seL4 Summit

Verifiable C and the Verified Software Toolchain

Lennart Beringer &
Andrew W Appel
Princeton University

deepspec

Verified Software Toolchain

November 14^{th} – 16^{th}, 2018

Styles of program verification

IDE-embedded verification tool

- annotation-enriched code
- verification carried out on intermediate form, using SAT/SMT
- assertions: expressions from the target programming language
- first-order quantification
- various verification/modeling styles, encoded e.g. as ghost state
- automated verification for correct annotations
- relationship to compiler's view of language unclear (soundness?)

VST: realization in interactive proof assistant (Coq)

- loop-invariants proof-embedded; function specs separate
- verification carried out on AST of source language
- assertions: mathematics (Gallina, dependent type theory)
- higher-order quantification
- specs can link to domain-specific theories (eg crypto, see below)
- interactive verification, enhanced by tactics + other automation
- formal soundness proof ("model") links to compiler (CompCert)

Styles of program verification

IDE-embedded verification tool

- annotation-enriched code
- verification carried out on intermediate form, using SAT/SMT
- assertions: expressions from the target programming language
- first-order quantification
- various verification/modeling styles, encoded e.g. as ghost state
- automated verification for correct annotations
- relationship to compiler's view of language unclear (soundness?)

VST: realization in interactive proof assistant (Coq)

- loop-invariants proof-embedded; function specs separate
- verification carried out on AST of source language
- assertions: mathematics (Gallina, dependent type theory)
- higher-order quantification
- specs can link to domain-specific theories (eg crypto, see below)
- interactive verification, enhanced by tactics + other automation
- formal soundness proof ("model") links to compiler (CompCert)

VST : goals and methodology

Functional-correctness verification technology for C that

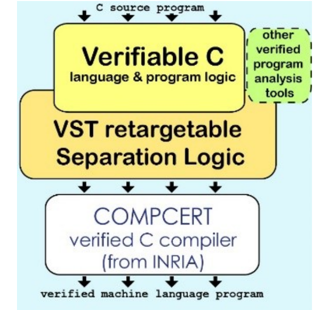
- applies to "real-world C"
  - support (almost) full C & virtually arbitrary programming styles
- permits expressive specifications and abstraction disciplines
  - e.g. custom-designed object protocols with opaque implementation invariants
  - interaction with external world (operating system, network, . . .)
  - top-to-bottom proof chains by integration with domain-specific model-level reasoning
- scales modularly to nontrivial code bases (see examples on later slides)
  - (concurrent) separation logic: 21st century variant of Hoare logic
  - semi-automated symbolic execution over abstract SL formulae inside Coq
- is foundationally justified w.r.t. the compiler's view of C
  - soundness proof in Coq w.r.t. CompCert's Clight language

(Current) limitations, TCB:

- main focus: partial-correctness, incl. safety (but no liveness)
- no intensional properties (time consumption, cache behavior…)
- no goto, no Duff's device, no embedded assembly (yet)
- TCB: Coq (incl Ocaml & below)
  CompCert x86/ARM/Power/RiscV but not Clight!

Verified Software Toolchain

Main features



Floyd: forward-symbolic analysis, partial solution of side conditions using Ltac or verified decision procedures.

Concurrency (Dijkstra-Hoare + fine-grained), impredicative quantification, ...

Partial correctness + safety + limited information flow.

Expressive, modular, foundational, semi-automatic program logic for C.

Higher-order separation logic

Clight, as formalized in CompCert

Soundness proof for step-indexed model formalized w.r.t. operational semantics.

CompCert: compilation to x86-32/64, ARM, PowerPC, RiscV preserves externally visible behavior

User supplied

Typical workflow

1. Write a C program

```
#include <stddef.h>

struct list {int head; struct list *tail;};

struct list *append (struct list *x, struct list *y) {
  struct list *t, *u;
  if (x==NULL)
    return y;
  else {
    t = x;
    u = t->tail;
    while (u!=NULL) {
      t = u;
      u = t->tail;
    }
    t->tail = y;
    return x;
  }
}
```

append.c

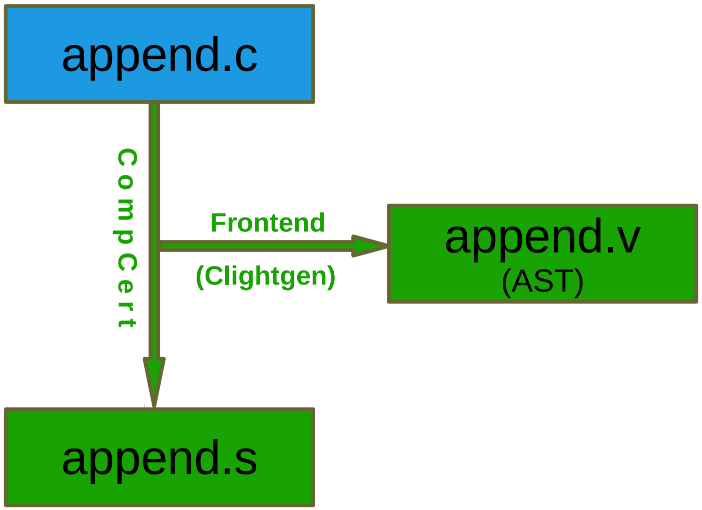
Dynamically generated | User supplied

Typical workflow

1. Write a C program

```
#include <stddef.h>

struct list {int head; struct list *tail;};

struct list *append (struct list *x, struct list *y) {
  struct list *t, *u;
  if (x==NULL)
    return y;
  else {
    t = x;
    u = t->tail;
    while (u!=NULL) {
      t = u;
      u = t->tail;
    }
    t->tail = y;
    return x;
  }
}
```

2. Parse and compile using Clightgen/Compcert



Dynamically generated | User supplied

Typical workflow

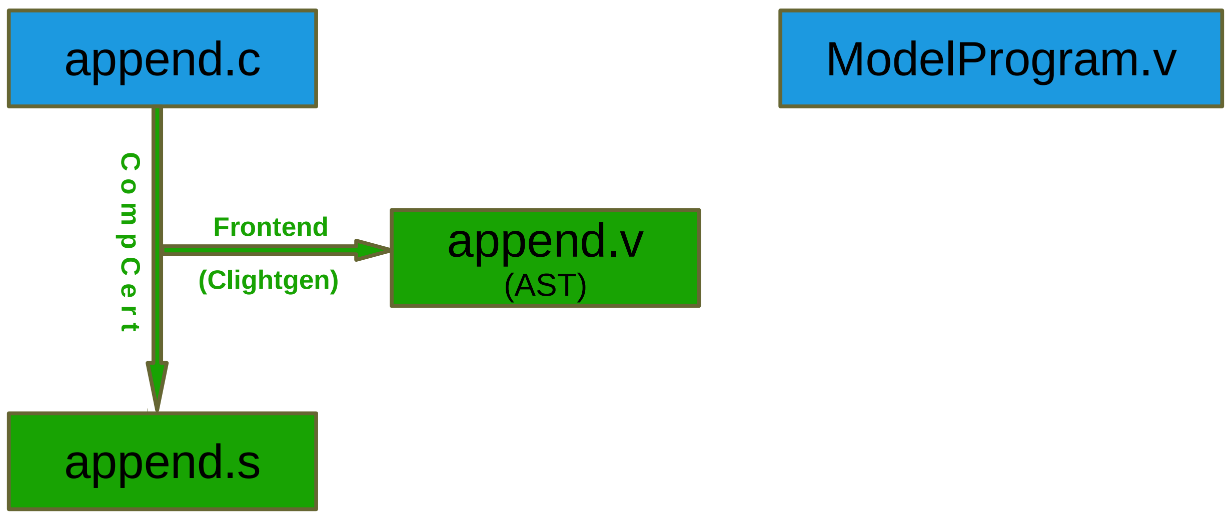
1. Write a C program

```
#include <stddef.h>

struct list {int head; struct list *tail;};

struct list *append (struct list *x, struct list *y) {
  struct list *t, *u;
  if (x==NULL)
    return y;
  else {
    t = x;
    u = t->tail;
    while (u!=NULL) {
      t = u;
      u = t->tail;
    }
    t->tail = y;
    return x;
  }
}
```

2. Parse and compile using Clightgen/Compcert

3. Write a model program in Gallina

```
Fixpoint app (al bl: list Z) : list Z :=
 match al with
 | nil => bl
 | a::al' => a :: app al' bl
 end.
```



Dynamically generated | User supplied

Typical workflow

1. Write a C program

```c
#include <stddef.h>

struct list {int head; struct list *tail;};

struct list *append (struct list *x, struct list *y) {
  struct list *t, *u;
  if (x==NULL)
    return y;
  else {
    t = x;
    u = t->tail;
    while (u!=NULL) {
      t = u;
      u = t->tail;
    }
    t->tail = y;
    return x;
  }
}
```

2. Parse and compile using Clightgen/Compcert

3. Write a model program in Gallina

```
Fixpoint app (al bl: list Z) : list Z :=
 match al with
 | nil => bl
 | a::al' => a :: app al' bl
 end.
```

4. Write a VST specification

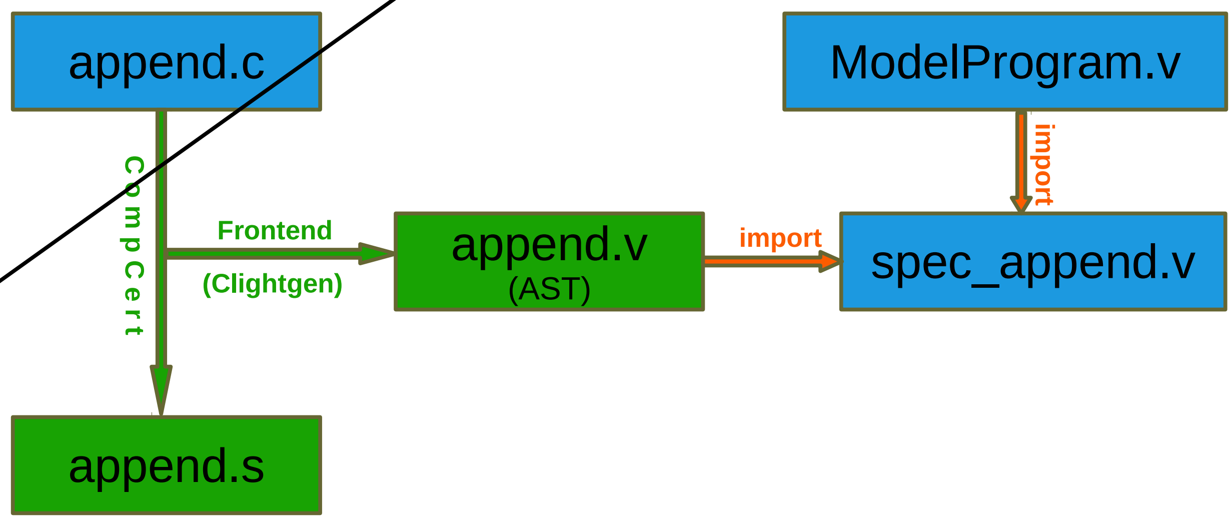
```
Definition append_spec :=
 DECLARE _append
  WITH sh : share, x: val, y: val, s1: list val, s2: list val
  PRE [ _x OF (tptr t_struct_list) , _y OF (tptr t_struct_list)]
     PROP(writable_share sh)
     LOCAL (temp _x x; temp _y y)
     SEP (lseg LS sh s1 x nullval; lseg LS sh s2 y nullval)
  POST [ tptr t_struct_list ]
    EX r: val,
     PROP()
     LOCAL(temp ret_temp r)
     SEP (lseg LS sh (s1++s2) r nullval).
```

Aux. Variables (arb. Coq type)

Precondition

User-defined repr. predicate

Postcondition



Typical workflow

Statically provided | Dynamically generated | User supplied

1. Write a C program

```
#include <stddef.h>

struct list {int head; struct list *tail;};

struct list *append (struct list *x, struct list *y) {
  struct list *t, *u;
  if (x==NULL)
    return y;
  else {
    t = x;
    u = t->tail;
    while (u!=NULL) {
      t = u;
      u = t->tail;
    }
    t->tail = y;
    return x;
  }
}
```

2. Parse and compile using Clightgen/Compcert

3. Write a model program in Gallina

```
Fixpoint app (al bl: list Z) : list Z :=
 match al with
 | nil => bl
 | a::al' => a :: app al' bl
 end.
```

4. Write a VST specification

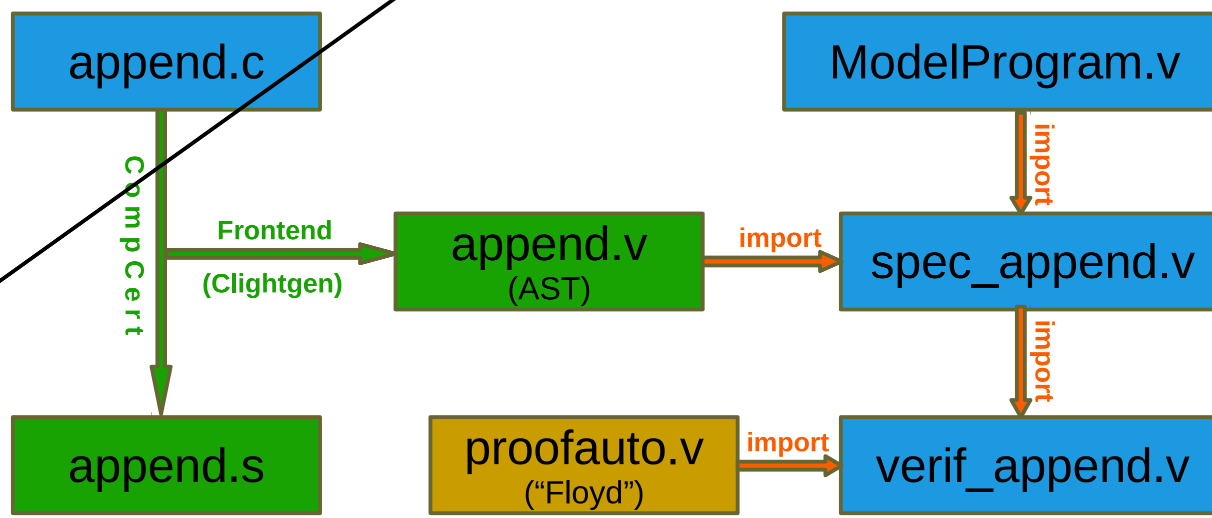
Aux. Variables (arb. Coq type)

```
Definition append_spec :=
 DECLARE _append
  WITH sh : share, x: val, y: val, s1: list val, s2: list val
  PRE [ _x OF (tptr t_struct_list) , _y OF (tptr t_struct_list)]
     PROP(writable_share sh)
     LOCAL (temp _x x; temp _y y)
     SEP (lseg LS sh s1 x nullval; lseg LS sh s2 y nullval)
  POST [ tptr t_struct_list ]
    EX r: val,
     PROP()
     LOCAL(temp ret_temp r)
     SEP (lseg LS sh (s1++s2) r nullval).
```

Precondition

User-defined repr. predicate

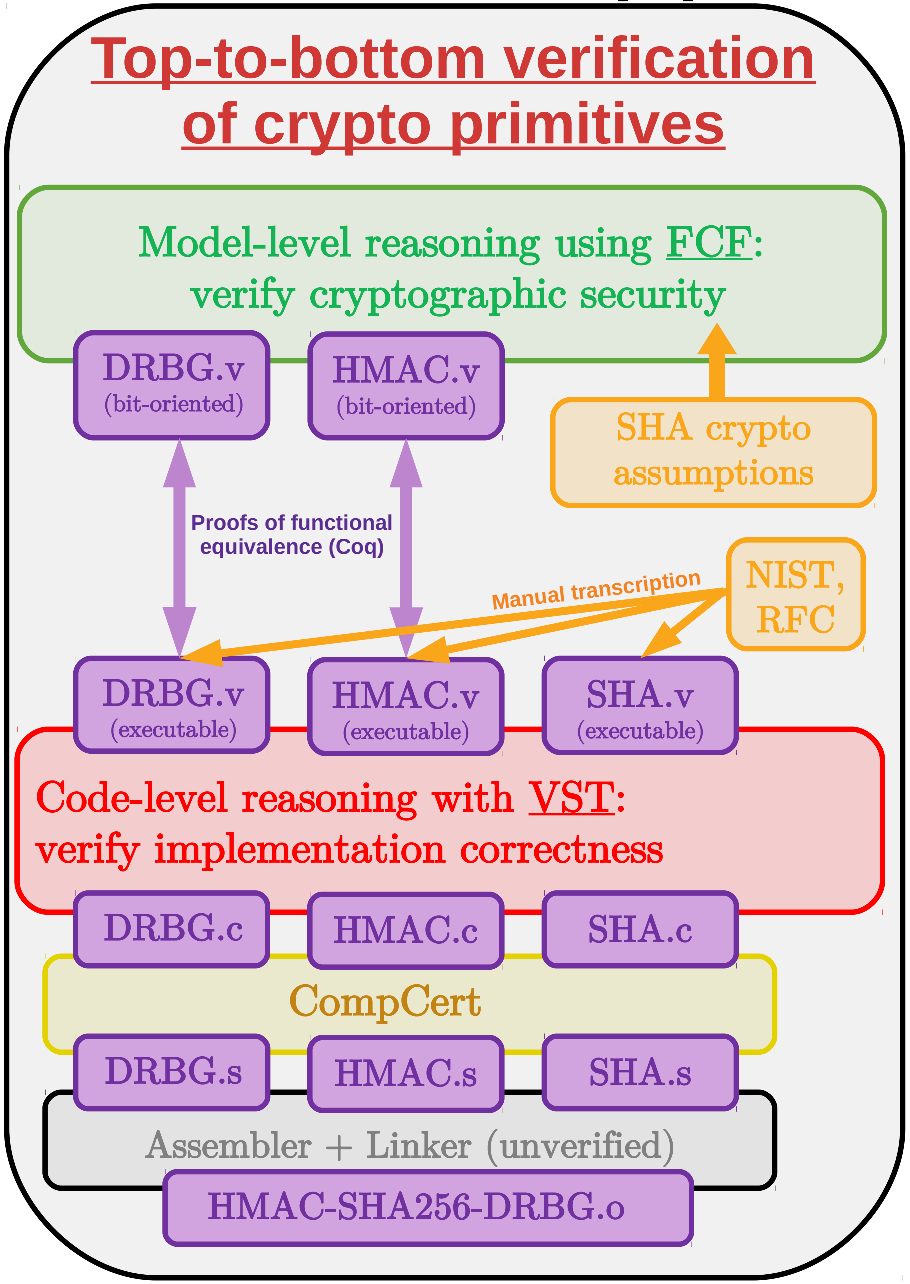
Postcondition



5. Prove the function body (define loop invariants on demand)

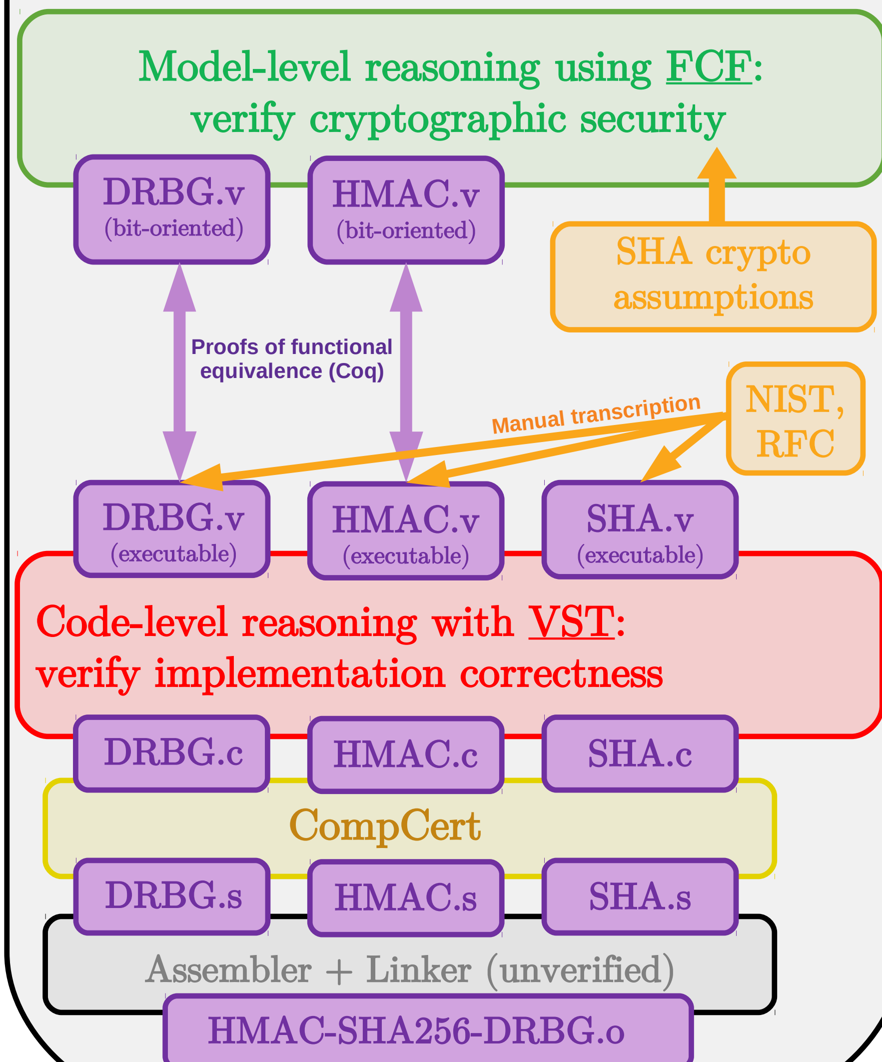
```
Lemma body_append: semax_body Vprog Gprog f_append append_spec.
Proof. start_function. ... ( proof script ) ... . Qed.
```

HACMS applications (also see A. Nogin's talk)



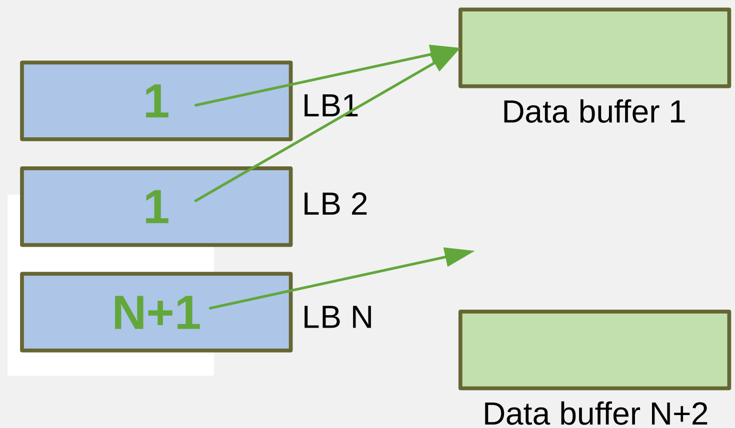
HACMS applications (also see A. Nogin's talk)

Top-to-bottom verification of crypto primitives



Nonblocking concurrency

N readers, 1 writer



1) W selects free data buffer 0 < **b** < N+3 and writes data to **b**
2) W communicates **b** to all N readers using atomic exchanges to all LB's
3) Reader **i** inspects LB**i** to find location of next data item
4) Reader **i** acknowledges receipt of **b** using atomic exchange "Empty" in Lb**i**
5) Accesses to data buffers use ordinary load/store operations

N+2: W can always find a free data buffer !

Further case studies

Abstract data types: binary search trees (implemented by hash table)
- magic-wand-as-frame proof technique for descending into data structures

Runtime components:
malloc/free library (D. Naumann)
garbage collector (S. Wang)

External interactions: DeepSpec server
- reasoning about state of external world and operating system
(socket API specs reusable in seL4 context?)

Custom object systems:
OpenSSL hash contexts (“envelopes”)
- how to specify function pointers and general “apply” functions in C; whitebox & blackbox abstraction

External uptake & next steps

Benoit Viguier (Nijmwegen): elliptic-curve cryptography
Russel O'Connor (Blockstream): interpreter for smart-contract language

integrate functional and imperative programming in Coq!

With HRL (A. Nogin, M. Warren) and Purdue (B. Delaware): provably correct & safe data format (de)serializers

With W. Mansky (UI Chicago): search data structures with optimistic concurrency control

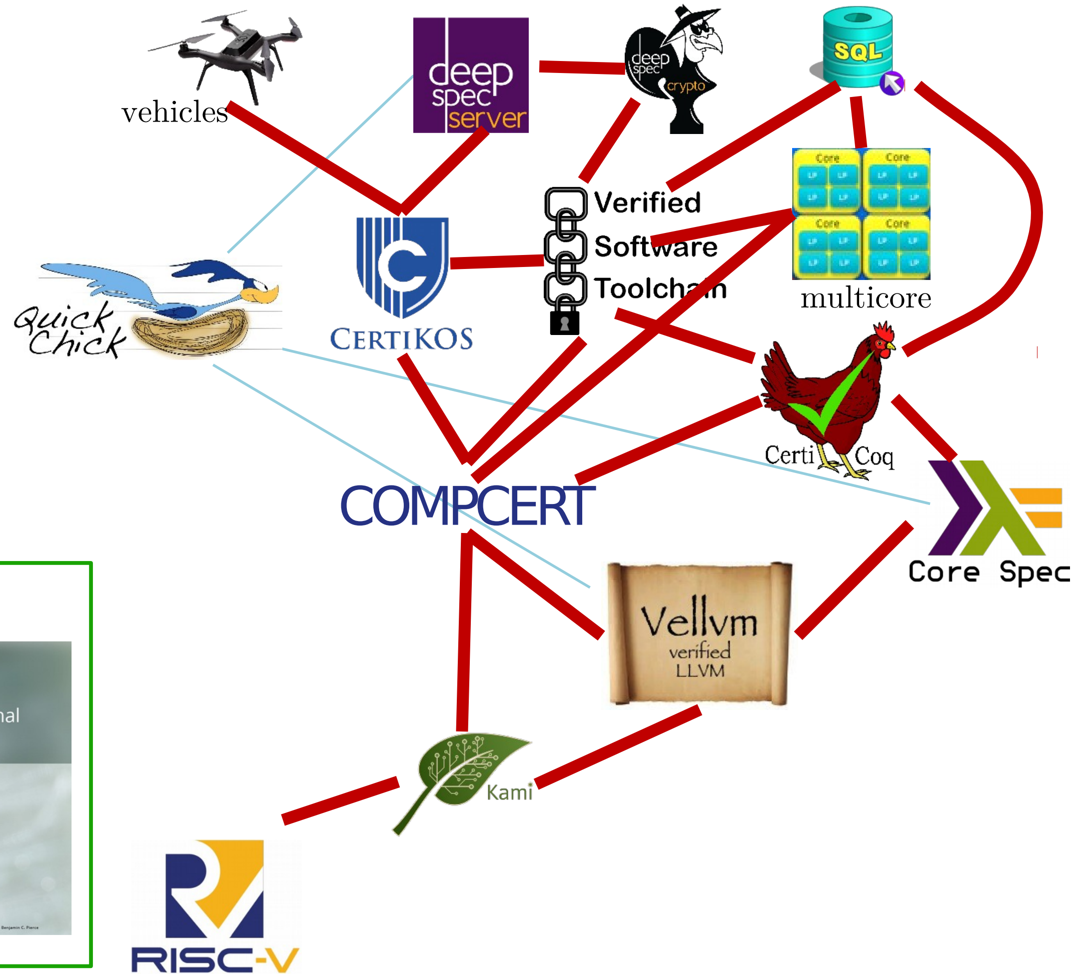
Try it yourself: http://vst.cs.princeton.edu/download

VST in context: deep spec (2016 – 2020), https://deepspec.org

RICH	describe complex behaviors in detail
FORMAL	in notation with a clear semantics
2-SIDED	connected to clients & implementations
LIVE	machine-checked connection to implementations

Community building:
- summer schools ‘17 & ‘18
- workshops at PLDI etc.

Curriculum development:



Coq/Isabelle: the IDEs for 21^{st}-century system stacks