High-Level Separation Logic for Low-Level Code

Jonas B. Jensen\textsuperscript{1},
Nick Benton\textsuperscript{2} and Andrew Kennedy\textsuperscript{2}

\textsuperscript{1} IT University of Copenhagen
\textsuperscript{2} Microsoft Research Cambridge

February 21, 2013
Introduction

- A Coq model of x86
  - Lets us generate, specify and verify machine-code programs.
  - Lets us boot a PC from extracted list of bytes.
- Two main messages:
  - Low-level machine code needs high-level separation logic
  - A single program logic can span several layers of abstraction
Overview of this talk

1. Motivation and background (triples are not general enough)
2. Explicit specifications (this is general but unmanageable)
3. Specification logic (this is more manageable)
4. Structured code (triples are back)
5. Read-only frames (structural rules generalise)
Traditional Hoare logic struggles with aliasing.

$$\{ x \mapsto 1 \land y \mapsto 2 \} \ [x] := 3 \ {\{ x \mapsto 3 \land (x \neq y \Rightarrow y \mapsto 2) \}$$

Separation logic makes non-aliasing of pointers the default.

$$\{ x \mapsto 1 \ast y \mapsto 2 \} \ [x] := 3 \ {\{ x \mapsto 3 \ast y \mapsto 2 \}$$

Derived using the frame rule

$$\begin{array}{c}
\{P\} \ c \ \{Q\} \\
\{P \ast R\} \ c \ \{Q \ast R\}
\end{array}$$
Problem and solution sketch

- High-level logics inherit structure. Machine code almost entirely unstructured.
  - Many entries and exits, code in heap, code pointers, no termination, no built-in allocation or procedures
  - Triples $\{P\} \ c \ \{Q\}$ are unsuitable here

- We break the triple into components that live in a specification logic. This logic
  - is as unstructured as assembly code,
  - has a (higher order) frame rule,
  - handles step-indexing implicitly,
  - can be used to derive standard triples
Machine model

Single-core 32-bit x86, ignoring interrupts and virtual memory.

- Code is data
- Arithmetic modulo $2^{32}$
- Flat, total memory

\[ S \triangleq (\text{reg} \rightarrow \text{DWORD}) \times (\text{flag} \rightarrow \{\text{true}, \text{false}, \text{undef}\}) \times (\text{DWORD} \rightarrow (\text{BYTE} \uplus \{\text{unmapped}\})) \]

\[ \text{step} : S \rightarrow S \]
Assertion logic

\[
\text{asn} \triangleq \mathcal{P}(\Sigma)
\]

\[
\Sigma \triangleq (\text{reg} \rightarrow \text{DWORD}) \times \\
(\text{flag} \rightarrow \{\text{true, false, undef}\}) \times \\
(\text{DWORD} \rightarrow (\text{BYTE} \uplus \{\text{unmapped}\}))
\]

\[
r \mapsto v \triangleq \{(r \mapsto v), [], []\}
\]

\[
f \mapsto b \triangleq \{[], [f \mapsto b], []\}
\]

\[
i..j \mapsto x \triangleq \{([], [])\} \times \text{“memory at } i..j \text{ is } x\”
\]

\[
i \mapsto x \triangleq \exists j. i..j \mapsto x
\]
Assertion logic connectives

\[ \forall x: T. P(x) \triangleq \bigcap_{x: T} P(x) \]

\[ \exists x: T. P(x) \triangleq \bigcup_{x: T} P(x) \]

\[ P \Rightarrow Q \triangleq \{ \sigma : \Sigma \mid \sigma \in P \Rightarrow \sigma \in Q \} \]

\[ \text{emp} \triangleq \{([], [], [])\} \]

\[ P \ast Q \triangleq \{ \sigma : \Sigma \mid \exists \sigma_1, \sigma_2. \sigma = \sigma_1 \uplus \sigma_2 \land \sigma_1 \in P \land \sigma_2 \in Q \} \]

\[ P \vdash Q \triangleq P \subseteq Q \]
Safety

- Standard Hoare triple \( \{ P \} \ c \ \{ Q \} \) unsuitable for machine code (unstructured, non-terminating, code is data).
- Instead, we define

\[
safe \triangleq \{(k, P) : \mathbb{N} \times \text{asn} | \forall \sigma \in P. \forall s \supseteq \sigma. \text{step}^k(s) \text{ defined}\}
\]

- Example: \( \forall k, i. \ (k, (EIP \mapsto i \ast i \mapsto jmp \ i)) \in safe \)
Wrapping \textit{safe} in logic

• First attempt at specifying \textit{jmp}:

\[
\begin{align*}
\text{For all } i, j, k, R, \\
\text{if } (k, EIP \mapsto j \ast i \mapsto jmp \ j \ast R) \in safe \\
\text{then } (k + 1, EIP \mapsto i \ast i \mapsto jmp \ j \ast R) \in safe
\end{align*}
\]

• We want to abbreviate that as

\[
\vdash safe \otimes (EIP \mapsto j \ast i \mapsto jmp \ j) \vdash safe \otimes (EIP \mapsto i \ast i \mapsto jmp \ j)
\]

• Or even

\[
\vdash (\vdash safe \otimes (EIP \mapsto j) \Rightarrow safe \otimes (EIP \mapsto i)) \otimes i \mapsto jmp \ j
\]
Specification logic

\[ \text{spec} \triangleq \{ S : \mathcal{P}(\mathbb{N} \times \text{asn}) \mid \forall (k, P) \in S. \ \forall k' \leq k. \ \forall R. \ (k', P \ast R) \in S \} \]

Notice that \( \text{safe} \in \text{spec} \).

\[ \forall x : T. \ S(x) \triangleq \bigcap_{x:T} S(x) \]
\[ \exists x : T. \ S(x) \triangleq \bigcup_{x:T} S(x) \]

\[ S \Rightarrow S' \triangleq \{ (k, P) \mid \forall k' \leq k. \ \forall R. \ (k', P \ast R) \in S \Rightarrow (k', P \ast R) \in S' \} \]

\[ \triangleright S \triangleq \{ (k, P) \mid \forall k' < k. \ (k', P) \in S \} \]

\[ S \vdash S' \triangleq S \subseteq S' \]
The frame rule

- Standard frame rule is

\[
\begin{array}{c}
\{P\} \proves c \{Q\} \\
\{P \ast R\} \proves c \{Q \ast R\}
\end{array}
\]

- We have instead the higher-order frame rule

\[
S \proves S \otimes R
\]

- Where

\[
S \otimes R \triangleq \{(k, P) | (k, P \ast R) \in S\}
\]
Using $\otimes$

- $\otimes$ distributes over every connective and quantifier; e.g.,
\[
(S \Rightarrow S') \otimes R \equiv (S \otimes R) \Rightarrow (S' \otimes R)
\]

- $\otimes$ is a monoid action of $asn$ on $spec$:
\[
(S \otimes R_1) \otimes R_2 \equiv S \otimes (R_1 \ast R_2)
\]

- Example:
\[
\vdash (\triangleright safe \otimes (EIP \hookrightarrow j)) \Rightarrow safe \otimes (EIP \hookrightarrow i)) \otimes i \mapsto jmp j
\]
\[
\vdash (\triangleright safe \otimes (EIP \hookrightarrow j)) \Rightarrow safe \otimes (EIP \hookrightarrow i)) \otimes i \mapsto jmp j \otimes R
\]
\[
\vdash (\triangleright safe \otimes (EIP \hookrightarrow j \ast R)) \Rightarrow safe \otimes (EIP \hookrightarrow i \ast R)) \otimes i \mapsto jmp j
\]
Hoare triples

- For code that behaves like a basic block,

\[
\{P\} \; c \; \{Q\} \approx \forall i, j. (\text{safe} \otimes (EIP \mapsto j \ast Q) \Rightarrow \\
\text{safe} \otimes (EIP \mapsto i \ast P)) \otimes \, i \ldots j \mapsto c
\]

- Example: \( \vdash \{r \mapsto _{}\} \; \text{mov} \; r, v \; \{r \mapsto v\} \)

- Follows the standard structural rules:

\[
\begin{align*}
\text{If } & P \vdash P' \quad S \vdash \{P'\} \; c \; \{Q'\} \quad Q' \vdash Q \\
& \text{then } S \vdash \{P\} \; c \; \{Q\}
\end{align*}
\]

\[
\begin{align*}
\text{If } & S \vdash \{P\} \; c \; \{Q\} \\
& \text{then } S \vdash \{P \ast R\} \; c \; \{Q \ast R\}
\end{align*}
\]

\[
\begin{align*}
\text{If } & S \vdash \forall x. \{P(x)\} \; c \; \{Q\} \\
& \text{then } S \vdash \{\exists x. \, P(x)\} \; c \; \{Q\}
\end{align*}
\]

- Permits internally-unstructured code
Assembly labels and macros

- Writing machine code in practice requires an assembler
- An assembler provides labels and macros
- Our assembler uses this little language:

\[ p ::= \_ | \text{skip} | p; p | l: | \text{LOCAL } l; \ p(l) \]

- A macro is just a Coq definition

\[
\text{if } (t, b) \text{ then } p_1 \text{ else } p_2 \triangleq \text{LOCAL } l_1, l_2; \\
\ jcc \ t, b, l_1; \\
\ p_2; \\
\ jmp \ l_2; \\
\ l_1: p_1; \\
\ l_2:
\]

- We can put programs \( p \) in triples (\( \{P\} \ p \ \{Q\} \)).
Rules for structured programs

\[ \vdash \{ P \} \text{skip} \{ P \} \]

\[ S \vdash \{ P \} \ p_1 \ \{ Q \} \quad S \vdash \{ Q \} \ p_2 \ \{ R \} \]

\[ S \vdash \{ P \} \ p_1; \ p_2 \ \{ R \} \]

\[ S \vdash \{ P(b) \ast \text{cond}(t, b) \} \ p_1 \ \{ Q \} \]
\[ S \vdash \{ P(\neg b) \ast \text{cond}(t, \neg b) \} \ p_2 \ \{ Q \} \]

\[ S \vdash \{ \exists b'. \ P(b') \ast \text{cond}(t, b') \} \text{ if } (t, b) \text{ then } p_1 \text{ else } p_2 \ \{ Q \} \]

\[ \triangleright f \mapsto \{ P \} \{ Q \} \vdash \{ P \} \text{ call } f \ \{ Q \} \otimes EDX \mapsto - \]
Recall the tentative definition of Hoare triples:

\[
\{ P \} \ c \ \{ Q \} \approx \forall i, j. (\text{safe} \otimes (\text{EIP} \mapsto j \ast Q) \Rightarrow \\
\text{safe} \otimes (\text{EIP} \mapsto i \ast P)) \otimes i..j \mapsto c
\]

We actually define that as

\[
\{ P \} \ c \ \{ Q \} \triangleq \forall i, j. (\text{safe} \otimes (\text{EIP} \mapsto j \ast Q) \Rightarrow \\
\text{safe} \otimes (\text{EIP} \mapsto i \ast P)) \otimes i..j \mapsto c
\]

Where \( S \otimes R \triangleq \forall \sigma \in R. S \otimes \{ \sigma \} \)
Structural rules for $\otimes$

Frame rule

$$
\frac{
\Gamma \vdash \{ P \} \ c \ \{ Q \}
}{
\Gamma \vdash \{ P \ast R \} \ c \ \{ Q \ast R \}
}
\quad
\frac{
\Gamma \vdash S
}{
\Gamma \vdash S \otimes R
}
$$

Rule of consequence

$$
\frac{
P \vdash P' \quad \Gamma \vdash \{ P' \} \ c \ \{ Q' \} \quad Q' \vdash Q
}{
\Gamma \vdash \{ P \} \ c \ \{ Q \}
}
\quad
\frac{
P \vdash P' \quad \Gamma \vdash S' \otimes P' \quad S' \vdash S
}{
\Gamma \vdash S \otimes P
}
$$

Existential rule

$$
\frac{
\Gamma \vdash \forall x. \{ P(x) \} \ c \ \{ Q \}
}{
\Gamma \vdash \{ \exists x. \ P(x) \} \ c \ \{ Q \}
}
\quad
\frac{
\Gamma \vdash \forall x. \ S \otimes P(x)
}{
\Gamma \vdash S \otimes (\exists x. \ P(x))
}
$$

The story on these rules for $\otimes$ is longer; see paper
Example: toy memory allocator

(* info: Src is pointer to two-word heap information block
  n: nat representing number of bytes to be allocated
  fail: DWORD is label to branch to on failure
*)

Definition allocImp info (n: nat) (fail: DWORD) : program :=
  mov ESI, info;;
  mov EDI, [ESI];;
  add EDI, n;;
  jc fail;; (* Wrapped around *)
  cmp [ESI+4], EDI;;
  jc fail;; (* Exceeded memory limit *)
  mov [ESI], EDI.

Definition allocSpec n fail inv code :=
  Forall i, Forall j, (  
    safe @ (EIP ~= fail ** EDI?) \/\
    safe @ (EIP ~= j ** Exists p, EDI ~= p +# n ** memAny p (p +# n))
    -->>
    safe @ (EIP ~= i ** EDI?)
  )
  @ (ESI? ** OSZCP.Any ** inv)
  <@ (i -- j :-> code).
Conclusions

- A single, general specification logic provides building blocks for higher-level abstractions
- We have a frame rule for completely unstructured programs
- We also have rules of consequence and exists without triples
- All formalised in Coq, with tactic support
Related work

- Myreen et al.
  - Separation logic for x86/ARM in HOL4
  - Extends triples with address transformers, sets, ...
  - Total correctness (what happens after reaching $Q$?)

- Shao et al.
  - Separation logic for x86 (and more) in Coq
  - Family of specialised logics (XCAP, GCAP, SCAP, ISCAP, ...)
  - Most have built-in allocation, call, etc.

- Chlipala et al.
  - Separation logic for assembly code in Coq
  - No frame rule
  - Emphasis on automation over expressiveness
Triple-like patterns

- Example: jump if zero

\[ \vdash (\triangleright safe \otimes (b \land EIP \mapsto i') \land \\
safe \otimes (\neg b \land EIP \mapsto j) \Rightarrow \\
safe \otimes (EIP \mapsto i)) \]
\[ \otimes (ZF \mapsto b \ast i..j \mapsto jz \ i') \]

- Example: procedure specification

\[ f \mapsto \{P\}\{Q\} \triangleq \forall i_{ret}. \ safe \otimes (EIP \mapsto i_{ret} \ast EDX \mapsto _\ast Q) \Rightarrow \\
safe \otimes (EIP \mapsto f \ast EDX \mapsto i_{ret} \ast P) \]
Assembler correctness

Function \( assemble : DWORD \times program \rightarrow list(BYTE) \).

\[
i..j \mapsto assemble(i, p) \vdash i..j \mapsto p
\]

Here, \( i..j \mapsto p \) is defined recursively as follows.

\[
\begin{align*}
i..j & \mapsto (i) \; \triangleq i..j \mapsto i \\
i..j & \mapsto \text{skip} \; \triangleq i = j \land \text{emp} \\
i..j & \mapsto p_1; p_2 \; \triangleq \exists i'. \; i..i' \mapsto p_1 \ast i'..j \mapsto p_2 \\
i..j & \mapsto \text{LOCAL} \; l; \; p \; \triangleq \exists l. \; i..j \mapsto p(l) \\
i..j & \mapsto l: \; \triangleq i = j = l \land \text{emp}
\end{align*}
\]
Future work

- Model of I/O; e.g., screen/keyboard
- High-level model of processes
- Verified operating-system components such as scheduler, allocator, loader
- Eventual aim: process isolation theorem