Foundational Program verification using VST

Lennart Beringer, William Mansky, Andrew Appel

Princeton University

deep spec

Verified Software Toolchain

IBM Programming Languages Day 2017

T.J. Watson Research Center
Monday, December 4th 2017
https://ibm.biz/plday2017

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
- multitude of verification/modeling styles, encoded e.g. as ghost state
- automated verification for correct annotations
- relationship to compiler’s view of language unclear (soundness?)

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
- multitude of 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)

Formal **Program** Correctness **Verification**

"Prove"?

"Correct"?

Prove that your
C program is *correct*.

Prove in Coq that your
C program *satisfies* its
functional specification.

Q: How to express a functional spec?
A: Write a **functional** program!

Corollaries:
- safety, incl. memory safety: no buffer overruns etc.
- information-flow guarantees captured in functional model

Provides the expected functionality

Prove your **C** program
functionally correct ***in Coq***.

Not covered: intensional properties
- execution time, power consumption, cache behavior
- information flow via these side channels

Further refinements:

- **"prove": in Coq, semi-automatically**
- **"program": fragment (modularity)**
- **"satisfies": program logic with interpretation & soundness proof w.r.t. operational semantics**
- **"program": proof for C, guarantee for ASM via compiler correctness (CompCert)**
- **assumptions: Coq kernel, ASM model,…**

Gallina

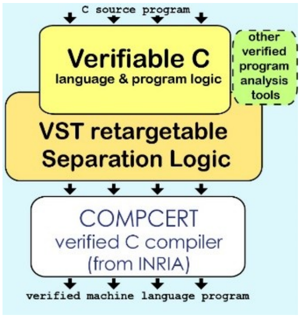
The pure functional language inside Coq's logic has a nice clean proof theory. **This enables us to write specs that are easy to reason about, for students, practitioners,....**

Gallina is **executable** inside Coq, so specifications can be **tested**.

Many kinds of applications are best **programmed** in a safe, garbage-collected functional programming language. Gallina is **extractable** to OCaml so can be integrated into existing software infrastructures.

Verified Software Toolchain

Verified
Software
Toolchain



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

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

Clight, as formalized in CompCert

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

Separation logic

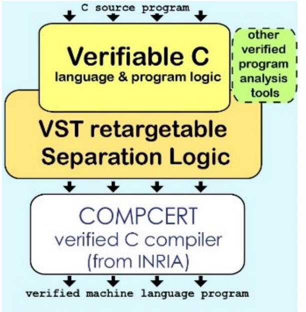
Soundness proof for step-indexed model formalized w.r.t. operational semantics in Coq.

Partial correctness + safety + limited information flow.

X86-32/64, Arm, PowerPC, RiscV, RTL, …

Verified Software Toolchain

Verified
Software
Toolchain



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

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

Clight, as formalized in CompCert

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

Separation logic

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

Partial correctness + safety + limited information flow.

X86-32/64, Arm, PowerPC, RiscV, RTL, …

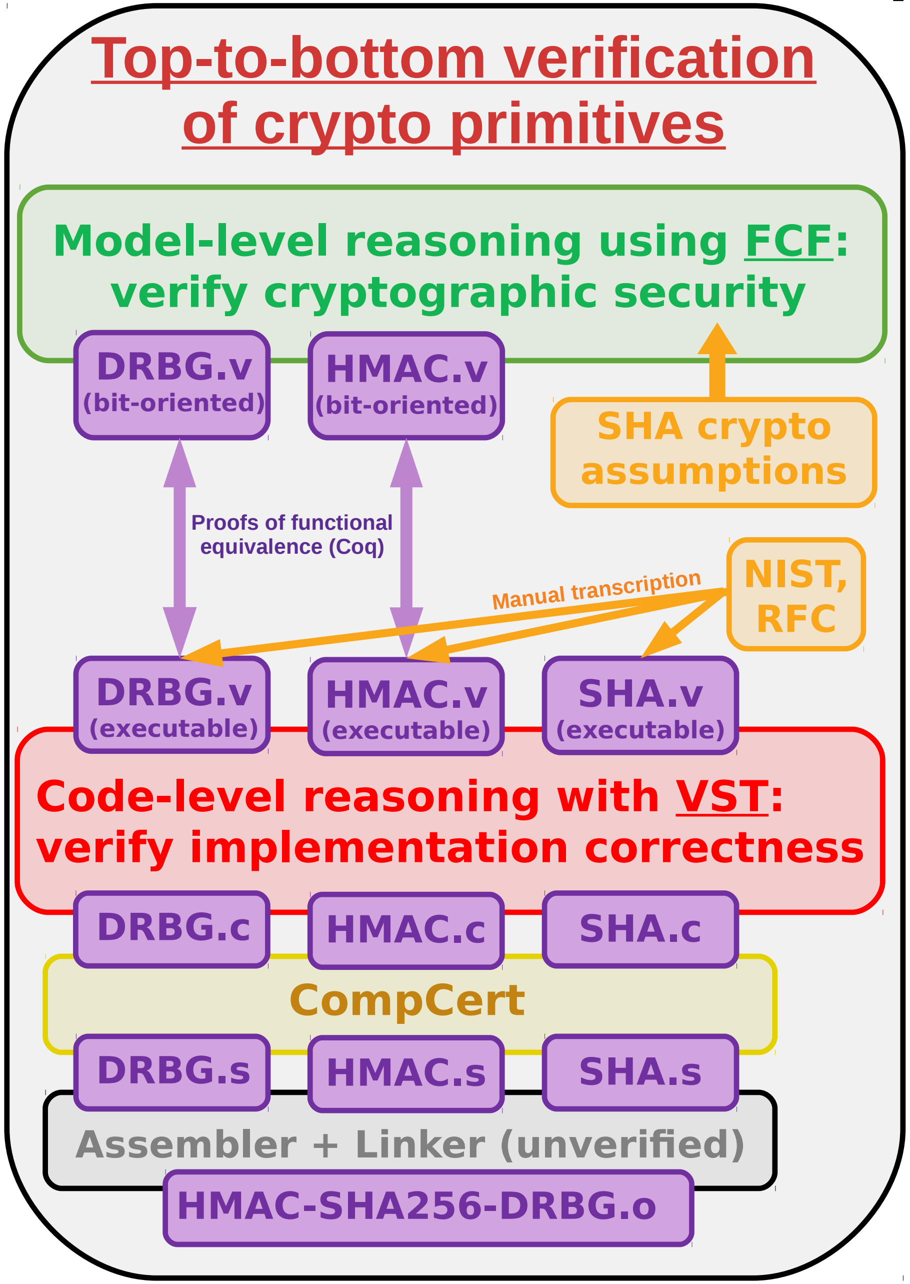
Typical use: exploit convenience of Gallina:

1. write a (functional) **model program** **p** in Gallina
2. structure of **p**: one function **f** for each C function **c**
3. Function spec for **c** refers to specification function **f**

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

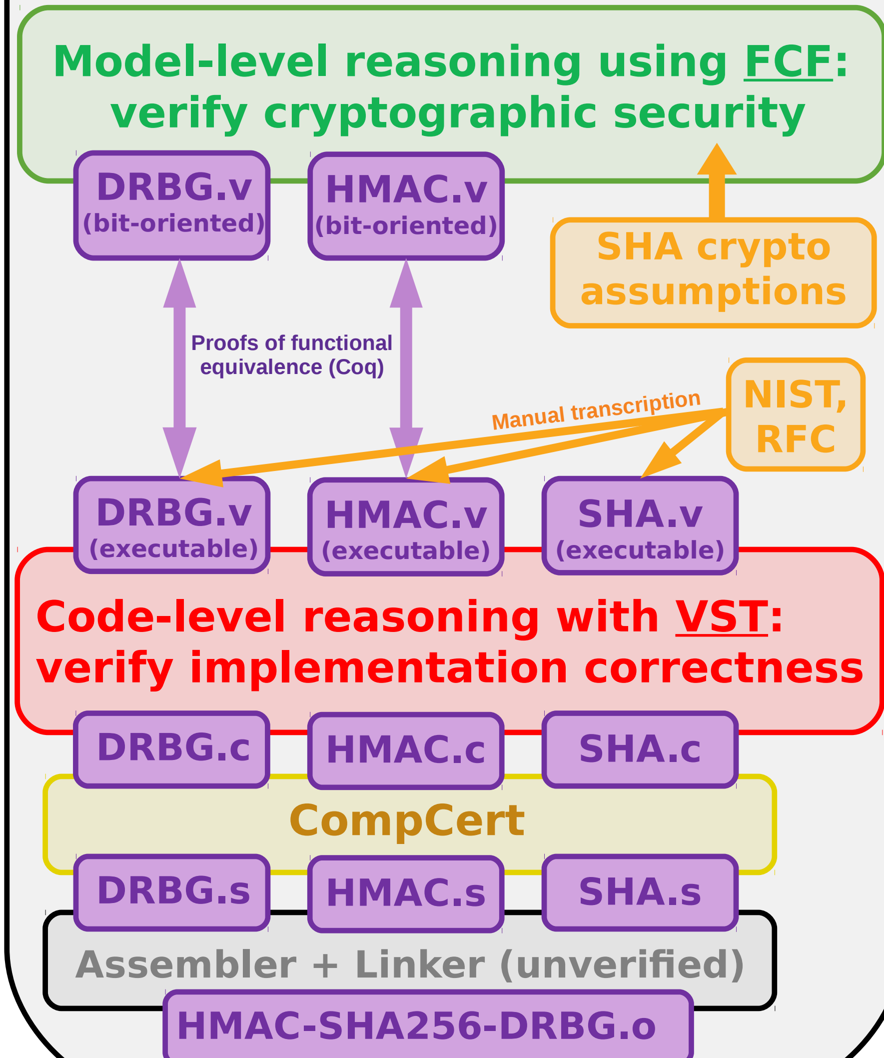
{ listseg α x **null** * listseg β y **null** } append(x,y) { listseg (app α β) retval **null** }

Recent applications



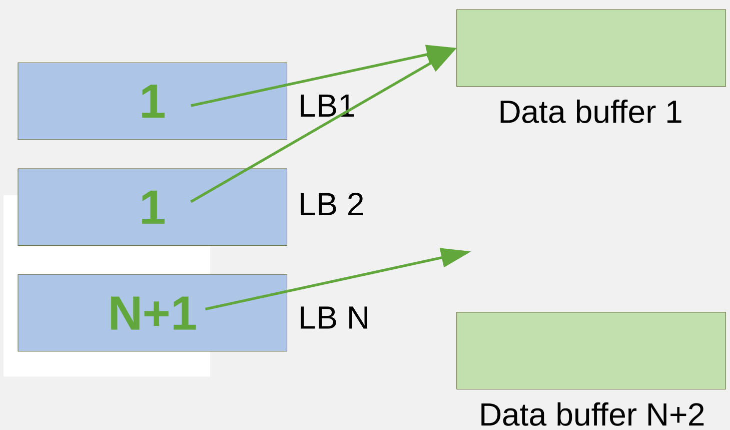
Recent applications

Top-to-bottom verification of crypto primitives



Nonblocking concurrency

N readers, 1 writer

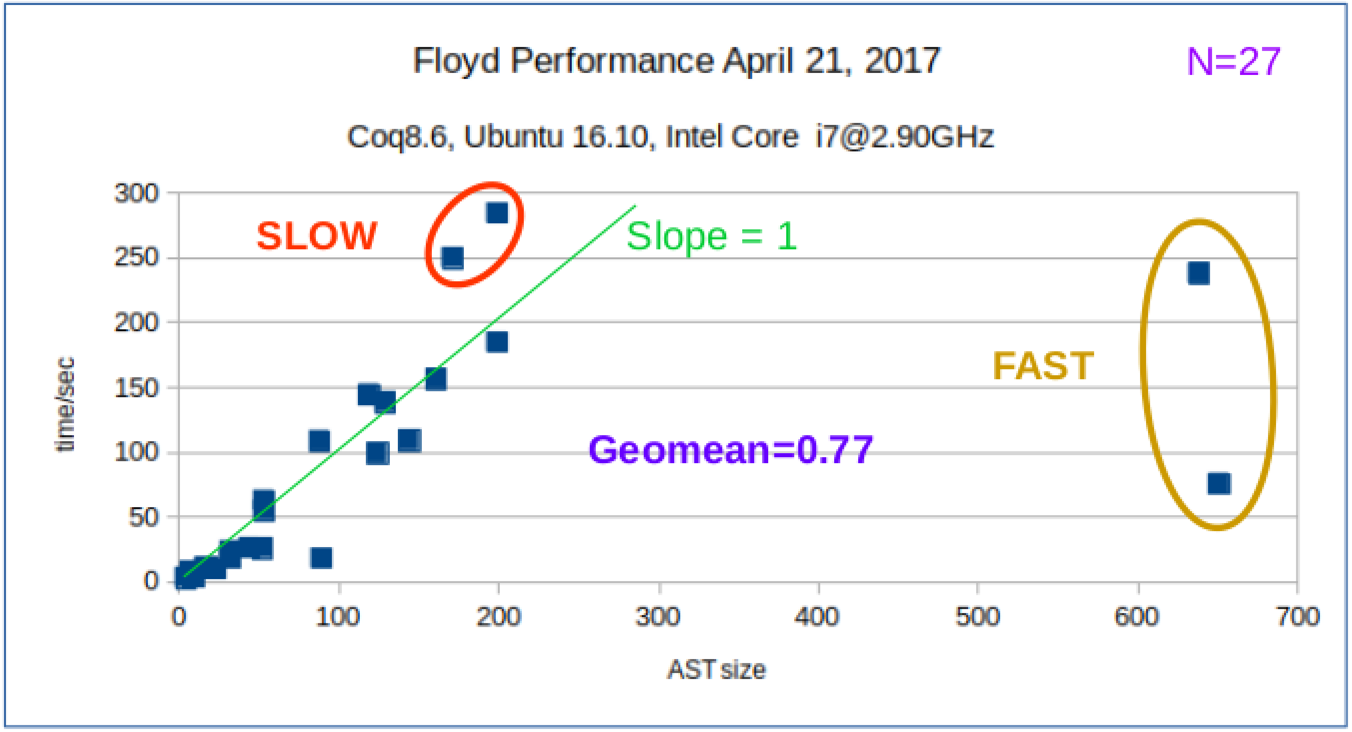


1) W selects free data buffer 0 < **b** < N+3 and writes data to **b**
2) W communicates **b** to all N readers using atomic exchanges to all LB’s
3) Reader **i** inspects LB**i** to find location of next data item
4) Reader **i** acknowledges receipt of **b** using atomic exchange “Empty” in Lb**i**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

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

Automation & Performance

- assertions in **canonical form**: PROP (P) LOCAL (Q) SEP (R)
- SL proof rules for C complex! Many entailments!
- full employment theorem for tactics programmers
- horizontal frame, not vertical: **PROP (P) LOCAL (Q) SEP (R) FR (F)**



Current & Future Work

Concurrency:

- Semantic justification of concurrent ghost state a la Iris/GPS
- Derivation of proof rules for C11 atomics
- Application to nonblocking algorithms and data structures

DeepSpec (NSF):

- **crypto** primitives and protocols: integration with FiatCrypto's ECC, TLS 1.3, ...
- specification of **CertiKOS** system call API
- specification and verification of web server
- Interaction with **Vellum**, **CoreHaskell**, and **CertiCoq**

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