & z & \text{integer constant } (z \in \mathbb{Z}) \\
| & M & \text{memory value} \\
| & T\ \{+, -, \times, /, \%\}\ T & \text{arithmetic operation} \\
M ::= & L & \text{left value} \\
| & M\ ++\ T & \text{displacement} \\
L ::= & x & \text{C variable} \\
| & \star M & \text{dereference}
\end{array}
$$

Outline

How to turn a precondition into a context?

```
/*@ requires defined(buf + (0..size-1));
  @ requires 4 <= size <= 16;
  @ requires size % 2 == 0;
  @ requires *(buf + n) == 0xC0000001; */
int bar (int *buf, int size, int n)
```

How to turn a precondition into a context?

```
/*@ requires defined(buf + (0..size-1));
  @ requires 4 <= size <= 16;
  @ requires size % 2 == 0;
  @ requires *(buf + n) == 0xC0000001; */
int bar (int *buf, int size, int n)
```

First attempt: Directly implement predicates one-by-one.

- Declare and properly initialize each left value involved;
- Turn term comparisons into conditionals.

```
int size;
make_int(&size, 1);
int* buf = (int*) malloc(size * sizeof(int));
make_int(buf, size);
if (4 <= size) && (size <= 16) {
  ...
}
```

How to turn a precondition into a context?

```
/*@ requires defined(buf + (0..size-1));
  @ requires 4 <= size <= 16;
  @ requires size % 2 == 0;
  @ requires *(buf + n) == 0xC0000001; */
int bar (int *buf, int size, int n)
```

First attempt: Directly implement predicates one-by-one.

- Declare and properly initialize each left value involved;
- Turn term comparisons into conditionals.

```
int size;
make_int(&size, 1);
int* buf = (int*) malloc(size * sizeof(int));
make_int(buf, size);
if (4 <= size) && (size <= 16) {
  ...
}
```

Shortcoming: Erratic dependencies among left values.

How to turn a precondition into a context?

```
/*@ requires defined(buf + (0..size-1));
  @ requires 4 <= size <= 16;
  @ requires size % 2 == 0;
  @ requires *(buf + n) == 0xC0000001; */
int bar (int *buf, int size, int n)
```

Revision: Pre-compute dependency graph among left values.

```
int size, n;
make_int(&size, 1);
make_int(&n, 1);
if (4 <= size) && (size <= 16) {
  if (size % 2 == 0) {
    int* buf = (int*) malloc(size * sizeof(int));
    make_int(buf, size);
    *(buf + n) = 0xC0000001;
    bar(buf, size, n);
  }
}
```

How to turn a precondition into a context?

```
/*@ requires defined(buf + (0..size-1));
  @ requires 4 <= size <= 16;
  @ requires size % 2 == 0;
  @ requires *(buf + n) == 0xC0000001; */
int bar (int *buf, int size, int n)
```

Revision: Pre-compute dependency graph among left values.

```
int size, n;
make_int(&size, 1);
make_int(&n, 1);
if (4 <= size) && (size <= 16) {
  if (size % 2 == 0) {
    int* buf = (int*) malloc(size * sizeof(int));
    make_int(buf, size);
    *(buf + n) = 0xC0000001;
    bar(buf, size, n);
  }
}
```

Shortcoming: Relation between `size` and `n` is overlooked.

Simplify Predicates into State Constraints

Each predicate is turned into:

- Symbolic variation domain computed for every left value;
- Side-condition to be checked at runtime (*i.e.* runtime check).

Simplify Predicates into State Constraints

Each predicate is turned into:

- Symbolic variation domain computed for every left value;
- Side-condition to be checked at runtime (*i.e.* runtime check).

```
/*@ requires defined(buf + (0..size-1)); // (1)
  @ requires 4 <= size <= 16;           // (2)
  @ requires size % 2 == 0;             // (3)
  @ requires *(buf + n) == 0xC0000001;  // (4) */
  int bar (int *buf, int size, int n)
```

Simplify Predicates into State Constraints

Each predicate is turned into:

- Symbolic variation domain computed for every left value;
- Side-condition to be checked at runtime (*i.e.* runtime check).

```
/*@ requires defined(buf + (0..size-1)); // (1)
  @ requires 4 <= size <= 16;           // (2)
  @ requires size % 2 == 0;             // (3)
  @ requires *(buf + n) == 0xC0000001;  // (4) */
  int bar (int *buf, int size, int n)
```

- From `(1)`: $\{\texttt{buf} \mapsto [0, \texttt{size} - 1], \texttt{size} \mapsto [-\infty, +\infty]\}$;

Simplify Predicates into State Constraints

Each predicate is turned into:

- Symbolic variation domain computed for every left value;
- Side-condition to be checked at runtime (*i.e.* runtime check).

```
/*@ requires defined(buf + (0..size-1)); // (1)
  @ requires 4 <= size <= 16;           // (2)
  @ requires size % 2 == 0;             // (3)
  @ requires *(buf + n) == 0xC0000001;  // (4) */
  int bar (int *buf, int size, int n)
```

- From `(1)`: $\{\texttt{buf} \mapsto [0, \texttt{size} - 1], \texttt{size} \mapsto [-\infty, +\infty]\}$;
- From `(2)`: $\texttt{size} \mapsto [-\infty, +\infty] \sqcap [4, 16] = [4, 16]$;

Simplify Predicates into State Constraints

Each predicate is turned into:

- Symbolic variation domain computed for every left value;
- Side-condition to be checked at runtime (*i.e.* runtime check).

```
/*@ requires defined(buf + (0..size-1)); // (1)
  @ requires 4 <= size <= 16;           // (2)
  @ requires size % 2 == 0;             // (3)
  @ requires *(buf + n) == 0xC0000001;  // (4) */
  int bar (int *buf, int size, int n)
```

- From `(1)`: $\{\texttt{buf} \mapsto [0, \texttt{size} - 1], \texttt{size} \mapsto [-\infty, +\infty]\}$;
- From `(2)`: $\texttt{size} \mapsto [-\infty, +\infty] \sqcap [4, 16] = [4, 16]$;
- From `(3)`: $\texttt{size} \mapsto [4, 16], 0\%2$;

Simplify Predicates into State Constraints

Each predicate is turned into:

- Symbolic variation domain computed for every left value;
- Side-condition to be checked at runtime (*i.e.* runtime check).

```
/*@ requires defined(buf + (0..size-1)); // (1)
  @ requires 4 <= size <= 16;           // (2)
  @ requires size % 2 == 0;             // (3)
  @ requires *(buf + n) == 0xC0000001;  // (4) */
  int bar (int *buf, int size, int n)
```

- From `(1)`: $\{\texttt{buf} \mapsto [0, \texttt{size} - 1], \texttt{size} \mapsto [-\infty, +\infty]\}$;
- From `(2)`: $\texttt{size} \mapsto [-\infty, +\infty] \sqcap [4, 16] = [4, 16]$;
- From `(3)`: $\texttt{size} \mapsto [4, 16], 0\%2$;
- From `(4)`:

$$\left\{\begin{array}{l} \texttt{n} \mapsto [-\infty, +\infty], *(\texttt{buf+n}) \mapsto [\texttt{0xC0000001}, \texttt{0xC0000001}], \\ \texttt{buf} \mapsto [0, \texttt{size} - 1] \sqcup [0, \texttt{n}] = [0, \texttt{max}(\texttt{size} - 1, \texttt{n})] \end{array}\right\}$$

Some Details

State Constraints:

- Encode requirements on each left value L;
- Defined as a pair (R, X):
  - R is the symbolic range of runtime values for L;
  - X is a set of runtime checks to be verified as conditions on L.

Symbolic Range Domain $(\mathbb{R}, \sqsubseteq)$:

- Usual range (or interval) domain, but on symbolic ranges R;
- Usual join ($\sqcup$) and meet ($\sqcap$) operators, and in particular:

$$[E_1, E_2] \sqcup [E_3, E_4] = [\texttt{min}(E_1, E_3), \texttt{max}(E_2, E_4)]$$
$$[E_1, E_2] \sqcap [E_3, E_4] = [\texttt{max}(E_1, E_3), \texttt{min}(E_2, E_4)]$$

Inferring State Constraints

Simplification judgments $\Sigma \vdash P \Rightarrow \Sigma'$:

- P is a predicate literal,
- Σ, Σ' association maps from left values to state constraints.

Inference rules:

- Describe how to update Σ into Σ' for the left values in P;
- Build dependency graph $\mathcal{G}$ among left values;
- Assume formulæ in DNF, but no rule for disjunctions.

Inferring State Constraints

Simplification judgments $\Sigma \vdash P \Rightarrow \Sigma'$:

- P is a predicate literal,
- Σ, Σ' association maps from left values to state constraints.

Inference rules:

- Describe how to update Σ into Σ' for the left values in P;
- Build dependency graph $\mathcal{G}$ among left values;
- Assume formulæ in DNF, but no rule for disjunctions.

Theorem (Soundness)

For all conjunctive $\mathcal{C}$, either $\emptyset \vdash \mathcal{C} \Rightarrow \Sigma$ and $\Sigma \models \mathcal{C}$, or it fails.

Outline

Generation Scheme

For each conjunctive clause $\mathcal{C}$, consider its $(\Sigma, \mathcal{G})$:

- Topologically iterate over left values of $\mathcal{G}$;
- For every visited L, consider its state constraint (R, X):
  - Initialize L with R by using `make_range`;
  - Guard the rest of the code under conditionals implementing X.
- Repeat till last left value, then generate function call.

Generation Scheme

For each conjunctive clause $\mathcal{C}$, consider its $(\Sigma, \mathcal{G})$:

- Topologically iterate over left values of $\mathcal{G}$;
- For every visited L, consider its state constraint (R, X):
  - Initialize L with R by using `make_range`;
  - Guard the rest of the code under conditionals implementing X.
- Repeat till last left value, then generate function call.

For precondition formulæ $\bigvee_{i=1}^{n} \mathcal{C}_i$:

int `clauses` = `make_range`(1, `n`);
switch (`clauses`) {
  case 1 : { $[\![\mathcal{C}_1]\!]$; **break**; }
  ...
  case `n` : { $[\![\mathcal{C}_n]\!]$; **break**; }
}

Formal specification:

```
/*@ requires defined(buf + (0..size-1));
  @ requires 4 <= size <= 16;
  @ requires size % 2 == 0;
  @ requires *(buf + n) == 0xC0000001; */
int bar (int *buf, int size, int n)
```

Analysis context:

```
1   int bar_context (void){
2     int n;
3     Frama_C_make_unknown(&n, sizeof(int));
4     int size = Frama_C_int_interval(4, 16);
5     if (size % 2 == 0) {
6       int max = size > n ? size : n;
7       int* buf = (int*) malloc(max * sizeof(int));
8       if (buf != (int*) 0) {
9         Frama_C_make_unknown(buf, max * sizeof(int));
10        *(buf + n) = 0xC0000001;
11        bar(buf, size, n);
12      }
13    }
14    return 0;
15  }
```

Conclusions

- Single function analysis requires a context to be useful;
- This talk has shown:
  - Method to generate analysis contexts from formal specifications;
  - Precise and sound formalization.
- Implemented as a Frama-C plug-in (used at TrustInSoft).

Thanks!