Specification-based Software Engineering

Formal Methods @ Scale, September 2019

Thomas Ball and Jonathan Protzenko
Microsoft Research

Perspective from 20 years at MSR

- Solid progress in the use of specifications, automated solvers/analyzers and interactive proof assistants
- The complexity of both specifications and the systems we can reason about has increased substantially
- More to be done to achieve *specification-based software engineering*

Specification-based Software Engineering

- Deliver verified and efficient components into existing systems
  - CompCert (C compiler)
  - EverCrypt (Cryptographic algorithms)
  - EverParse (binary parsers)
  - …

- What's different?
  - Specification is a major and important artifact
  - Specification-based languages for proof and performance
  - Automation and interaction: predictability, stability, transparency of tools

Formal Specification: When?

- Critical components
- Widely used components
- Standardized interfaces (standardization)
- Multiple competing implementations
- New domains (e.g., smart contracts, network verification)

Specification-based programming languages

If we want verified code at scale, we need languages that are designed to enable productive verification *and* produce efficient code:

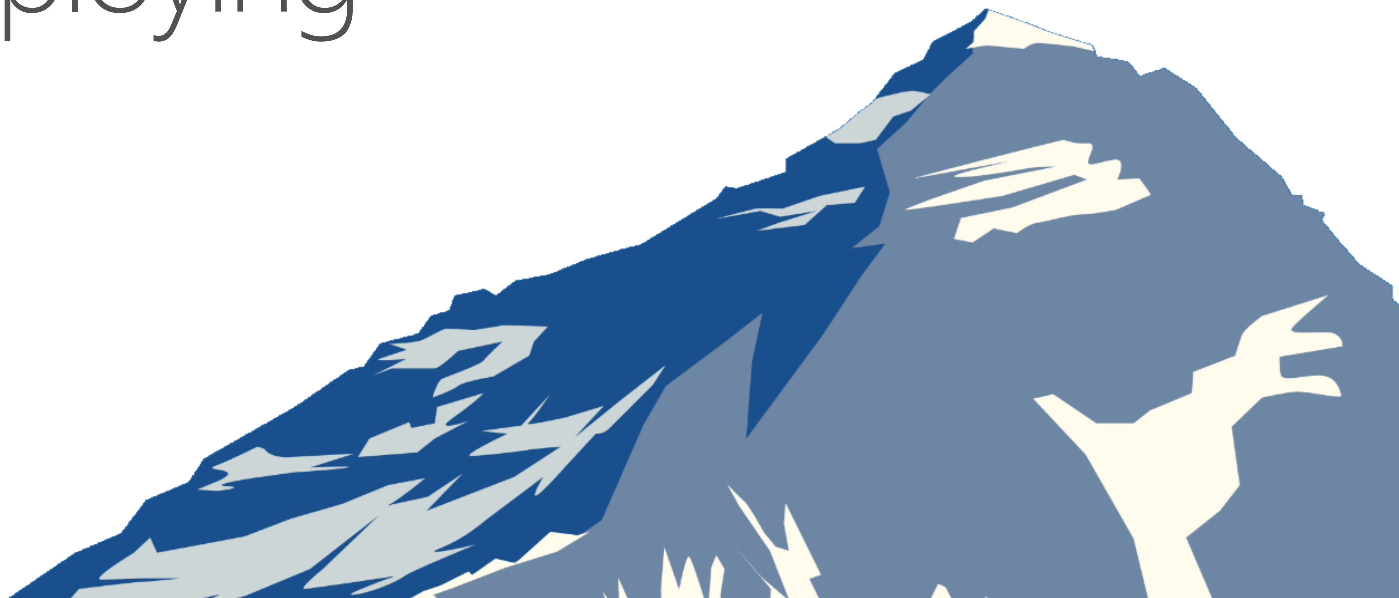
- **F*** (https://www.fstar-lang.org/) powers up the F# (mostly) functional language with a dependent type system deeply integrated with SMT
- **Ivy** (https://github.com/Microsoft/ivy) guides the designer to structure their systems so that verification tasks are reduced to decidable fragments of first-order logic
- **Lean** (https://github.com/leanprover/) is a pure functional language and proof assistant, with the goal of high automation

Long-term investment required

Microsoft
Microsoft Research - Inria JOINT CENTRE
Inria INVENTEURS DU MONDE NUMÉRIQUE
Carnegie Mellon University
THE UNIVERSITY OF EDINBURGH

Everest

Building and Deploying Verified Security Components

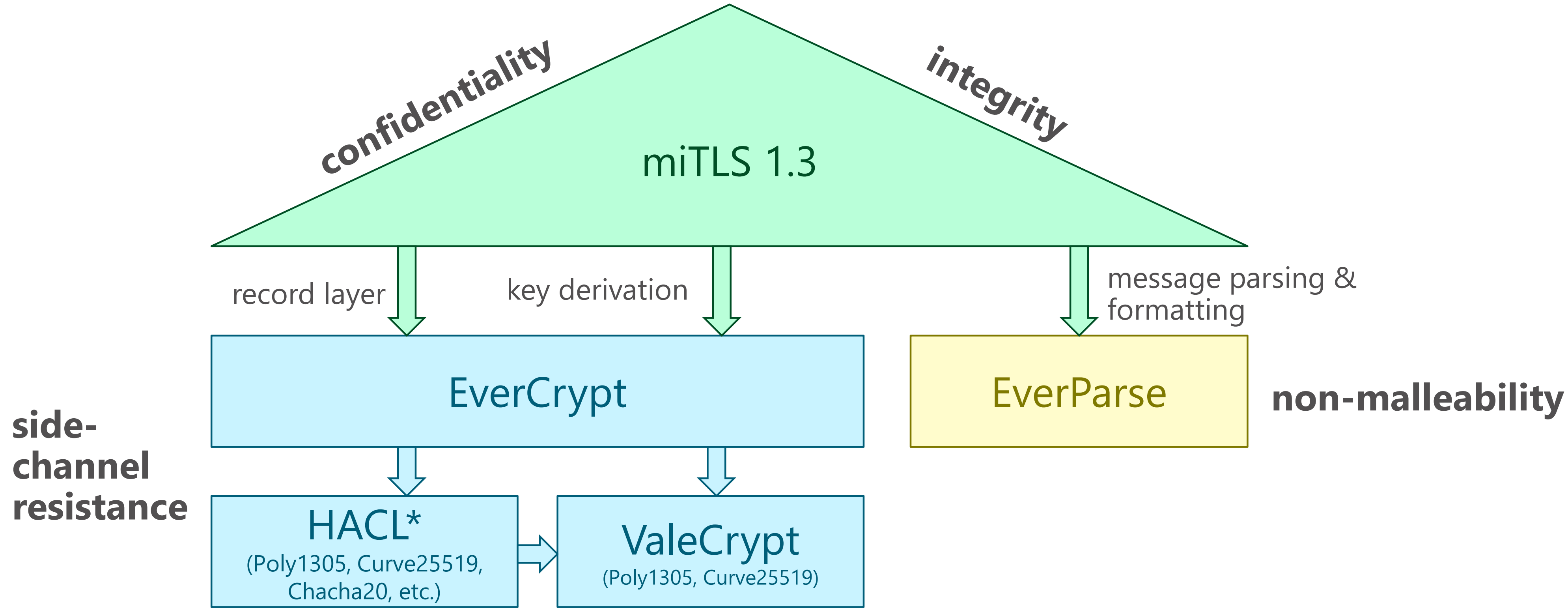


Everest in one slide

- Verifying the TLS 1.3 stack
  - high liability; long flawed history; formal-methods friendly
- Using F* and companion DSLs
- 500,000 lines of verified source; 200,000 lines of compiled C & ASM
- Work in progress; releasing independent components as we go
  - EverCrypt = HACL* + ValeCrypt
  - EverParse
- Everest code used in: Firefox, Windows, Azure, Tezos, Wireguard, etc.

4 years into the project; now 25+ participants, 6 locations, 5 timezones

Overview of Everest: what do we prove?



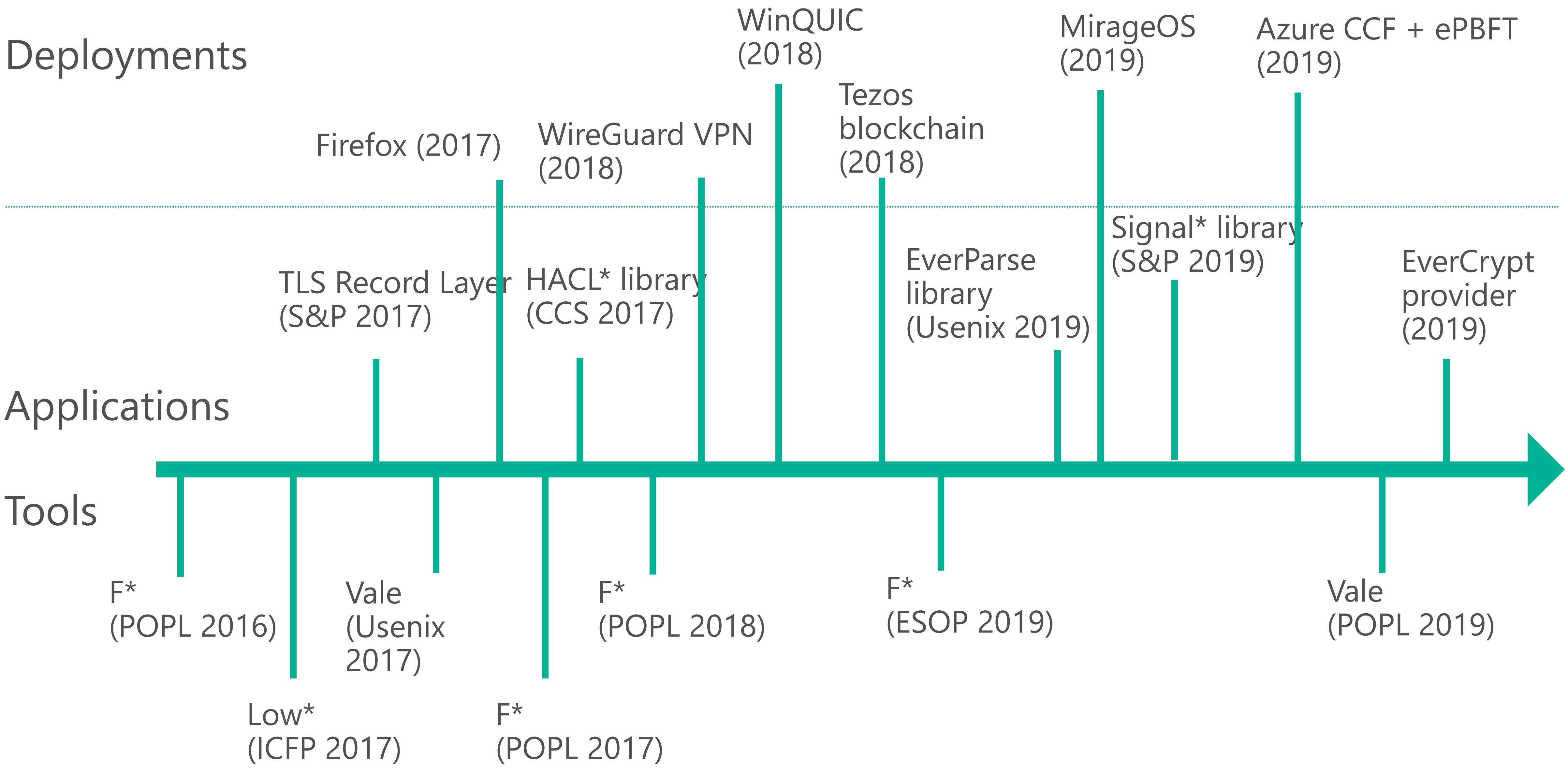
+ memory safety, functional correctness

A journey in verification at scale

- HACL* (2017): standalone, separate algorithms
  - separate verification scopes
  - monolithic verification invocations
  - no notion of abstraction
  - 23kloc
- EverCrypt (2019): a provider that unifies all the crypto
  - all in one scope
  - modular, parallel verification
  - abstraction boundaries
  - spec equivalence; agility, multiplexing, CPU auto-detection
  - 115kloc (5x increase)

What made it possible?

A journey in time



Constant improvements in core research

- The **core** F* technology (N. Swamy)
  - Monotonicity (D. Ahman)
  - Revamped core Low* libraries (T. Ramananandro, A. Rastogi)
  - Tactics (G. Martinez)
  - Support for Vale v2 meta-programmed WPs (C. Hawblitzel)
- Driven by applications
  - Close feedback loop between applications and tool authors
  - Only possible with dedicated, full-time language / PL experts
- Challenges
  - Priorities
  - Communication with time difference / separate institutions & agendas
  - Temptation of the next iteration

Constant improvements in tools

- F*: from research project to actual language
  - interactive Emacs mode (C. Pit-Claudel)
  - parallel builds with binary artifacts
  - style guides and formatters
- Everest support
  - 24/7 continuous integration (all 500k lines of code)
  - build reliability, reproducibility
- Challenges
  - Temptation of corporate-only solutions
  - Hard to field positions for build & CI
  - Open-source is non-negotiable

Constant interaction with industry

- Pick your battles
  - Early champions were essential to our success (Mozilla)
  - Open-source easier to work with
- Be ready to listen
  - Integration blockers are not what you may think
  - Performance matters: be ready to learn a lot
- Challenges
  - Endless integration / engineering work
  - Coordination, internally & externally

The art of writing verified software?

- Specifications are *deep*; how to remain modular?
  - A single architect that designs and sketches (does not scale)
  - Abstract code & specifications (verbose, but better)
  - Is verified software inherently monolithic?
- SMT does not scale
  - The beginner's lament: small examples = ok, large projects = sadness
  - A discipline of tight abstraction boundaries and abstract reasoning
  - A carefully-learned set of good practices to scale: a healthy dose of skepticism towards SMT along with tactics

The art of writing verified software?

- How to coordinate 25+ people?
  - The price of diverging libraries, abstractions and specifications is **enormous**
  - Social: a distributed, loosely-tied group
  - Technical: fear of changes, non-modularity, SMT-fragility
- "Write once, fixup forever"
  - The elusive art of a robust proof
  - Requires deep expertise and understanding
  - Alleviated by tools, CI, etc. (could be better)
  - Always go back to the drawing board for core improvements

Looking forward

- Everest takes place in the long time scale!
  - The culmination of a decade of research on F* and TLS
  - Building expertise takes time; PhD students come and go
  - Need stability; the ability to do fundamental work on languages and theory
  - A testament to basic research vs. squeezing immediate results
- Things we continue to grapple with
  - Tool improvements are the most visible and effective; yet hard to retain or even have anyone dedicated
  - Scaling up our own workforce: need for verification engineers; need to reward training materials and documentation

Points for Further Discussion

- Automation
  - More work to be done (scaling safety checking still difficult)
  - Predictability, Stability, Transparency
- Composition
  - how to integrate independently verified components and verify that the integration is correct?
- Use of domain-specific languages
  - Make more programmers productive through restrictions on expressiveness
- Common specifications for critical components