Runtime Verification of C Memory Safety

Grigore Roșu

University of Illinois at Urbana-Champaign (UIUC)

joint work with

Traian Florin Serbanuta (UIUC) and
Wolfram Schulte (Microsoft Research)

June 27, 2009

Why is runtime verification of C memory safety interesting / challenging?

- ... that is, why not just run the program and if does not segfaults then conclude it is memory safe?
- Because that guarantees nothing about the memory safety of the program, due to input non-determinism, uninitialized variables, ...
- Moreover, it guarantees nothing about memory safety even when the program is closed (no input), variables are initialized, ...

Problem: Memory Nondeterminism

- Even closed programs can behave non-deterministically because of non-deterministic memory allocation and because one can use pointers as arbitrary integers in C

Memory safety (in C)

- A major source of bugs and security vulnerabilities
- Formal approaches usually require user input, such as program annotations; we want no user input here
- Current runtime analysis techniques: mostly through ad-hoc (good) engineering techniques; we want precision here

What we do:

- Memory safety based on formal, executable programming language semantics
- Why formal semantics? Rigorous definition of memory safety is impossible without rigorous language definition
- Why executable? Because we want no user annotations and want the program to get “stuck” in the memory unsafe operation

Outline

Formal definition of KERNELC

Memory safety

Runtime verification of memory safety

Outline

What is KERNELC?

- Fragment of C, including pointers and memory management

(abstract syntax)

$$
\begin{array}{rl}
\mathit{Int} & ::= \text{ integers} \\
\mathit{Id} & ::= \text{ identifiers, to be used as variable names} \\
K & ::= \mathit{Int} \mid \mathit{Id} \mid \texttt{null} \mid \texttt{*}K \mid \texttt{!}K \mid K_1 \ \mathrm{op}\ K_2 \mid K_1 \texttt{ \&\& } K_2 \mid K_1 \texttt{ || } K_2 \mid \\
& \quad \mid K_1 \texttt{=} K_2 \mid K\texttt{;} \mid K_1 \ K_2 \mid \texttt{\{}K\texttt{\}} \mid \texttt{\{\}} \mid \texttt{malloc(}K\texttt{)} \mid \texttt{free(}K\texttt{)} \\
& \quad \mid \texttt{if (}K_1\texttt{) } K_2 \mid \texttt{if (}K_1\texttt{) } K_2 \texttt{ else } K_3 \mid \texttt{while (}K_1\texttt{) } K_2
\end{array}
$$

(desugaring of non-core constructs)

$$
\begin{array}{ll}
\texttt{null} = 0 & \\
\texttt{!}\,K = \texttt{if (}K\texttt{) 0 else 1} & \\
K_1 \texttt{ \&\& } K_2 = \texttt{if (}K_1\texttt{) } K_2 \texttt{ else 0} & K_1 \texttt{ || } K_2 = \texttt{if (}K_1\texttt{) 1 else } K_2 \\
\texttt{\{}K\texttt{\}} = K & \texttt{if (}K_1\texttt{) } K_2 = \texttt{if (}K_1\texttt{) } K_2 \texttt{ else \{\}}
\end{array}
$$

A program in KERNELC

- One can use KERNELC to write quite interesting programs
- Such as an in-place list reverse algorithm

```
if(p) {
  x = *(p+1);
  *(p+1) = NULL ;
  while(x){
   y = *(x+1);
   *(x+1) = p;
   p = x;
   x = y;
  }
 }
```

Rewriting Logic Semantics and K

- Define a programming language as a rewrite theory (Σ, E, R)
- Algebraic signature Σ defines syntax for language and configurations
- Equations E define non-computational steps (structural identities)
- Rewrite rules R define computational steps
- K: A rewriting logic technique used to define arbitrarily complex languages using unconditional equations / rules
  - Everything executed via AC rewriting (we use Maude)

Strictness of KERNELC Constructs in K

- Add task sequentialization (associative operation):
  $K \ ::= \ \ldots \mid \mathrm{Seq}_{\curvearrowright}[K]$
- Add strictness attributes:

K_1 op K_2	`strict(`K_1, K_2`)`
X = K	`strict(`K`)`
`if (`K_1`)` K_2 `else` K_3	`strict(`K_1`)`
* K	`strict(`K`)`
* K_1 = K_2	`strict(`K_1, K_2`)`
`malloc(`K`)`	`strict(`K`)`
`free(`K`)`	`strict(`K`)`

- Strictness is syntactic sugar for equations, e.g.:
  $(*\, K_1 = K_2) = K_1 \curvearrowright (*\, \square = K_2)$
  $(*\, P_1 = K_2) = K_2 \curvearrowright (*\, P_1 = \square)$

KERNELC in K: Pointerless

$\mathit{Config} ::= \langle \mathrm{Bag}\_[\mathit{ConfigItem}] \rangle$ (configuration)

$\mathit{ConfigItem} ::= \langle K \rangle_k \mid \langle \mathit{Env} \rangle_{env}$

$K ::= \ldots \mid \mathrm{Seq}\_\curvearrowright\_[K]$ $\quad \mathit{Env} ::= \mathrm{Map}\_\text{'}\_[\mathit{Id}, \mathit{Int}]$

$\{\} = \cdot, \quad K\,; = K$ (semantics)

$K_1\ K_2 = K_1 \curvearrowright K_2$

$I_1\ \mathtt{op}\ I_2 \rightarrow I_1\ op_{Int}\ I_2$

$\mathtt{if}\,(I)\ K_2\ \mathtt{else}\ K_3 \rightarrow K_3$, where $I = 0$

$\mathtt{if}\,(I)\ K_2\ \mathtt{else}\ K_3 \rightarrow K_2$, where $I \neq 0$

$\langle X \curvearrowright K \rangle_k\ \langle X \mapsto I, \rho \rangle_{env} \rightarrow \langle I \curvearrowright K \rangle_k\ \langle X \mapsto I, \rho \rangle_{env}$

$\langle X{=}I \curvearrowright K \rangle_k\ \langle \rho \rangle_{env} \rightarrow \langle K \rangle_k\ \langle \rho[X \leftarrow I] \rangle_{env}$

$\langle \mathtt{while}\,(K_1)\ K_2 \curvearrowright K \rangle_k \rightarrow \langle \mathtt{if}\,(K_1)\ K_2\,;\mathtt{while}\,(K_1)\ K_2\ \mathtt{else}\ \cdot \curvearrowright K \rangle_k$

Pointer semantics for KERNELC

$$\begin{array}{l}
\textit{ConfigItem} ::= \ldots \mid \langle \textit{Mem} \rangle_{mem} \mid \langle \textit{Ptr} \rangle_{ptr} \quad \text{(configuration)} \\
\textit{Mem} ::= \text{Map}_{_}^{\circledast}{}_{_}[\textit{Nat}^{+}, \textit{Int}] \qquad \textit{Ptr} ::= \text{Map}_{_}^{\prime}{}_{_}[\textit{Nat}^{+}, \textit{Nat}] \\
\hline
\langle *P \curvearrowright K \rangle_k \; \langle P \mapsto I \circledast \sigma \rangle_{mem} \rightarrow \langle I \curvearrowright K \rangle_k \; \langle P \mapsto I \circledast \sigma \rangle_{mem} \quad \text{(semantics)} \\
\langle *P{=}I \curvearrowright K \rangle_k \; \langle P \mapsto I' \circledast \sigma \rangle_{mem} \rightarrow \langle K \rangle_k \; \langle P \mapsto I \circledast \sigma \rangle_{mem} \\
\langle \texttt{malloc}(N) \curvearrowright K \rangle_k \langle \sigma \rangle_{mem} \langle \pi \rangle_{ptr} \rightarrow \langle P \curvearrowright K \rangle_k \langle \sigma \circledast \sigma' \rangle_{mem} \langle \pi[P \leftarrow N] \rangle_{ptr} \\
\qquad \text{where } \textit{Dom}(\sigma') = \overline{P, P + N - 1} \\
\langle \texttt{free}(P); \curvearrowright K \rangle_k \; \langle \sigma \circledast \sigma' \rangle_{mem} \; \langle P \mapsto N, \pi \rangle_{ptr} \rightarrow \langle K \rangle_k \; \langle \sigma \rangle_{mem} \; \langle \pi \rangle_{ptr} \\
\qquad \text{where } \textit{Dom}(\sigma') = \overline{P, P + N - 1}
\end{array}$$

Underspecification of malloc

- Capturing the high degree of non-determinism in allocation

Outline

Formal definition of memory safety

Definition: KERNELC program p memory safe iff no execution of p gets stuck in a memory access operation using the KERNELC semantics

Theorem / Problem: Memory safety is undecidable even for programs which are closed and which terminate.
Proof Sketch:

- KERNELC Turing complete, can encode any decidable property;
- Encode $\varphi(n)$ as PGM_φ with input n and output out$\in\{0,1\}$.
- Next program is memory safe iff $(\forall n)\varphi(n)$ holds:

```
n = malloc(1);
PGM_φ
if (!out) *1 = 0;
```

Bad Coding Practices

It is generally considered bad practice to use result of allocation as:

- Source of non-deterministic input:

```
n = malloc(1); s = 0;
while(n){
 s = s + n;
 n = n - 1;
}
```

- A way to model non-deterministic choice:

```
x = malloc(1); y = malloc(1);
if (x<y) something else something else
```

Solution: Stronger notion of memory safety rejecting such programs

Strong memory safety: SAFE-KERNELC

- Return a new symbolic positive integer address for each allocation

$$\langle \texttt{malloc}(N) \curvearrowright K \rangle_k \langle \sigma \rangle_{mem} \langle \pi \rangle_{ptr} \rightarrow \langle P \curvearrowright K \rangle_k \langle \sigma \circledast \sigma' \rangle_{mem} \langle \pi[P \leftarrow N] \rangle_{ptr}$$

where P fresh symbol of sort Nat^+ and $Dom(\sigma') = \overline{P, P + N - 1}$

- Therefore, SAFE-KERNELC is deterministic
  - Modulo α-equivalence of symbolic integers
- Executable with minimal support for symbolic pointer arithmetic
  - Addition, subtraction, multiplication with constants

Properties of SAFE-KERNELC

- Yields a stronger definition of memory safety
  - Symbolic evaluation might get stuck for "bad programming practice" programs, e.g., if(malloc(10) $<$ 1000) $x = 1$; else $x = 2$;
- Symbolic execution comprises all KERNELC concrete executions
  - As valuations from symbolic to real pointers
- Strong memory safety guarantees memory safety!
  - If it doesn't get stuck, no KERNELC executions would either
- Strong memory safety can be monitored!
  - By running the program symbolically

Outline

Runtime verification of (strong) memory safety

Given a program to be checked for memory safety

- Simply execute it using SAFE-KERNELC
- Strong memory safety guaranteed while execution can proceed
- . . . which ensures memory safety for any possible execution

If the execution gets stuck:

- If because of memory addressing: program is unsafe
- If the oracle cannot solve constraints: maybe bad coding?

Decision procedure for memory safety

Assuming a class of programs $\mathcal{P}$ such that

- Each program $p \in \mathcal{P}$ terminates
  - Which implies it would also terminate under SAFE-KERNELC
- Any symbolic constraints of any program $p \in \mathcal{P}$ are decidable
  - That is, no non-standard use of pointers in p

Memory safety is decidable on $\mathcal{P}$

- SAFE-KERNELC execution is deterministic and terminates
  - Since each execution step is decidable
- If it does not get stuck, then program is memory safe
- If it gets stuck, then program is not memory safe
  - Since it cannot get stuck on constraint solving

Conclusion

First formal definition of memory safety

- Using formal definition of C-like language memory management
- Undecidable because of bad use of allocation non-determinism

Argued for a slightly stronger version of memory safety

- Based on symbolic, thus deterministic, treatment of pointers
- Runtime verifiable by monitoring its execution
- Equivalent with memory safety for “well written” programs
- Even decidable for terminating programs