AAA615: Formal Methods

Lecture 7 — SAT-based Program Analysis

Hakjoo Oh
2017 Fall

Bit-Level Static Analysis via Boolean Satisfiability

- Finding bugs in low-level systems code is challenging due to difficult-to-model language constructs such as pointers, bit-wise operators, and type casts, which requires bit-level reasoning.
- One successful approach is to encode program semantics at bit-level and translate it to a boolean satisfiability formula, exploiting impressive advances in solving boolean satisfiability.
- References:
  - A Tool for Checking ANSI-C Programs. TACAS 2004.
  - Scalable Error Detection using Boolean Satisfiability. POPL 2005.

Saturn[1]

- A static error detection tool based on boolean satisfiability.
  - Yichen Xie and Alex Aiken. Scalable Error Detection using Boolean Satisfiability. POPL 2005.
- The analysis is path-sensitive, precise down to bit level, and models pointers and heap data. Yet highly scalable thanks to modern SAT solvers and various optimization techniques.
- In particular, Saturn performs a bottom-up analysis, which computes a summary of each analyzed function and reuses the summary when the function is called later.

[1] SATisfiability-based failURe aNalysis

A Low-Level Programming Language

A subset of the language that does not include structures and pointers:

$$
\begin{array}{rcl}
\tau & \rightarrow & (n, \textbf{signed} \mid \textbf{unsigned}) \\
e & \rightarrow & \textbf{unknown}(\tau) \mid \textbf{const}(n, \tau) \mid x \mid -e \mid \; !e \mid e_1 \oplus e_2 \\
 & & \mid e_1 \textbf{ band } e_2 \mid e_1 \textbf{ lshift } e_2 \mid (\tau)e \mid \textbf{lift}_e(c, \tau) \\
c & \rightarrow & \textbf{true} \mid \textbf{false} \mid \neg c \mid e_1 = e_2 \mid c_1 \wedge c_2 \mid c_1 \vee c_2 \mid \textbf{lift}_c(e) \\
s & \rightarrow & \textbf{skip} \mid x := e \mid \textbf{assert}(c) \mid \textbf{if } c \; s_1 \; s_2 \mid \textbf{while } c \; s \mid s_1; s_2
\end{array}
$$

Representation:

$$
\begin{array}{rcl}
\beta & \rightarrow & [b_{n-1} \ldots b_0]_s \quad \text{where } s \in \{\textbf{signed}, \textbf{unsigned}\} \\
b & \rightarrow & 0 \mid 1 \mid \alpha \mid b_1 \wedge b_2 \mid b_1 \vee b_2 \mid \neg b
\end{array}
$$

Translation Rules (Expressions)

$$\frac{\beta = \psi(x)}{\psi \vdash x \overset{\mathtt{E}}{\Rightarrow} \beta} \qquad \frac{(n, s) = \tau}{\psi \vdash \textbf{unknown}(\tau) \overset{\mathtt{E}}{\Rightarrow} [\alpha_{n-1} \ldots \alpha_0]_s}\ \alpha_i \text{ fresh}$$

$$\frac{\begin{array}{c}\psi \vdash e \overset{\mathtt{E}}{\Rightarrow} [b_{n-1} \ldots b_0]_{s'} \\ \tau = (m, s)\end{array} \qquad b'_i = \begin{cases} b_i & \text{if } 0 \leq i < n \\ 0 & \text{if } s = \textbf{unsigned} \ \wedge n \leq i < m \\ b_{n-1} & \text{if } s = \textbf{signed} \ \wedge n \leq i < m \end{cases}}{\psi \vdash (\tau)\ e \overset{\mathtt{E}}{\Rightarrow} [b'_{m-1} \ldots b'_0]_s}$$

$$\frac{(n, s) = \tau \quad \psi \vdash c \overset{\mathtt{C}}{\Rightarrow} b}{\psi \vdash \textbf{lift}_e(c, \tau) \overset{\mathtt{E}}{\Rightarrow} [0 \ldots 0b]_s}$$

$$\frac{\psi \vdash e \overset{\mathtt{E}}{\Rightarrow} [b_{n-1} \ldots b_0]_s \quad \psi \vdash e' \overset{\mathtt{E}}{\Rightarrow} [b'_{n-1} \ldots b'_0]_s}{\psi \vdash e \textbf{ band } e' \overset{\mathtt{E}}{\Rightarrow} [b_{n-1} \wedge b'_{n-1} \ldots b_0 \wedge b'_0]_s}$$

Translation Rules (Conditionals)

$$\frac{}{\psi \vdash \textbf{true} \stackrel{\mathtt{C}}{\Rightarrow} 1} \qquad \frac{}{\psi \vdash \textbf{false} \stackrel{\mathtt{C}}{\Rightarrow} 0}$$

$$\frac{\psi \vdash c \stackrel{\mathtt{C}}{\Rightarrow} b}{\psi \vdash \neg c \stackrel{\mathtt{C}}{\Rightarrow} \neg b} \qquad \frac{\psi \vdash e \stackrel{\mathtt{E}}{\Rightarrow} [b_{n-1} \ldots b_0]_s \quad \psi \vdash e' \stackrel{\mathtt{E}}{\Rightarrow} [b'_{n-1} \ldots b'_0]_s}{\psi \vdash e_1 = e_2 \stackrel{\mathtt{C}}{\Rightarrow} \bigwedge_i (b_i \wedge b'_i) \vee (\neg b_i \wedge \neg b'_i)}$$

$$\frac{\psi \vdash c_1 \stackrel{\mathtt{C}}{\Rightarrow} b_1 \quad \psi \vdash c_2 \stackrel{\mathtt{C}}{\Rightarrow} b_2}{\psi \vdash c_1 \wedge c_2 \stackrel{\mathtt{C}}{\Rightarrow} b_1 \wedge b_2} \qquad \frac{\psi \vdash c_1 \stackrel{\mathtt{C}}{\Rightarrow} b_1 \quad \psi \vdash c_2 \stackrel{\mathtt{C}}{\Rightarrow} b_2}{\psi \vdash c_1 \vee c_2 \stackrel{\mathtt{C}}{\Rightarrow} b_1 \vee b_2}$$

$$\frac{\psi \vdash e \stackrel{\mathtt{E}}{\Rightarrow} [b_{n-1} \ldots b_0]_s}{\psi \vdash \textbf{lift}_c(e) \stackrel{\mathtt{C}}{\Rightarrow} \bigvee_i b_i}$$

Translation Rules (Statements)

$$\frac{\psi \vdash e \stackrel{\mathtt{E}}{\Rightarrow} \beta}{\mathcal{G}, \psi \vdash x := e \stackrel{\mathtt{S}}{\Rightarrow} \mathcal{G}, \psi[x \mapsto \beta]}$$

$$\frac{\psi \vdash c \stackrel{\mathtt{C}}{\Rightarrow} b \quad (\mathcal{G} \wedge \neg b) \text{ unsatisfiable}}{\mathcal{G}, \psi \vdash \textbf{assert}(c) \stackrel{\mathtt{S}}{\Rightarrow} \mathcal{G}, \psi}$$

$$\frac{\mathcal{G} \wedge c, \psi \vdash s_1 \stackrel{\mathtt{S}}{\Rightarrow} \mathcal{G}_1, \psi_1 \quad \mathcal{G} \wedge \neg c, \psi \vdash s_2 \stackrel{\mathtt{S}}{\Rightarrow} \mathcal{G}_2, \psi_2}{\mathcal{G}, \psi \vdash \textbf{if } c\ s_1\ s_2 \stackrel{\mathtt{S}}{\Rightarrow} \mathcal{G}_1 \vee \mathcal{G}_2, \lambda x.[(\mathcal{G}_1 \wedge b_m) \vee (\mathcal{G}_2 \wedge b'_m) \dots]}$$

where $\psi_1(x) = [b_m \dots b_0]_s, \psi_2(x) = [b'_m \dots b'_0]_s$

$$\frac{\mathcal{G}, \psi \vdash \textbf{if } c\ s\ \textbf{skip}; \textbf{if } c\ s\ \textbf{skip} \stackrel{\mathtt{S}}{\Rightarrow} \mathcal{G}', \psi'}{\mathcal{G}, \psi \vdash \textbf{while } c\ s \stackrel{\mathtt{S}}{\Rightarrow} \mathcal{G}', \psi'}$$

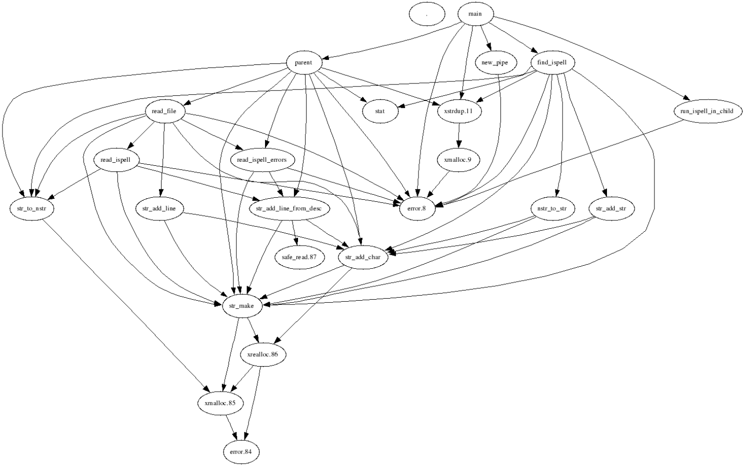
Structures and Pointers

- The extension with structures is rather straightforward.
- Translation of pointers is more involved and maintains path sensitivity for heap locations by introducing *guarded locations*.
  - A pointer points to a set of guarded locations.
  - A guarded location is a location associated with a boolean guard that represents the condition under which the points-to relationship holds.

$$
\begin{array}{ll}
\textbf{if } (c)\ p = \&x; & //\ p : \{(\textbf{true}, x)\} \\
\textbf{else } p = \&y; & //\ p : \{(\textbf{true}, y)\} \\
*p = 3; & //\ p : \{(c, x), (\neg c, y)\}
\end{array}
$$

Interprocedural Analysis

- A common technique is inlining. However, inlining incurs exponential blow-up, not practical for large software systems.
- Saturn uses a bottom-up, summary-based interprocedural analysis.



Experimental Results

- Saturn can be instantiated into various error detection tools by defining appropriate function summaries.
- E.g., memory leak detection, lock checking (e.g. double lock/unlock)
- In Linux kernel (4.8 MLoC), Saturn found 179 lock-related errors with a false positive rate of 40%.