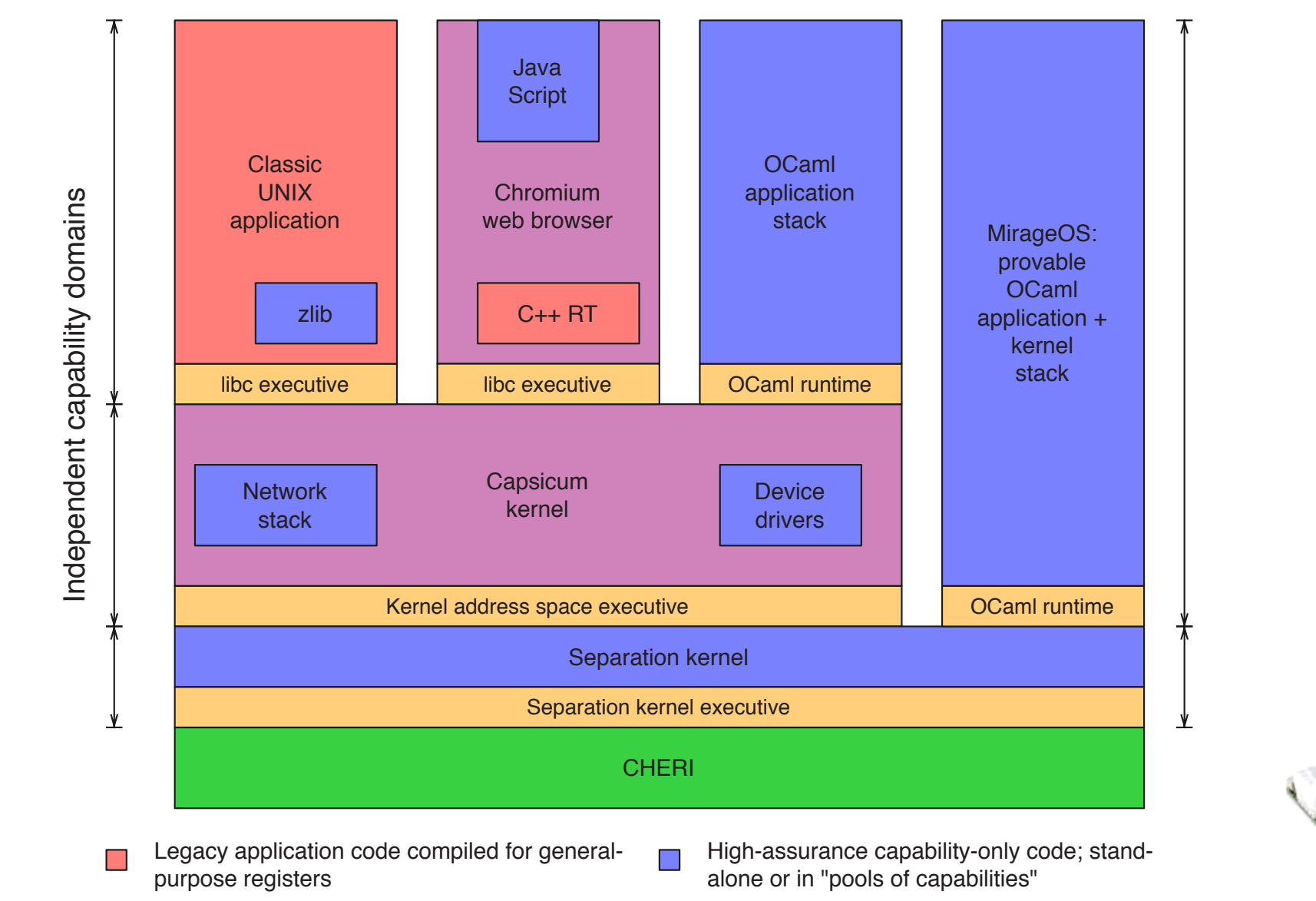
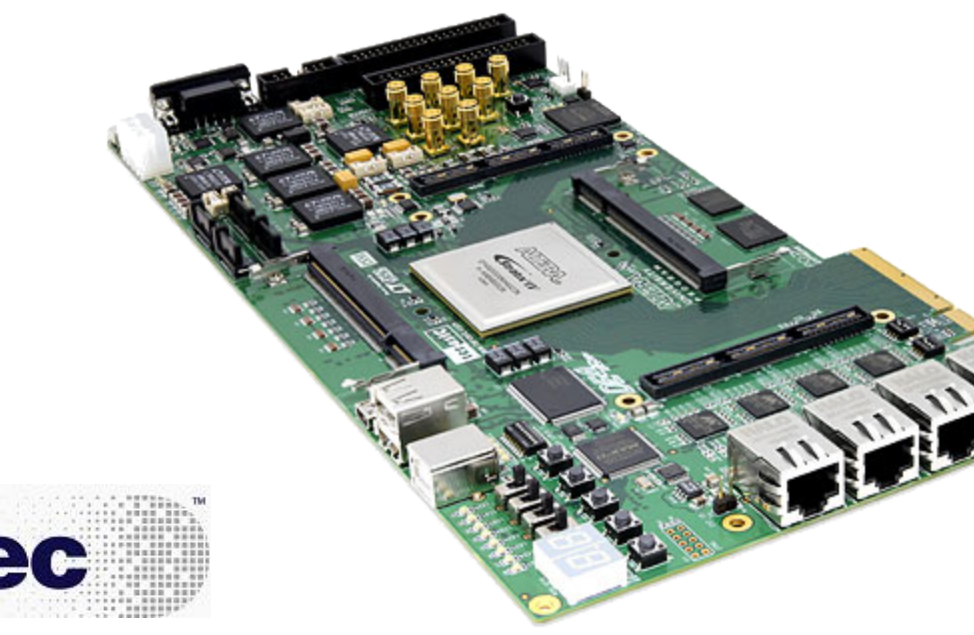
CTSRD

Peter G. Neumann, Robert N. M. Watson, Ross Anderson, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Rance DeLong, Khilan Gudka, Steven Hand, Ben Laurie, Patrick Lincoln, Anil Madhavapeddy, Andrew W. Moore, Alan Mujumdar, Steven J. Murdoch, Robert Norton, Philip Paeps, Michael Roe, John Rushby, Hassen Saidi, Muhammad Shahbaz, Stacey Son, Jonathan Woodruff.

CTSRD is a principled, formally-supported, robust, programmer-friendly, high-performance and incrementally adoptable hardware/software platform that is designed for efficient software implementation of the principle of least privilege. Software security structures and design principles are reinforced by **Capability Hardware Enhanced RISC Instructions (CHERI)** and **Temporally Enforced Security Logic Assertions (TESLA)**. CTSRD adopts a hybrid approach, able to run existing C-language operating systems and applications while supporting gradual adoption of advanced security features by critical Trusted Computing Bases (TCBs) and high-risk software components. CTSRD allows programmers to wipe the slate clean incrementally.

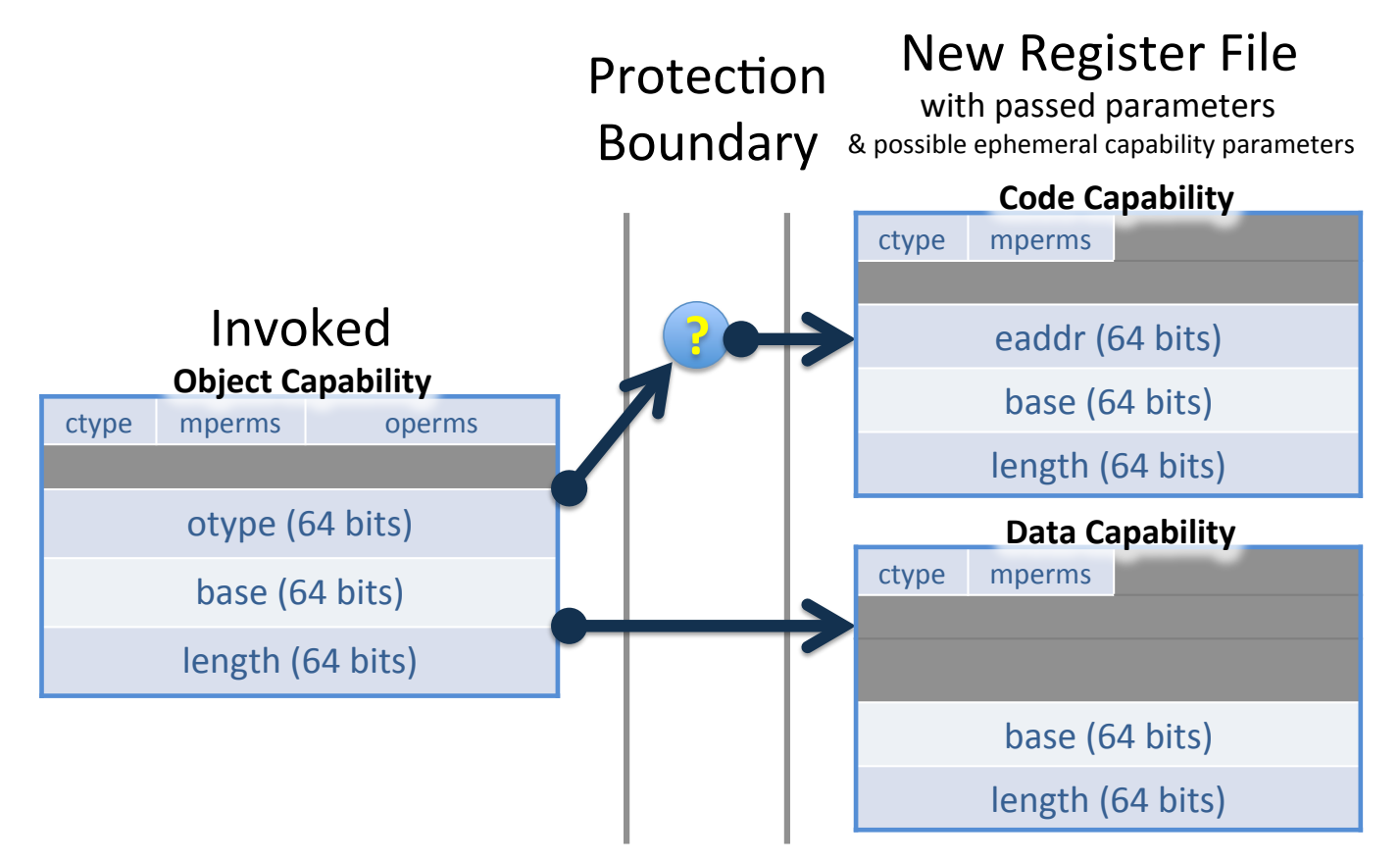
CHERI PROCESSOR AND SOFTWARE


Independent capability domains
Classic UNIX application
zlib
libc executive
Java Script
Chromium web browser
C++ RT
libc executive
OCaml application stack
OCaml runtime
MirageOS: provable OCaml application + kernel stack
Network stack
Capsicum kernel
Device drivers
Kernel address space executive
OCaml runtime
Separation kernel
Separation kernel executive
CHERI
General Purpose Registers
Capability Registers
PC
R0 == 0
R1
R2
R31
CR0 (general purpose capability)
CR1
CR2
CR31
perms otype base length
Legacy application code compiled for general-purpose registers
High-assurance capability-only code; stand-alone or in "pools of capabilities"
Hybrid code blending general-purpose registers and capabilities
Per-address space memory management and capability executive

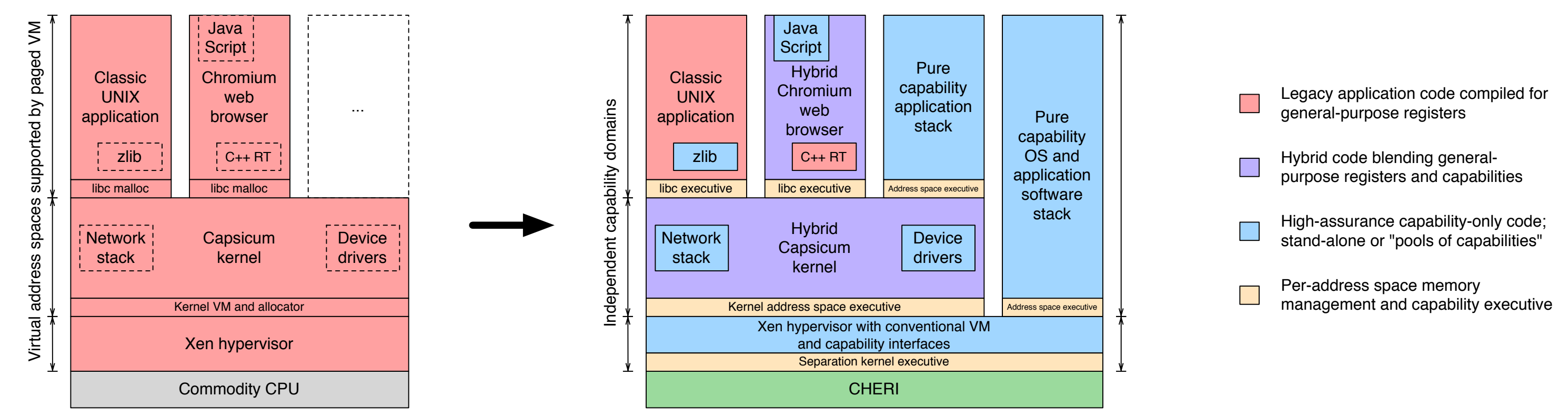


The **Capability Hardware Enhanced RISC Instructions (CHERI)** CPU architecture is motivated by the compartmentalisation problem: current instruction set architectures (ISAs) are unable to easily or efficiently represent decomposed software designs implementing the principle of least privilege. This problem results from a conceptual mismatch with Memory Management Unit (MMU)-based virtual address separation: Translation Look-aside Buffer (TLB)-related costs scale disproportionately with increases in compartmentalisation granularity, high memory overheads are implied by insufficient sharing, and communication between tightly coupled but mutually untrusting components is limited. Virtual addressing also makes it harder to develop and debug compartmentalised software. As a result, software developers are deterred from decomposing applications to mitigate security vulnerabilities or map distributed system security policies into local enforcement primitives – one of the only known techniques for mitigating both known and unknown vulnerabilities.

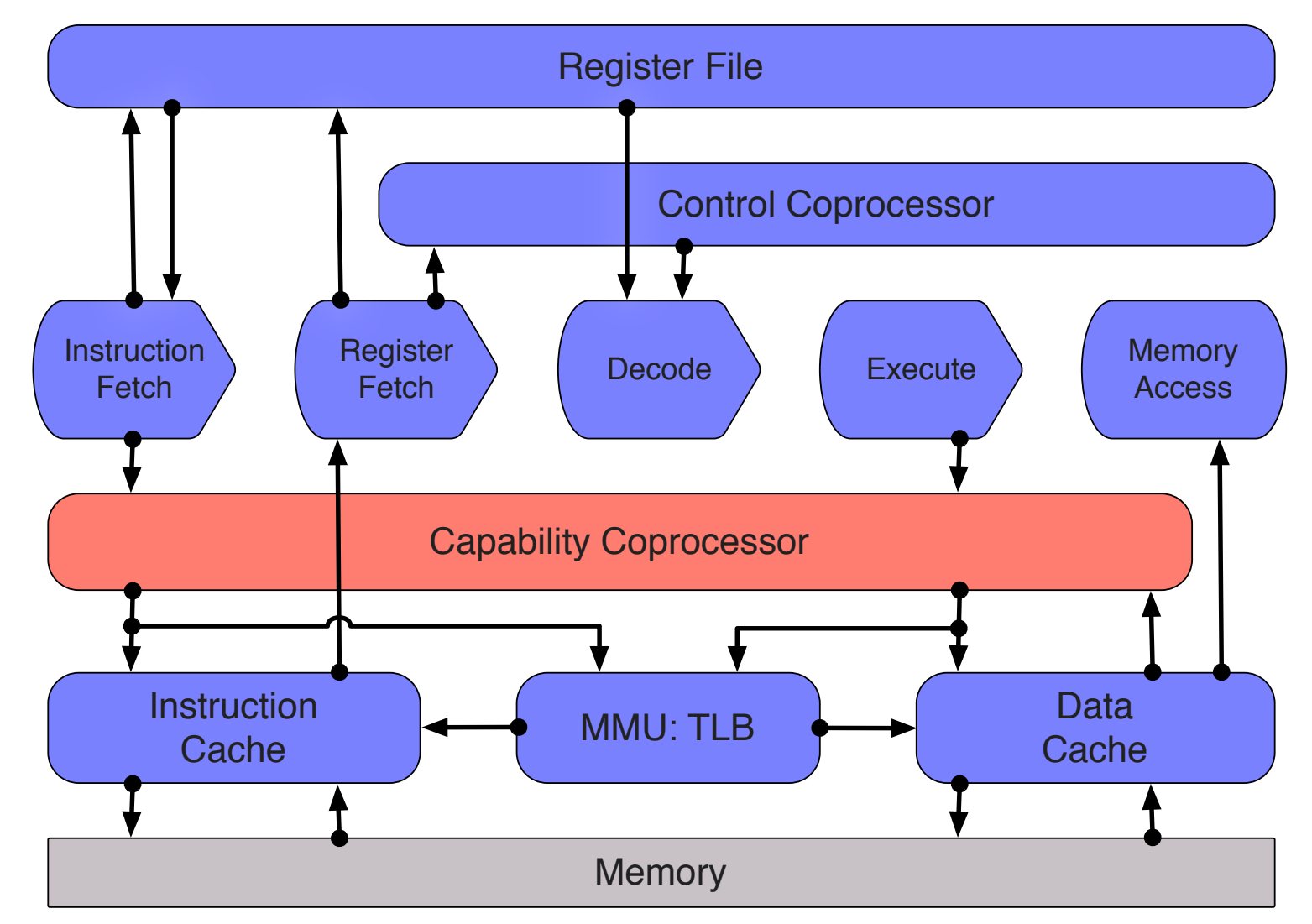
In CHERI, capability registers supplement general-purpose registers, allowing protection to be managed directly by the compiler. Capability features allow a large number of simultaneous and frequently switching security domains to co-exist efficiently, instead utilising more scalable data and code caches in the CPU rather than the TLB. Tagged memory allows capabilities, code, and data to safely coexist in system memory.



CHERI adopts a reduced instruction set computer (RISC) approach to capabilities, providing tools for compiler and operating system writers while minimising hardware complexity. CHERI's primitives allow simultaneous implementation of different security models, reflecting diverse OS, programming language, and application requirements. CHERI targets low-level software TCBs: OS kernels, language runtimes and web browsers, as well as high-risk data processing such as video decoding. CHERI's hybrid capability architecture allows capabilities to be adopted one software component at a time – unmodified but sandboxed programs fetch instructions, and load and store data via reserved capability registers. CHERI's composition of capabilities and the MMU places capability environments "above" the virtual address space: each UNIX process, as well as the kernel, has its own capability model. Subdivision within kernel and application address spaces using capabilities allows software to play by single address space rules, avoiding distributed system programming problems. The result is a complete software stack from day one.

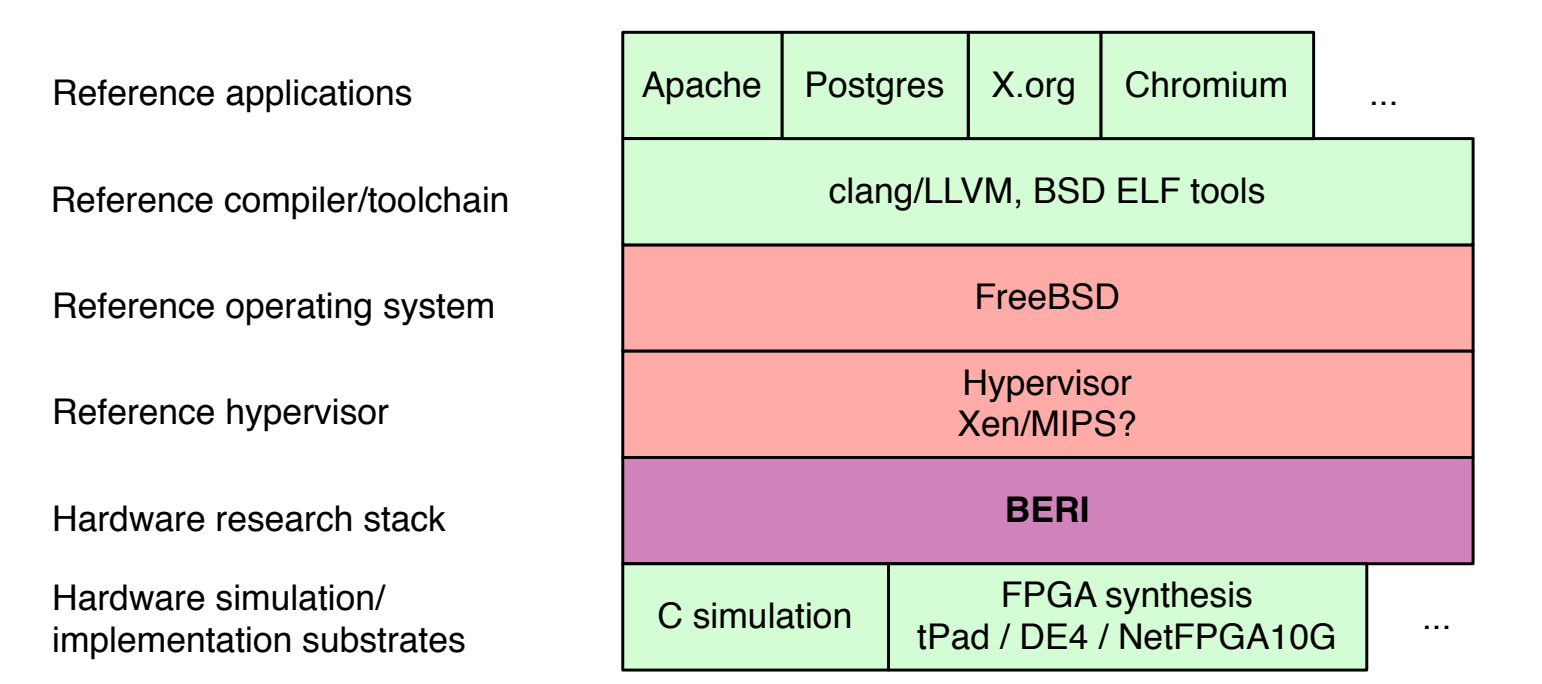

Virtual address spaces supported by paged VM
Classic UNIX application
zlib
libc malloc
Java Script
Chromium web browser
C++ RT
libc malloc
...
Network stack
Capsicum kernel
Device drivers
Kernel VM and allocator
Xen hypervisor
Commodity CPU
Independent capability domains
Classic UNIX application
Hybrid Chromium web browser
Pure capability application stack
Pure capability OS and application software stack
Network stack
Hybrid Capsicum kernel
Device drivers
Kernel address space executive
Address space executive
Xen hypervisor with conventional VM and capability interfaces
Separation kernel executive
CHERI
Legacy application code compiled for general-purpose registers
Hybrid code blending general-purpose registers and capabilities
High-assurance capability-only code; stand-alone or in "pools of capabilities"
Per-address space memory management and capability executive

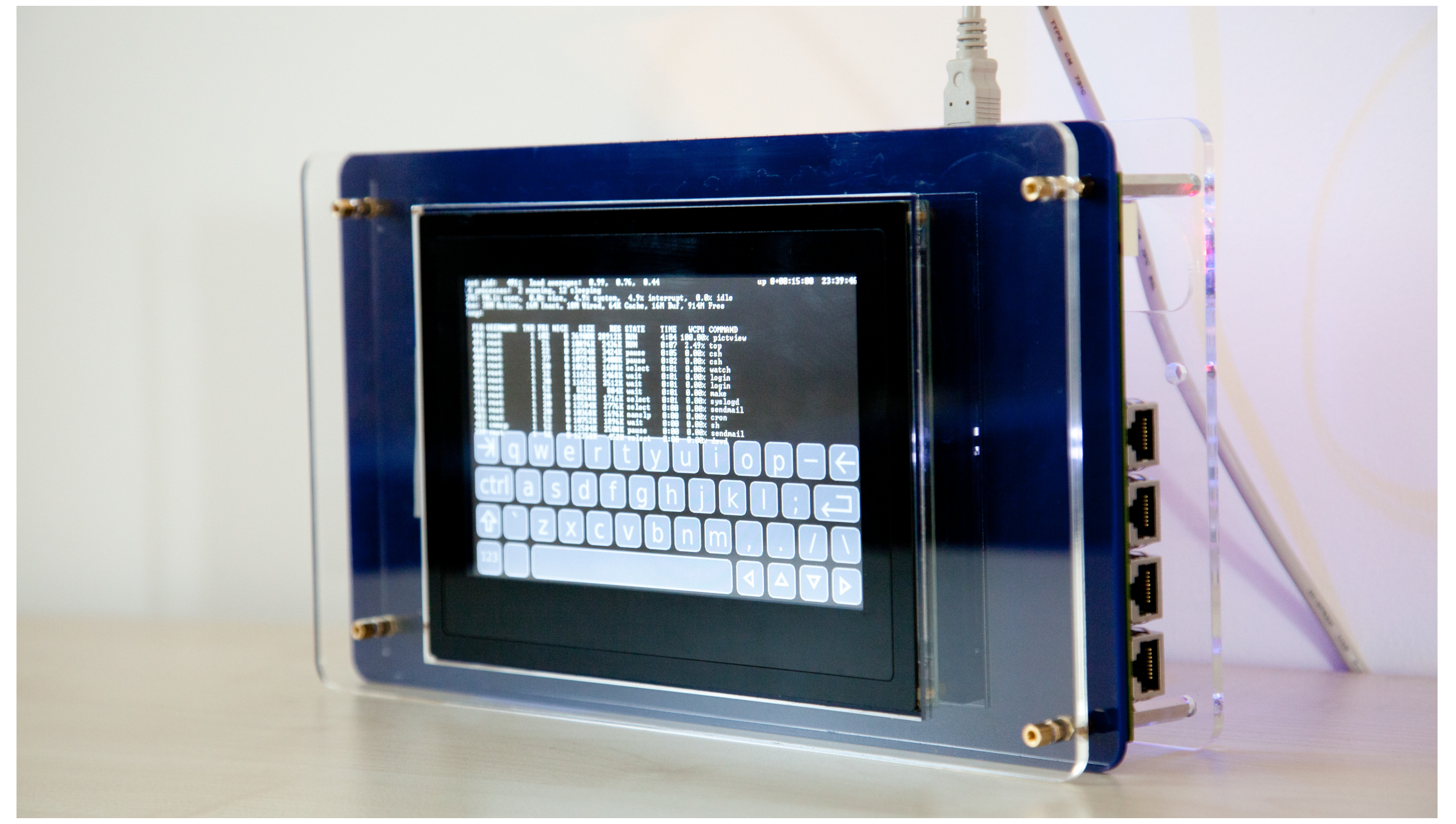
Within an address space, a thread's security context is captured by its capability register set: thread context switches are security context switches. CHERI is a multi-threaded processor supporting low-latency message passing of general-purpose and capability registers, which translates into efficient protected subsystem invocation – potentially orders of magnitude faster than can be supported in MMU-based designs.



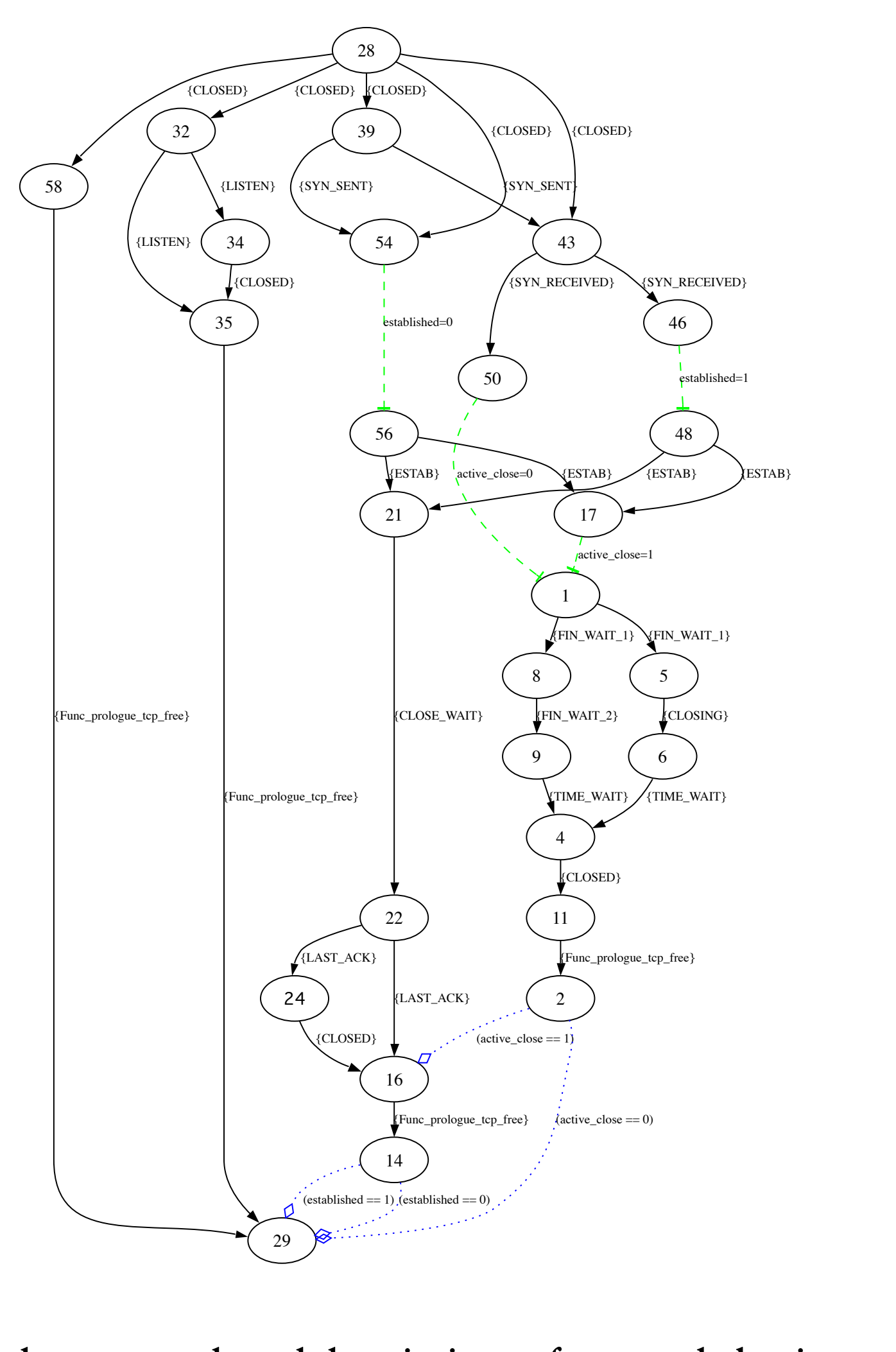
CHERI is built on the Bluespec Extensible RISC Instructions (BERI) 64-bit MIPS ISA processor, a generalisable platform for future research on the hardware/software interface. BERI and CHERI are written in the Bluespec SystemVerilog hardware description language (HDL), which facilitates rapid prototyping and design space exploration. We are investigating a mapping from Bluespec into SRI's Evidentiary Tool Bus (ETB), including PVS, SAL, and the Yices SMT solver, offering the promise of a formal grounding from hardware up – a technique we also hope to extend to verifying hardware and software in composition. BERI currently runs in a cycle-accurate software simulation, and in Altera Stratix® IV GX FPGAs on the Terasic DE4 board at 100 MHz.

The BERI stack is a complete open-source hardware/software research platform: CPU, OS, compiler, and applications. Employing commodity software demonstrates that CHERI is incrementally adoptable with immediate security benefits, while offering a long-term capability system vision motivated by the principle of least privilege.


Reference applications
Reference compiler/toolchain
Reference operating system
Reference hypervisor
Hardware research stack
Hardware simulation/ implementation substrates
Apache
Postgres
X.org
Chromium
...
clang/LLVM, BSD ELF tools
FreeBSD
Hypervisor Xen/MIPS?
BERI
C simulation
FPGA synthesis tPad / DE4 / NetFPGA10G
...


The CHERI tablet prototype, based on a Terasic DE4 FPGA development board, runs an enhanced version of the open source FreeBSD operating system. Capability-based compartmentalisation allows software to achieve scalable and fine-grained protection, mitigating known and unknown vulnerabilities.

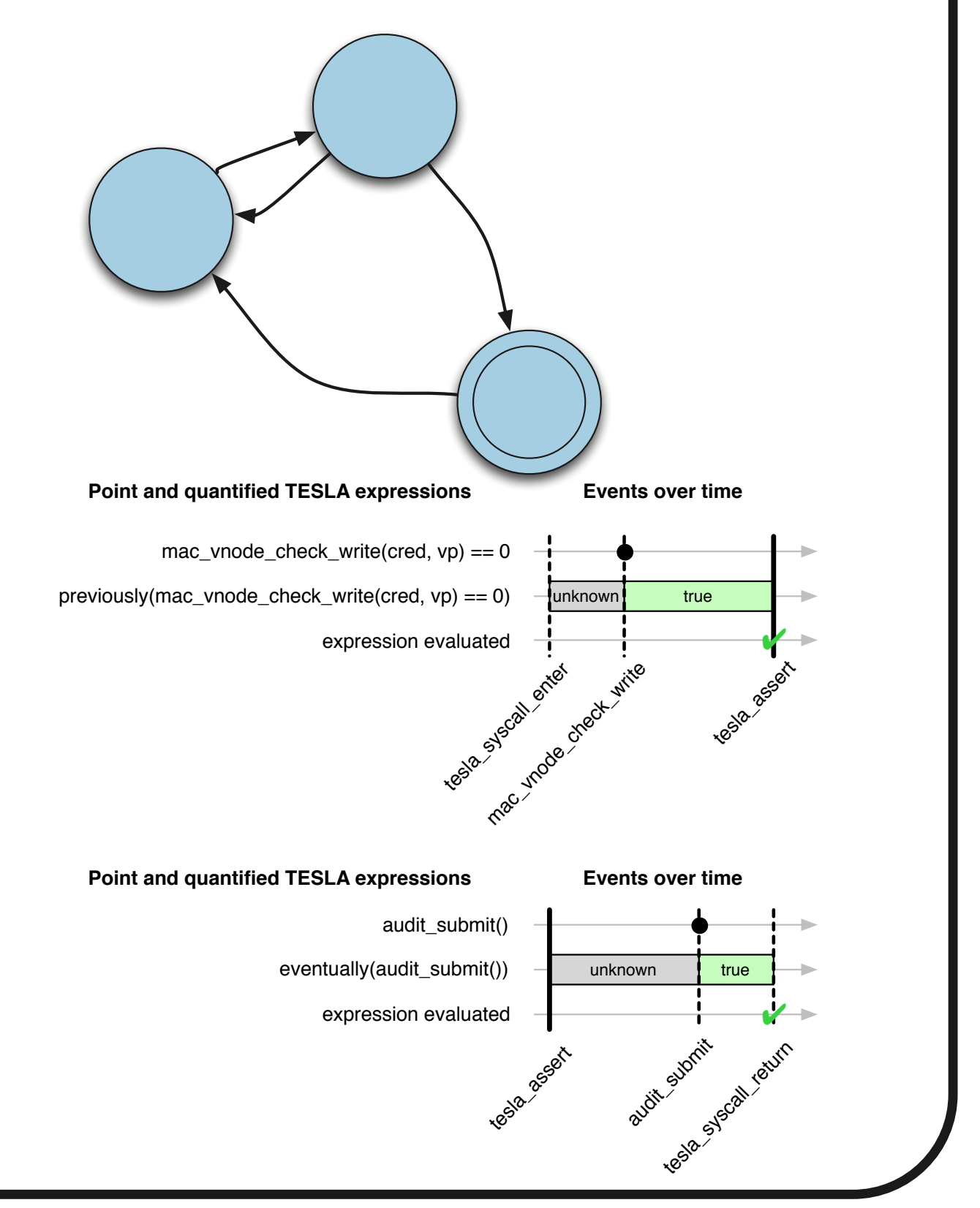
```
#include <netinet/tcp_fsm.h>

automaton tcp_connect() {

  void active_close(struct tcpcb *tp) {
    tp->t_state = TCPS_FIN_WAIT_1;
    either {
      tp->t_state = TCPS_CLOSING;
      tp->t_state = TCPS_TIME_WAIT;
    } or {
      tp->t_state = TCPS_FIN_WAIT_2;
      tp->t_state = TCPS_TIME_WAIT;
    }
    tp->t_state = TCPS_CLOSED;
    tcp_free(tp); exit;
  };

  void established(struct tcpcb *tp) {
    tp->t_state = TCPS_ESTABLISHED;
    either {
      active_close(tp->t_state);
    } or {
      tp->t_state = TCPS_CLOSE_WAIT;
      tp->t_state = TCPS_LAST_ACK;
      optional { tp->t_state = TCPS_CLOSED; }
    }
    tcp_free(tp); exit;
  };

  void main(struct tcpcb *tp) {
    tp->t_state = TCPS_CLOSED;
    either {
      tp->t_state = TCPS_LISTEN;
      optional { tp->t_state = TCPS_CLOSED; }
      tcp_free(tp);
    } or {
      optional { tp->t_state = TCPS_SYN_SENT; }
      either {
        tp->t_state = TCPS_SYN_RECEIVED;
        either {
          established (tp->t_state);
        } or {
          active_close(tp->t_state);
        }
      } or {
        established (tp->t_state);
      }
    } or {
      tcp_free(tp); exit;
    }
}
}
```

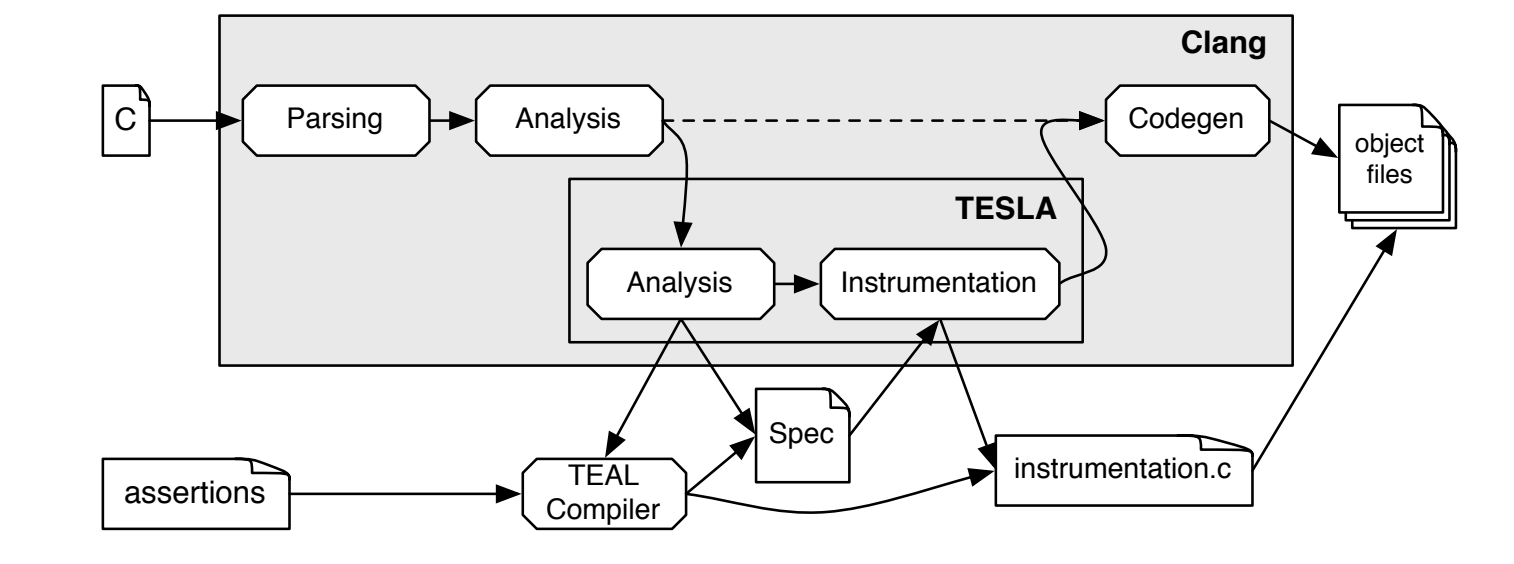


The TESLA assertion system allows temporal and automata-based descriptions of correct behaviour to be composed with C-language TCBs, checking access control and protocol properties dynamically. In the future, these may act as input to static analysis techniques.

TESLA ASSERTION SYSTEM

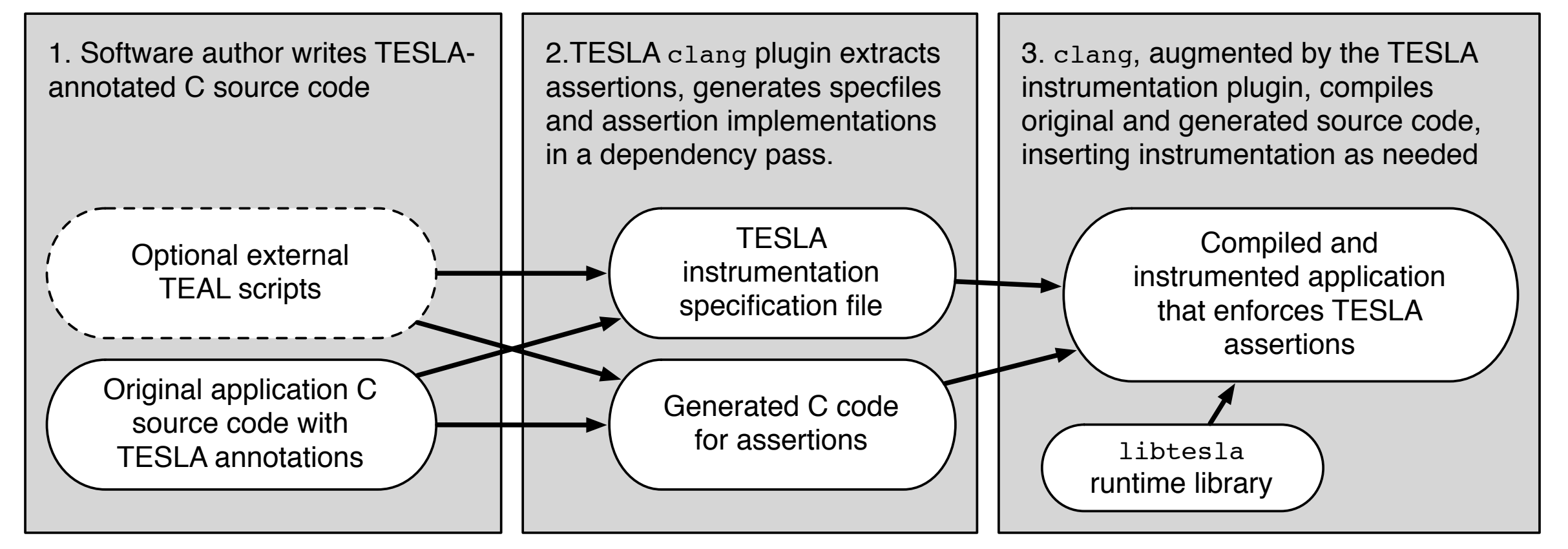
```
int
vn_rdwr(enum uio_rw, struct vnode *vp, ...,
  struct ucred *active_cred, struct ucred
  *file_cred, ...)
{
  if (rw == UIO_WRITE) {
    tassert_syscall(previously(mac_vnode_check_write(
      active_cred, file_cred, vp) == 0));
    tassert_syscall(eventually(audit_submit());
  ...
}
```


Point and quantified TESLA expressions
Events over time
mac_vnode_check_write(cred, vp) == 0
previously(mac_vnode_check_write(cred, vp) == 0)
expression evaluated
unknown
true
Point and quantified TESLA expressions
Events over time
audit_submit()
eventually(audit_submit())
expression evaluated
unknown
true

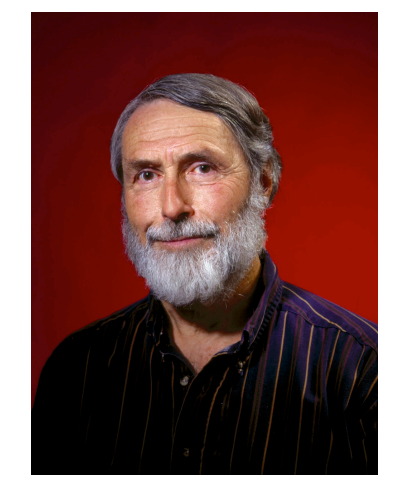


Temporally Enhanced Security Logic Assertions (TESLA)'s simple assertion language adds temporal quantifiers previously and eventually to the C assertion syntax, allowing assertions to refer to past and future events scoped to a timeline and programmer-selected ordering (per-thread or global). Assertions can also be expressed directly as automata using the TESLA Assertion Language (TEAL). Below, TESLA validates FreeBSD's TCP implementation by checking that assignments to the `tcpcb.t_state` field conform to the the TCP protocol specification. This technique can also be used to validate cryptographic protocol conformance (e.g.: IPSEC or SSH).

We are adding new assertion types to check sampled data distributions over time, and real-time properties. This will allow us to validate cryptographic and network protocol properties such as sequence number non-reuse within a window, and timely protocol rekeying.


1. Software author writes TESLA-annotated C source code
2.TESLA clang plugin extracts assertions, generates specfiles and assertion implementations in a dependency pass.
3. clang, augmented by the TESLA instrumentation plugin, compiles original and generated source code, inserting instrumentation as needed
Optional external TEAL scripts
Original application C source code with TESLA annotations
TESLA instrumentation specification file
Generated C code for assertions
Compiled and instrumented application that enforces TESLA assertions
libtesla runtime library

TESLA is implemented using a clang/LLVM-based C instrumentation framework and the `libtesla` run-time library. Assertions are converted into C, and the TESLA clang plugin instruments function prologues, epilogues, assignment through types, and other language-visible events. `libtesla` provides synchronisation and state management for in-flight automata. Fired assertions can trigger a kernel panic, stack trace, or DTrace probes that perform programmer, administrator, or user-scripted actions.


Dr Peter G. Neumann

Dr Robert N.M. Watson

Dr Simon W. Moore

Dr Nirav Dave

Mr Brooks Davis

Mr Rance DeLong

Dr Patrick Lincoln

Dr Andrew W. Moore

Mr Philip Paeps

Dr Michael Roe

Dr Hassen Saidi

Mr Stacey Son

Mr Jonathan Woodruff

Members of the CTSRD team and its external oversight group at our May 2011 review meeting in Cambridge, UK

Joe Stoy (Bluespec), Jonathan Woodruff (Cambridge), Ben Laurie (Google), Ross Anderson (Cambridge), Virgil Gligor (CMU), Philip Paeps (Cambridge), Li Gong (Mozilla), Peter Neumann (SRI)

Simon Cooper, Michael Roe (Cambridge), Robert Watson (Cambridge), Howie Shrobe (DARPA), Steven Murdoch (Cambridge), Sam Weber (NSF), Jonathan Anderson (Cambridge), Simon Moore (Cambridge)

Anil Madhavapeddy (Cambridge), Dan Adams (DARPA), Rance DeLong (LynuxWorks), Jeremy Epstein (SRI), Hassen Saidi (SRI)

UNIVERSITY OF CAMBRIDGE

Approved for public release. This research is sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contract FA8750-10-C-0237. The views, opinions, and/or findings contained in this article/presentation are those of the author/presenter and should not be interpreted as representing the official views or policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the Department of Defense.