CN: Verifying Systems C Code with Separation-Logic Refinement Types

Christopher Pulte Dhruv C. Makwana Thomas Sewell Kayvan Memarian Peter Sewell Neel Krishnaswami

University of Cambridge

3 June 2024

Why verify systems software?

Systems software: OSs, hypervisors, firmware.

Difficult to get right …

… but crucial for correctness and security of the whole system.

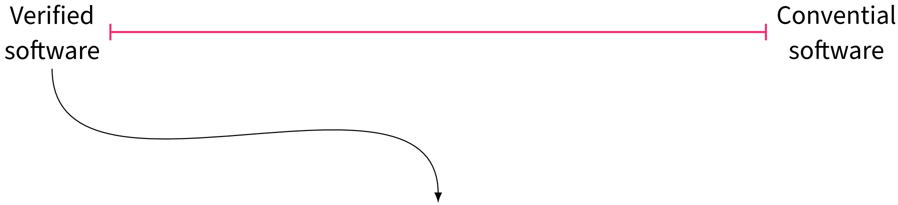
Systems software verification

Much progress in verification of low-level code,
verification successes like seL4 and CertiKOS

Systems software verification

Much progress in verification of low-level code,
verification successes like seL4 and CertiKOS

But:



- often designed for verification
- idealised semantics
- verification and maintenance very costly

CN

Goals:

- realistic language semantics
- target production systems code
- try to reduce cost of verification

pKVM

- Google hypervisor for Android,
- provides isolation between untrusted Linux OS and guest VMs accessing confidential data,
- written in C and ARM assembly,
- in Android since version 13.

pKVM verification

pKVM will be deployed on all Android phones
and it’s important for Android security.

verification project to prove pKVM provides isolation (Cambridge, MPI-SWS, Radboud, SNU, Aarhus)

pKVM verification

(overly?) ambitious plan:

1. verify pKVM C code with CN,

pKVM verification

(overly?) ambitious plan:

1. verify pKVM C code with CN,
2. *Google developers maintain the proof*, with limited involvement of verification experts

Challenges

Production systems software

pKVM is written by conventional software development team, not specifically to make verification easy, in C.

1. handle complex semantics of C
2. handle difficult low-level idioms (e.g. pointer arithmetic, aliasing data structures), complex invariants

Verification usability

Need large degree of proof automation, but fully automatic verifiers can behave unpredictably for undecidable problems (e.g. spinning forever).

3. predictable automation

C semantics

Type system design

Validation

Current work & open problems

C is complicated

- undefined behaviours
- implementation-defined behaviours
- unspecified values
- implicit type coercions
- mutable local variables that can be addressed
- complex control flow and variable scoping
- (WIP) subtle memory object model
- (todo) under-specified sequencing of memory accesses

Tedious and error-prone to handle directly.

Cerberus [Memarian et al.]

- well-validated C semantics
- elaborates C into functional(ish) language Core

$$\left[\!\!\left[\begin{array}{l}\texttt{signed int increment(signed int i) \{}\\ \quad\texttt{i = i + 1;}\\ \quad\texttt{return i;}\\ \texttt{\}}\end{array}\right]\!\!\right]$$

```
⟦ signed int increment(signed int i) {
    i = i + 1;
    return i;
  } ⟧
```

=

```
proc increment (i: integer): integer :=

  body =
    let i_l: pointer = create(4, 'signed int') in
    store('signed int', i_l, i);
    let v1: integer = load('signed int', i_l) in
    let sv: integer =
      let n: integer = conv_int('signed int', v1) + 1 in
      assert_undef(-2147483648 <= n /\ n <= 2147483647, <<UB036>>);
      n in
    store('signed int', i_l, conv_int('signed int', sv));
    let v2: integer = load('signed int', i_l) in
    kill('signed int', i_l);
    run ret1 (conv_int('signed int', v2))
    undef(<<UB088_reached_end_of_function>>))

  return label ret1
```

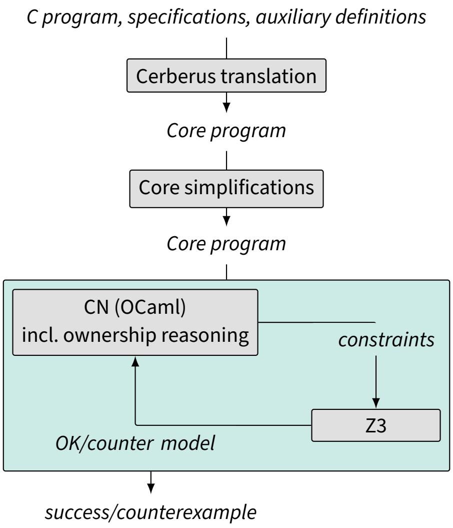
- explicit management of stack variables
- mathematical integers
- explicit UB checks

✗ omitted: weak sequencing, specified/unspecified

CN architecture

Instead of verifying the C code directly
we verify its Core semantics.

Compositional elaboration means CN instrumentation of C source can be mapped to Core instrumentation.



C semantics

Type system design

Validation

Current work & open problems

Overview

- handle difficult low-level idioms and invariants of real-world systems software
- provide predictable proof automation

simple first-order language of types

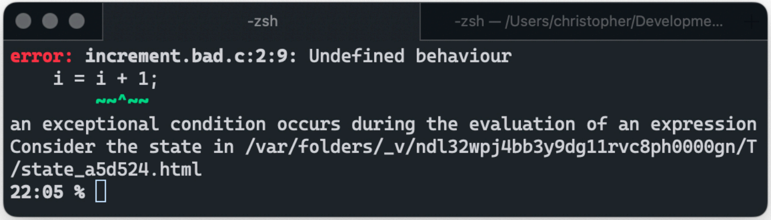
- refinement types by quantifier-free, decidable SMT solving
- linear resource types from separation logic
- … suitably restricted for predictable type checking

1. Liquid types

```
signed int increment(signed int i)
{
  i = i + 1;
  return i;
}
```

1. Liquid types

```
signed int increment(signed int i)
{
  i = i + 1;
  return i;
}
```



1. Liquid types

```
signed int increment(signed int i)
/*@ requires i < 2147483647i32;
    ensures return == i + 1i32; @*/
{
  i = i + 1;
  return i;
}
```

$$\Pi i : \mathrm{i}_{32}.$$
$$(i < 2147483647) \Rightarrow$$
$$\Sigma return : \mathrm{i}_{32}.$$
$$(return = i + 1) \wedge$$
$$\mathrm{I}$$

Standard liquid types [Rondon, Kawaguchi, Jhala]

- type-checking effectively by forward symbolic simulation,
- along each path, verify absence of UB and user-specified conditions …
- … by decidable, quantifier-free SMT solving

2. Resource types

CN has linear resource types from separation logic:

- built-in points-to type: for each C-type τ
  - Owned$\langle\tau\rangle$
  - Block$\langle\tau\rangle$ (for uninitialised memory + padding bytes)
- user-defined predicates

```
void zero(int *p)
/*@ requires take v = Owned<int>(p);
    ensures take w = Owned<int>(p); @*/
{
  *p = 0;
}
```

2. Resource types

Need more flexible types than Rust ownership types, e.g.: pointer aliasing, pointer arithmetic.

```
void zero_alias (int *p)
/*@ requires take v = Owned<int>(p);
    ensures take w = Owned<int>(p); @*/
{
  int *q = p;
  *q = 0;          // ok
  *p = 0;          // still ok
}
```

First class resource types: resources are types in their own right, de-coupled from pointer types, inspired by L^3 [Ahmed, Fluet, Morrisett].

Owned and `Block` are inferred, other resources require some manual effort.

3. Quantifiers

We need to abstract over “values” of resources, but we have to guarantee reliable inference.

$$\exists x. (p \mapsto x) * \dots$$
$$\exists q, x. (q \mapsto x) * \dots$$

3. Quantifiers

We need to abstract over “values” of resources, but we have to guarantee reliable inference.

$\exists x. (p \mapsto x) * \ldots$	ok: can be inferred
$\exists q, x. (q \mapsto x) * \ldots$	bad: imprecise, inference requires backtracking

3. Quantifiers

We need to abstract over "values" of resources, but we have to guarantee reliable inference.

$\exists x. (p \mapsto x) * \ldots$	ok: can be inferred
$\exists q, x. (q \mapsto x) * \ldots$	bad: imprecise, inference requires backtracking

Partition resource arguments into input and output arguments.

$$\textbf{input} \xrightarrow{\text{determines}} \text{output}$$

$$p \quad \mapsto \quad x$$

3. Quantifiers

Old: using resource inputs/outputs in inference.

New: To guarantee quantifier inference CN restricts expressible specifications quantify only over outputs.

separation logic	**CN**
$\exists x.\,(p \mapsto x) * \ldots$	CN: `take x = Owned<τ>(p); ...`
$\exists q, x.\,(q \mapsto x) * \ldots$	CN: cannot be expressed

Take-bindings enforce mode-correctness syntactically, via variable scoping.

3. Quantifiers – example

```
void increment_mem(int *p)
/*@ requires take v = Owned<int>(p);
             v < 2147483647i32;
    ensures  take w = Owned<int>(p);
             w == v + 1i32;          @*/
{
  *p = *p + 1;
}
```

3. Quantifiers – predicate definitions

```
struct node { int x; struct node *next; };
```

$$
\begin{aligned}
&\texttt{IntList}(p,xs) = \\
&\quad (\ p = \texttt{NULL} \wedge \\
&\quad\quad xs = \texttt{nil}\ ) \\
&\quad \vee \\
&\quad (\ \exists hd,q,tl. \\
&\quad\quad (p \mapsto (hd,q) * \\
&\quad\quad\ \texttt{IntList}(q,tl)) \wedge \\
&\quad\quad xs = \texttt{cons}(hd,tl) \\
&\quad)
\end{aligned}
$$

3. Quantifiers – predicate definitions

```
struct node { int x; struct node *next; };
```

$$
\begin{aligned}
&\mathtt{IntList}(p,xs) = \\
&\quad (\ p = \mathtt{NULL}\ \wedge \\
&\quad\quad xs = \mathtt{nil}\ ) \\
&\quad \vee \\
&\quad (\ \exists hd,q,tl. \\
&\quad\quad (p \mapsto (hd,q)\ * \\
&\quad\quad\ \mathtt{IntList}(q,tl))\ \wedge \\
&\quad\quad xs = \mathtt{cons}(hd,tl) \\
&\quad)
\end{aligned}
$$

```
predicate integer_list IntList (pointer p) {
  if (p == NULL) {
    return (Nil {});
  }
  else {
    take node = Owned<struct node>(p);
    take tl = IntList(node.next);
    return (Cons {hd: node.x, tl: tl});
  }
}
```

Predicate definitions look like function definitions:

- resource predicates claim ownership and *return* resource outputs
- if-then-else instead of disjunction to avoid backtracking

Grammar of types

$$bt \in \text{BaseType} ::= \mathsf{u}_w \mid \mathsf{i}_w \mid \text{pointer} \mid \text{struct } tag \mid \ldots \qquad w \in \mathbb{N}$$

$$\begin{aligned}
rt \in \text{ReturnTypes} ::= {} & \Sigma x : bt.rt \\
& \mid \text{take } x = P(e_1, \ldots, e_n) \ast rt \\
& \mid \text{take } x = (\circledast_{i.G} P(e \ast i + k, e_2, \ldots e_n)) \ast rt \\
& \mid lc \wedge rt \\
& \mid I
\end{aligned}$$

$$\begin{aligned}
ft \in \text{FunctionTypes} ::= {} & \Pi x : bt.ft \\
& \mid \text{take } x = P(e_1, \ldots, e_n) \multimap ft \\
& \mid \text{take } x = (\circledast_{i.G} P(e \ast i + k, e_2, \ldots e_n)) \multimap ft \\
& \mid lc \Rightarrow ft \\
& \mid rt
\end{aligned}$$

4. Constraint types

Limiting reasoning to decidable quantifier-free logic would be too restrictive.

CN lets users write recursive specification functions, all-quantified assertions.

- These are not handled by SMT automation, but require manual unfolding/instantiating
- … or lemmas, exported to ~~Coq~~ Rocq.

C semantics

Type system design

Validation

Current work & open problems

Formalisation: soundness of type checking

Kernel language: A-normalised Core with explicit resource terms, with bidirectional type system setup.

Theorem: soundness of type checking (not inference).

Case studies

pKVM buddy allocator: internally used by pKVM for managing page-table memory.

- (non-linear) pointer arithmetic,
- complex invariants,
- aliasing datastructure

part of pKVM pagetable code

- bit manipulation of pointers
- recursive pagetable ownership
- function pointers

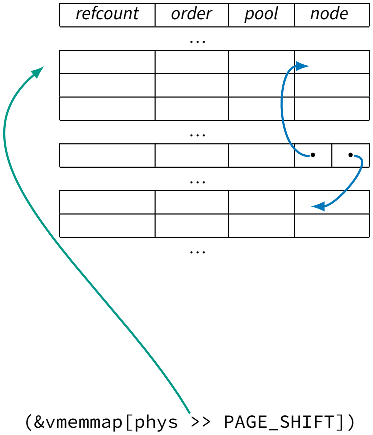
Buddy allocator

The vmemmap array holds meta-data about allocator pages.

```
struct list_head {
    struct list_head *next, *prev;
};

struct hyp_page {
    unsigned int refcount;      /* 0 when free */
    unsigned int order;         /* pages can be merged into pages of */
                                /* higher order */
    struct hyp_pool *pool;      /* (ignore) */
    struct list_head node;      /* prev, next linked list pointers to */
                                /* free pages of same order */
};
```

Buddy allocator verification challenges

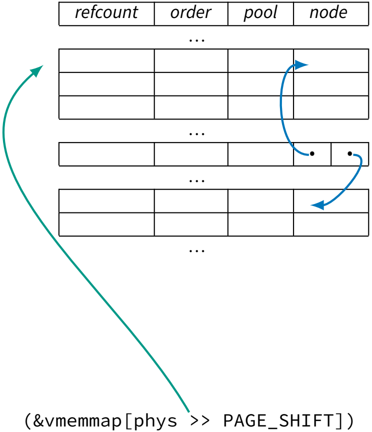


Two kinds of accesses to the vmemmap:

- linked-list prev/next pointers
- cell index computed from physical address

The safety of both kinds of accesses has to be justified.

Buddy allocator verification challenges



Two kinds of accesses to the vmemmap:

- linked-list prev/next pointers
- cell index computed from physical address

The safety of both kinds of accesses has to be justified.

```
take V = each (u64 i; start <= i && i < end)
              { Owned(array_shift(vmemmap,i)) };
each (u64 i; start <= i && i < end)
     { “V.value[i].node.prev and
        V.value[i].node.next are pointers into
        the vmemmap” }
```

Verification

Code verified mostly unchanged.

C source	**364 LoC**
generated Core	6169 LoC
CN specs	417 LoC
CN instrumentation	78 LoC
CN aux. definitions	249 LoC
CN lemma statements	165 LoC
Coq lemmas	1219 LoC
Verification overhead	**5.85x**

Type checking takes 155s

C semantics

Type system design

Validation

Current work & open problems

Current work & open problems

Original list of challenges:

1. complex semantics of C
2. difficult low-level idioms, invariants
3. predictable automation

Current work & open problems

Original list of challenges:

1. **complex semantics of C**
2. difficult low-level idioms, invariants
3. predictable automation

Handling C semantics using Cerberus works well.

Elaboration does not handle memory object model.

Dhruv Makwana extending CN with (variant of) VIP memory object model.

Current work & open problems

Original list of challenges:

1. complex semantics of C
2. **difficult low-level idioms, invariants**
3. predictable automation

Too early to tell.

- promising pKVM allocator and pagetable case studies
  … but those are the only two bigger examples
- CN will need some extensions:
  - converting between Owned and byte-representation,
  - concurrency, including simple relaxed concurrency
  - specification language (e.g. polymorphism)

Current work & open problems

Original list of challenges:

1. complex semantics of C
2. difficult low-level idioms, invariants
3. **predictable automation**

- New resource inference for predictable, reasonably automatic ownership reasoning: eager unfolding, lazy folding of predicates [inspired by Vazou et al. 2018*]
- Lots of automation for free from SMT solver
  … currently not always predictable

* Refinement Reflection: Complete Verification with SMT

Making verification actually usable

… needs not just predictable proof automation.

DARPA-funded project VERSE (Galois, UPenn, Cambridge, …) to make CN verification usable by non-experts (proof, testing, usability studies, etc.)

One main difficulty: diagnosing and repairing verification failure.

Diagnosing verification failure

```
struct node { int head; struct node* tail; };

struct node *reverse(struct node *xs)
{
  struct node *last = NULL;
  struct node *cur = xs;
  while(1)
  {
    if (cur == NULL) {
      return last;
    }
    struct node *tmp = cur->tail;
    cur->tail = last;
    last = cur;
    cur = tmp;
  }
}
```

Diagnosing verification failure

```
struct node *reverse(struct node *xs)
/*@ requires take L = IntList(xs);
    ensures  take L_ = IntList(return); L_ == rev(L); @*/
{
  struct node *last = NULL;
  struct node *cur = xs;
  while(1)
  /*@ inv take L1 = IntList(last); take L2 = IntList(cur);
          append(rev(L2), L1) == rev(L); @*/
  {
    if (cur == NULL) {
      return last;
    }
    struct node *tmp = cur->tail;
    cur->tail = last;
    last = cur;
    cur = tmp;
  }
}
```

Diagnosing verification failure

```
struct node *reverse(struct node *xs)
/*@ requires take L = IntList(xs);
    ensures  take L_ = IntList(return); L_ == rev(L); @*/
{
  struct node *last = NULL;
  struct node *cur = xs;
  while(1)
  /*@ inv take L1 = IntList(last); take L2 = IntList(cur);
          append(rev(L2), L1) == rev(L); @*/
  {
    if (cur == NULL) {
      return last;
    }
    struct node *tmp = c
    cur->tail = last;
    last = cur;
    cur = tmp;
  }
}
```

unproved constraint

append(rev(L), Nil {}) == rev(L)

counterexample

rev(L)	[1i32]
append(rev(L), Nil {})	[2i32]
…	…

-zsh

```
/Users/christopher/Desktop/rev/reverse_broken.c:95:3: error: Unprovable constraint
  while(1)
  ^~~~~~~~
Constraint from /Users/christopher/Desktop/rev/reverse_broken.c:97:11:
          append(rev(L2), L1) == rev(L); @*/
          ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Consider the state in /var/folders/_v/ndl32wpj4bb3y9dg11rvc8ph0000gn/T/state_0919f6.html
10:10 %
```

```
struct node *reverse(struct node *xs)
/*@ requires take L = IntList(xs);
    ensures  take L_ = IntList(return); L_ == rev(L); @*/
{
  struct node *last = NULL;
  struct node *cur = xs;
  /*@ apply append_nil(rev(L)); @*/
  while(1)
  /*@ inv take L1 = IntList(last); take L2 = IntList(cur);
          append(rev(L2), L1) == rev(L); @*/
  {
    if (cur == NULL) {
      return last;
    }
    struct node *tmp = cur->tail;
    cur->tail = last;
    last = cur;
    cur = tmp;
  }
}
/*@ lemma append_nil (datatype list l1)
      requires true;
      ensures append(l1, Nil {}) == l1;  @*/
```

```
struct node *reverse(struct node *xs)
/*@ requires take L = IntList(xs);
    ensures  take L_ = IntList(return); L_ == rev(L); @*/
{
  struct node *last = NULL;
  struct node *cur = xs;
  /*@ apply append_nil(rev(L)); @*/
  while(1)
  /*@ inv take L1 = IntList(last); take L2 = IntList(cur);
          append(rev(L2), L1) == rev(L); @*/
  {
    if (cur == NULL) {
      /*@ unfold rev(Nil {}); @*/
      /*@ unfold append(Nil {}, L1); @*/
      return last;
    }
    struct node *tmp = cur->tail;
    cur->tail = last;
    last = cur;
    cur = tmp;
    /*@ unfold rev(L2); @*/
    /*@ apply append_cons (rev (tl(L2)), hd(L2), L1); @*/
  }
}
```

Open questions for usable verification

Diagnosing and fixing verification failure is difficult.
Unclear whether lemma mechanism is good enough.

How can CN UI or type system help diagnose verification failure?

… help repair verification?

Can lemma application be reliably automated?

Are there better alternatives for CN/Rocq interaction?

… for partitioning the reasoning between decidable automation and manual proof?

PostDoc position

Plenty of open questions on making verification usable.

Future work

- verification usability and CN/Rocq interaction
- verify more pKVM code
- support (relaxed ARM systems) concurrency
- translation validation for ARM binary

CN + Cerberus (BSD 2-clause): `https://github.com/rems-project/cerberus`
CN tutorial: `https://github.com/rems-project/cn-tutorial`