CTSRD

Crash-worthy Trusted Systems Research and Development

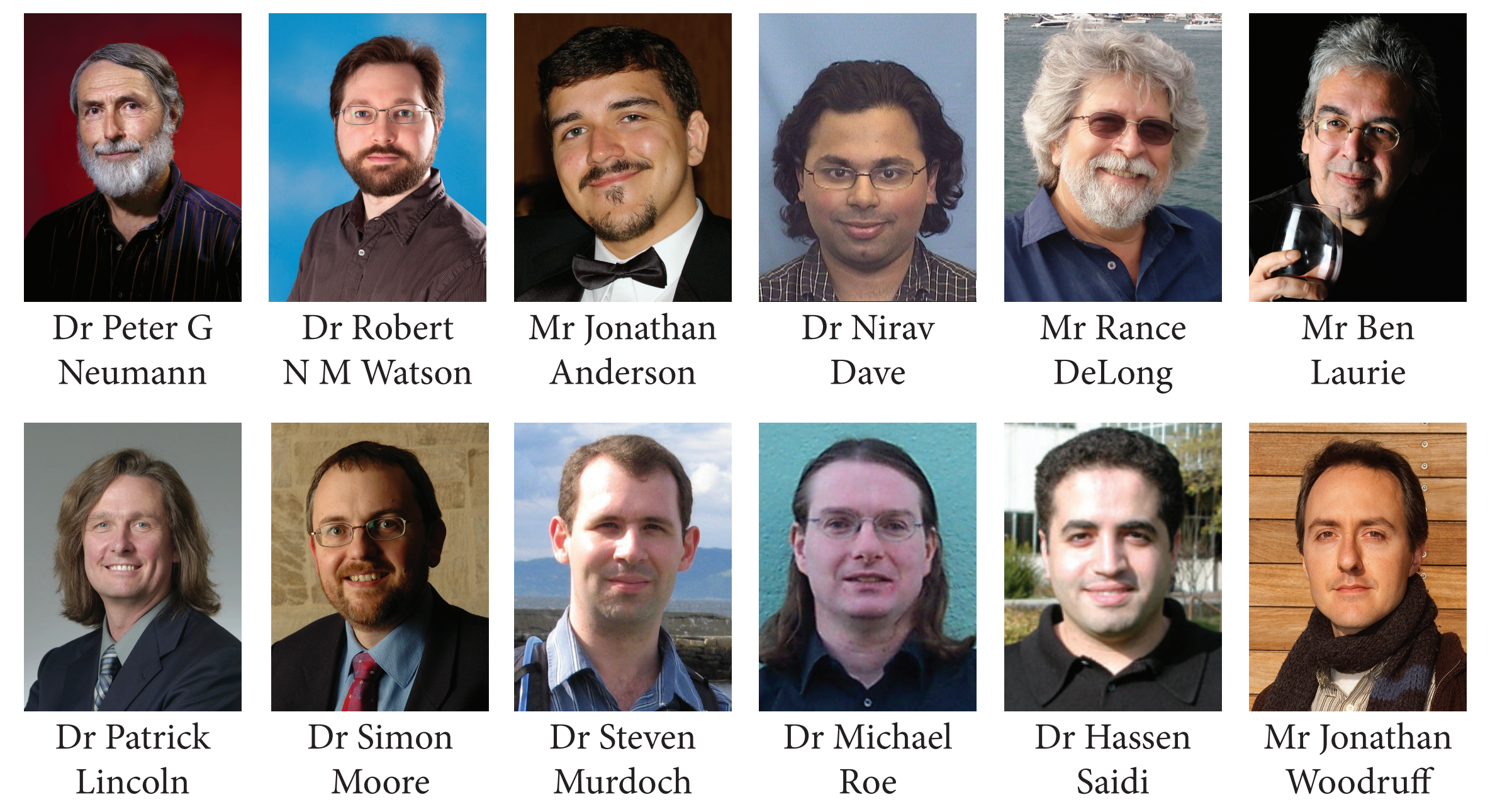
Peter G Neumann, Robert N M Watson, Ross Anderson, Jonathan Anderson, Nirav Dave, Steven M Hand, Wojciech Koszek, Ben Laurie, Patrick Lincoln, Anil Madhavapeddy,
Ilias Marinos, Andrew W Moore, Simon W Moore, William M Morland, Steven J Murdoch, Robert Norton, Philip Paeps, Michael Roe, John Rushby, Hassen Saidi, Jonathan Woodruff

CTSRD is a principled, formally-supported, robust, programmer-friendly, high-performance and incrementally adoptable hardware/software platform 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 operating systems and applications while supporting gradual adoption of advanced security features beginning with critical Trusted Computing Bases (TCBs) and high-risk software components.

CTSRD allows programmers to wipe the slate clean, one piece at a time.


Dr Peter G Neumann, Dr Robert N M Watson, Mr Jonathan Anderson, Dr Nirav Dave, Mr Rance DeLong, Mr Ben Laurie

Dr Patrick Lincoln, Dr Simon Moore, Dr Steven Murdoch, Dr Michael Roe, Dr Hassen Saidi, Mr Jonathan Woodruff

With External Oversight Group, May 2011

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)

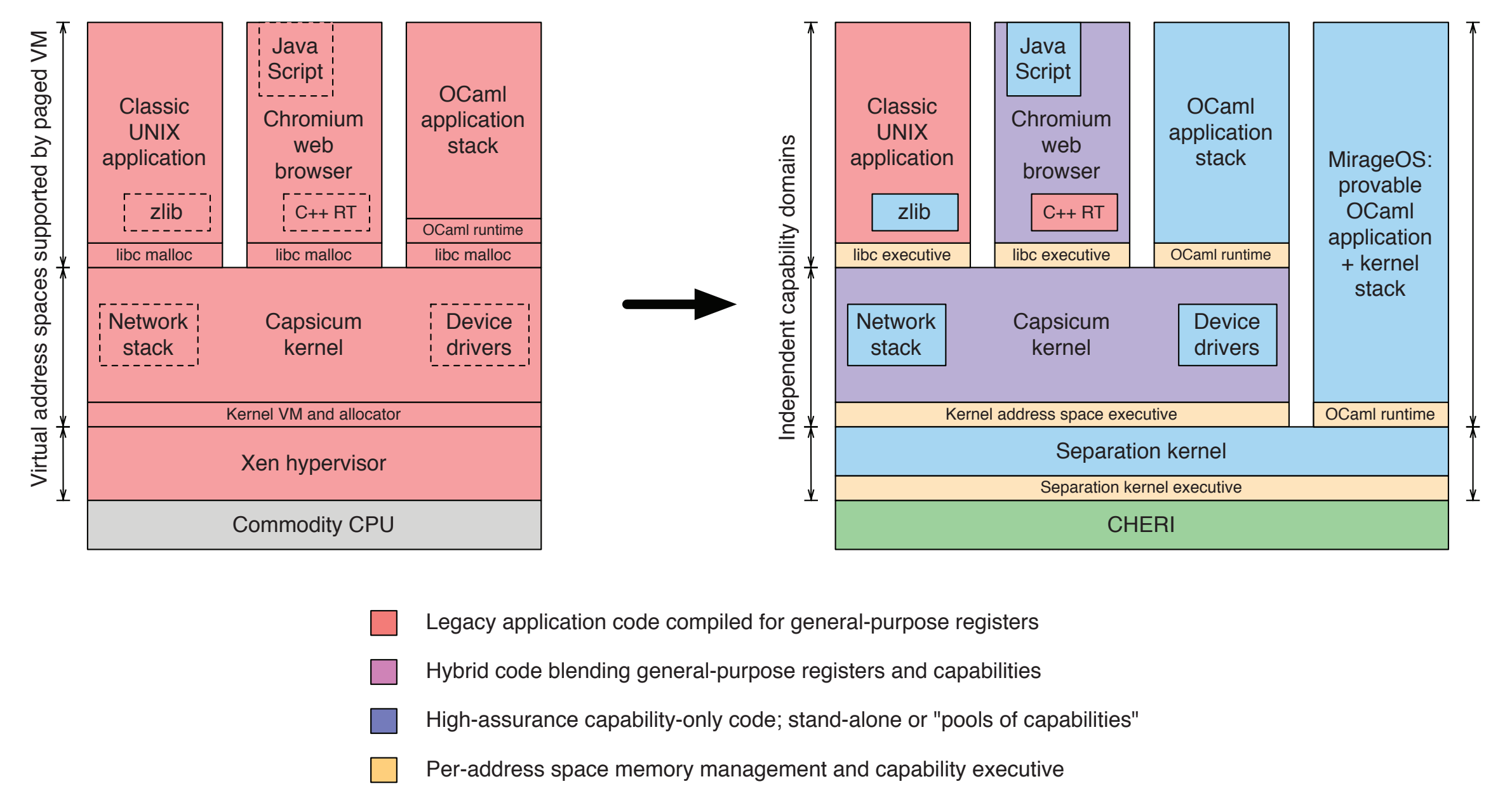
SRI International® UNIVERSITY OF CAMBRIDGE

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.

CHERI

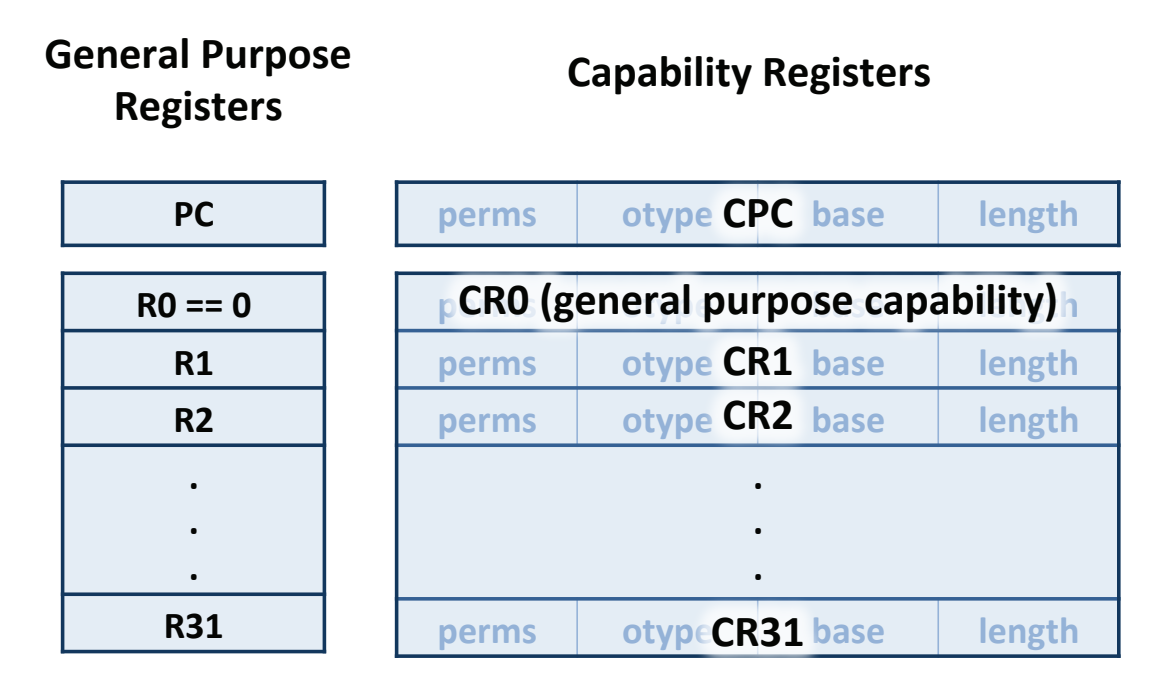
Capability Hardware Enhanced RISC Instructions

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 conceptual mismatch with **Memory Management Unit (MMU)**-based virtual address separation: **Translation Look-aside Buffer (TLB)**-related performance costs scale disproportionately to increase in compartmentalisation granularity. 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.


Virtual address spaces supported by paged VM
Classic UNIX application
zlib
libc malloc
Java Script
Chromium web browser
C++ RT
OCaml application stack
OCaml runtime
Network stack
Capsicum kernel
Device drivers
Kernel VM and allocator
Xen hypervisor
Commodity CPU
Independent capability domains
libc executive
MirageOS: provable OCaml application + kernel stack
Kernel address space executive
Separation kernel
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 "pools of capabilities"
Per-address space memory management and capability executive

CHERI addresses these problems through **efficient** and **compiler-friendly** hardware primitives to support the **object-capability security model**. In CHERI, manipulation of protection properties is as natural and lightweight as code and data manipulation is in commodity CPUs today.

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

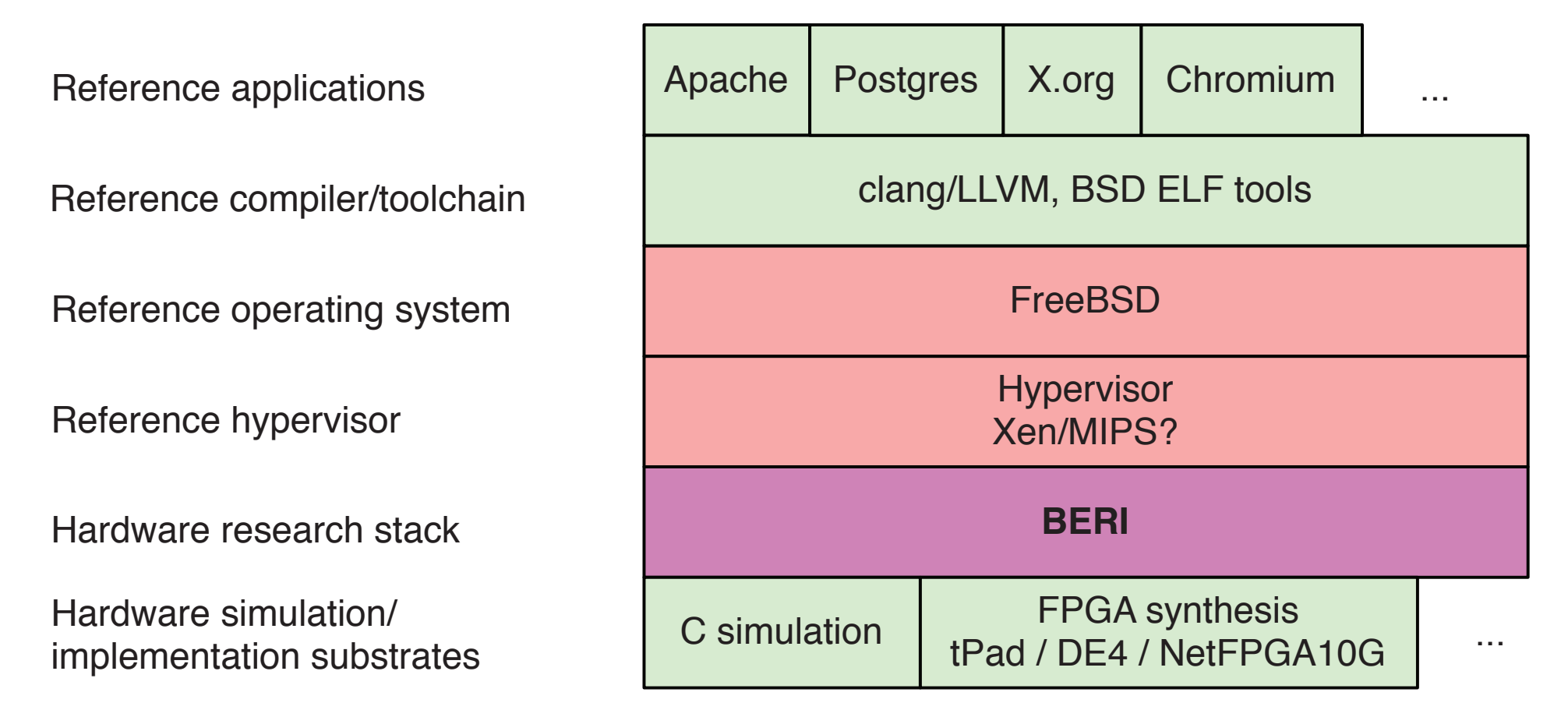


CHERI takes a **reduced instruction set computer (RISC) approach to capabilities**, providing tools for compiler and operating system writers while minimising hardware complexity. CHERI's primitives allow **simultaneous implementation of different security models**, reflecting diverse OSes, programming languages, 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. CHERI sandboxes unmodified RISC code by indirecting instruction fetches, loads and stores 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, can have 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**.

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

CHERI is founded on the **Bluespec Extensible RISC Instructions (BERI)** processor, a generalizable platform for future research on the **hardware/software interface**. The BERI hardware/software stack draws on existing **Apache- or BSD-licensed** software to provide a complete research platform, from the processor up to consumer applications and servers such as Chromium and Apache. CHERI is the first project to use BERI as a research platform.

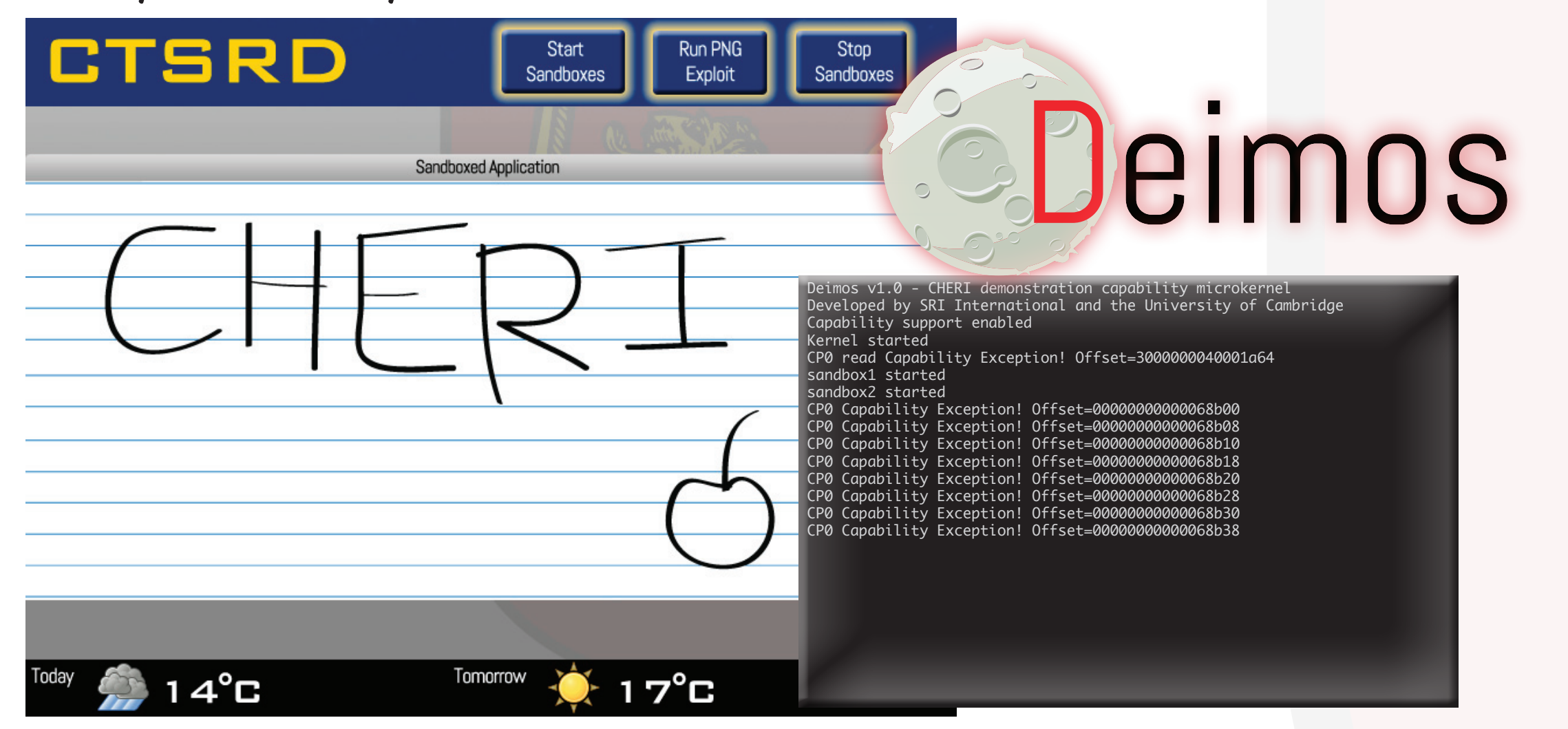

Reference applications
Reference compiler/toolchain
Reference operating system
Reference hypervisor
Hardware research stack
Hardware simulation/ implementation substrates
Apache
Postgres
X.org
Chromium
...
clang/LLVM, BSD ELF tools
FreeBSD
Hypervisor Xen/MIPS?
BERI
C simulation
FPGA synthesis tPad /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

CHERI runs in cycle-accurate software simulation, as provided by the Bluespec compiler, and in Altera Stratix® IV GX FPGAs on the Terasic DE4 development board, pictured to the right, at 100 MHz.

CHERI's 64-bit MIPS-derived CPU prototype, like the BERI prototyping platform it is built on, is written in the **Bluespec hardware description language** (HDL), which facilitates rapid prototyping and design space exploration. The starting point for our **hybrid software stack** is Cambridge's FreeBSD-derived **Capsicum hybrid capability operating system**. We are adapting the **clang and LLVM** compiler suite to directly support protection features in a modified IR. This software foundation will allow us to experiment with new ISA security features while running with a **complete software stack from day one**.

Capsicum

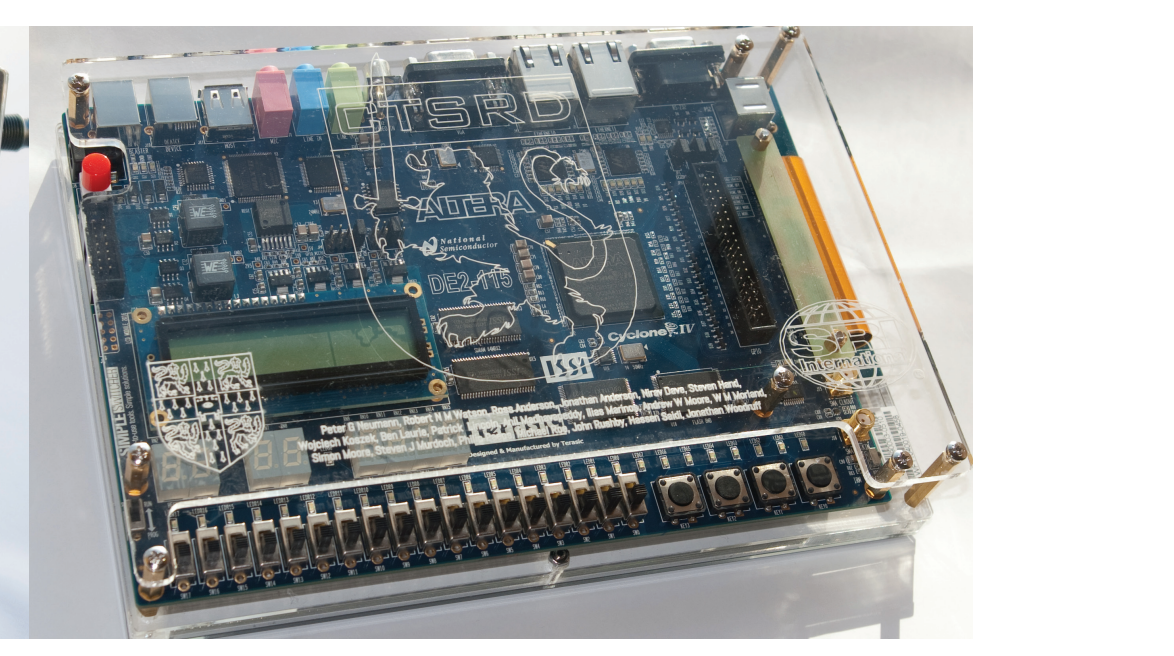
Recent work on CHERI has produced Deimos, a demonstration microkernel operating system which uses capabilities, rather than virtual memory, to isolate sandboxed processes. Processes running under Deimos can draw on portions of a touch screen, constrained by capabilities so that they cannot interfere with each other or with the system's trusted path. This all occurs within a single virtual address space: protection and virtual memory have finally been de-conflated.



```
Deimos v1.0 - CHERI demonstration capability microkernel
Developed by SRI International and the University of Cambridge
Capability support enabled
Kernel started
CP0 read Capability Exception! Offset=3000000040001a64
sandbox1 started
sandbox2 started
CP0 Capability Exception! Offset=00000000000068b00
CP0 Capability Exception! Offset=00000000000068b08
CP0 Capability Exception! Offset=00000000000068b10
CP0 Capability Exception! Offset=00000000000068b18
CP0 Capability Exception! Offset=00000000000068b20
CP0 Capability Exception! Offset=00000000000068b28
CP0 Capability Exception! Offset=00000000000068b30
CP0 Capability Exception! Offset=00000000000068b38
```

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.

CHERI is **incrementally adoptable** with **immediate security benefits**, while still offering a **long-term capability system vision** motivated by the principle of least privilege. System developers will be able to wipe the slate clean—one piece at a time.



TESLA

Temporally-enhanced Security Logic Assertions

TESLA is a system for writing and checking **assertions about the past and future** behaviour of software. It allows the authors of **Trusted Computing Bases (TCBs)**, such as operating system kernels and language runtimes, to ensure that their software is performing as expected, according to protocols such as "reads and writes must be preceded by access control checks".

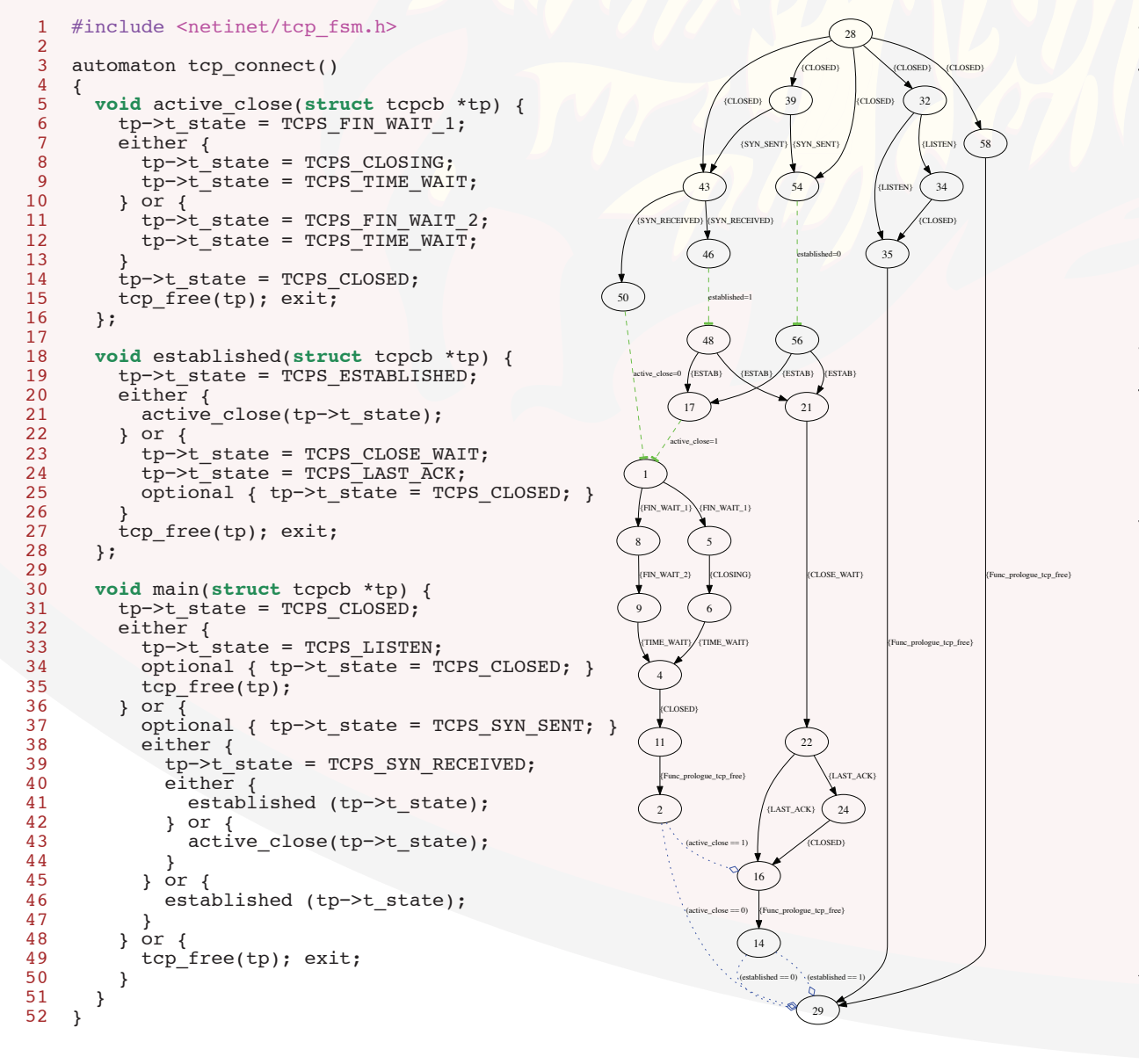
Like traditional, instantaneous assertions, TESLA assertions are written in the language of the source code itself, allowing the programmer to name functions, variables and values according to **normal C scoping rules**. Such assertions form a **high-level specification** of expected program behaviour, against which actual system behaviour can be **dynamically checked at runtime**.

```
  tesla_assert(
    TESLA_THREAD,   /* per-thread events */
    syscallenter(), /* start event */
    syscallret(),   /* stop event */
    (exp))          /* assertion to test */

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

Unlike traditional assertions, TESLA assertions can refer to events in the past or future, making them ideal for checking temporal properties such as:

- check before use (`!accessed(c, v) UNTIL checked(c, v)`)
- eventual audit (`accessed(v) → FINALLY audited(v)`)
- software and protocol state machines
- security meta-data life cycles and memory safety

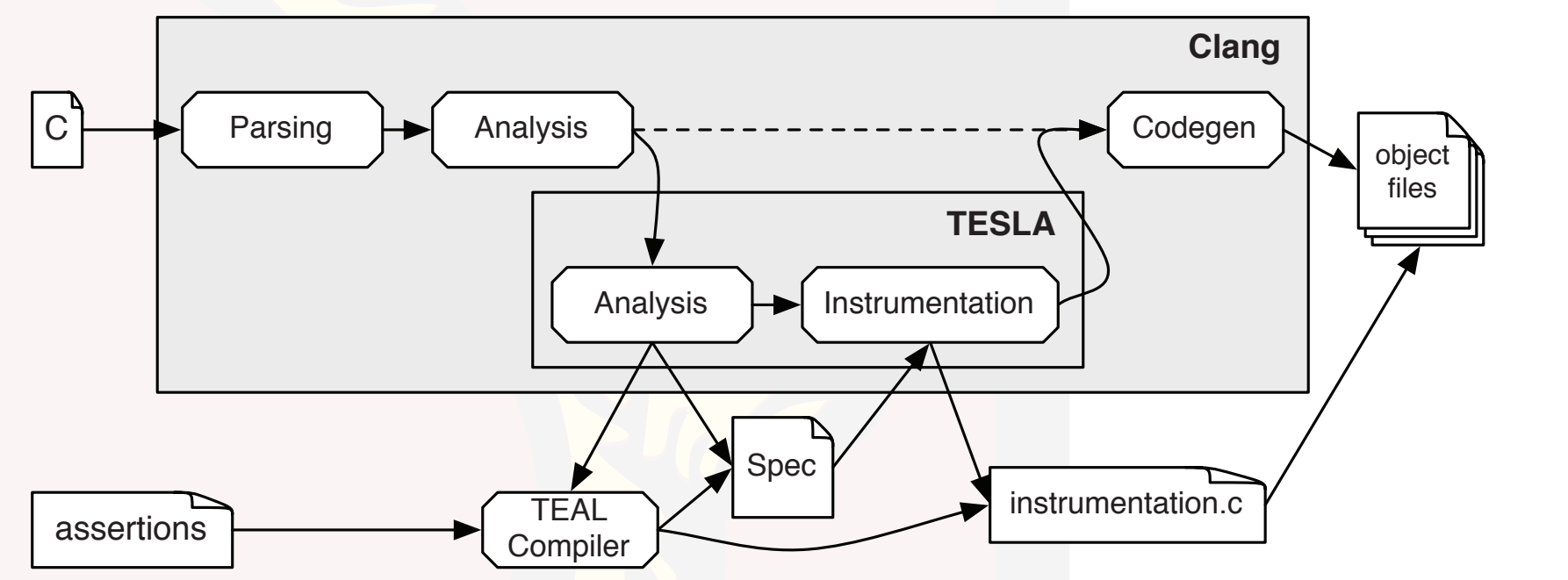
TESLA's simple assertion language adds the temporal quantifiers **previously** and **eventually** to the C assertion syntax; assertions refer to past and future events, scoped to a timeline and programmer-selected ordering (per-thread or global). We will add to TESLA's current syntax in order to express properties such as the statistical distribution of function outputs, so that programmers can write assertions about temporal properties such as the entropy of random number generators or the reuse of TCP ports.



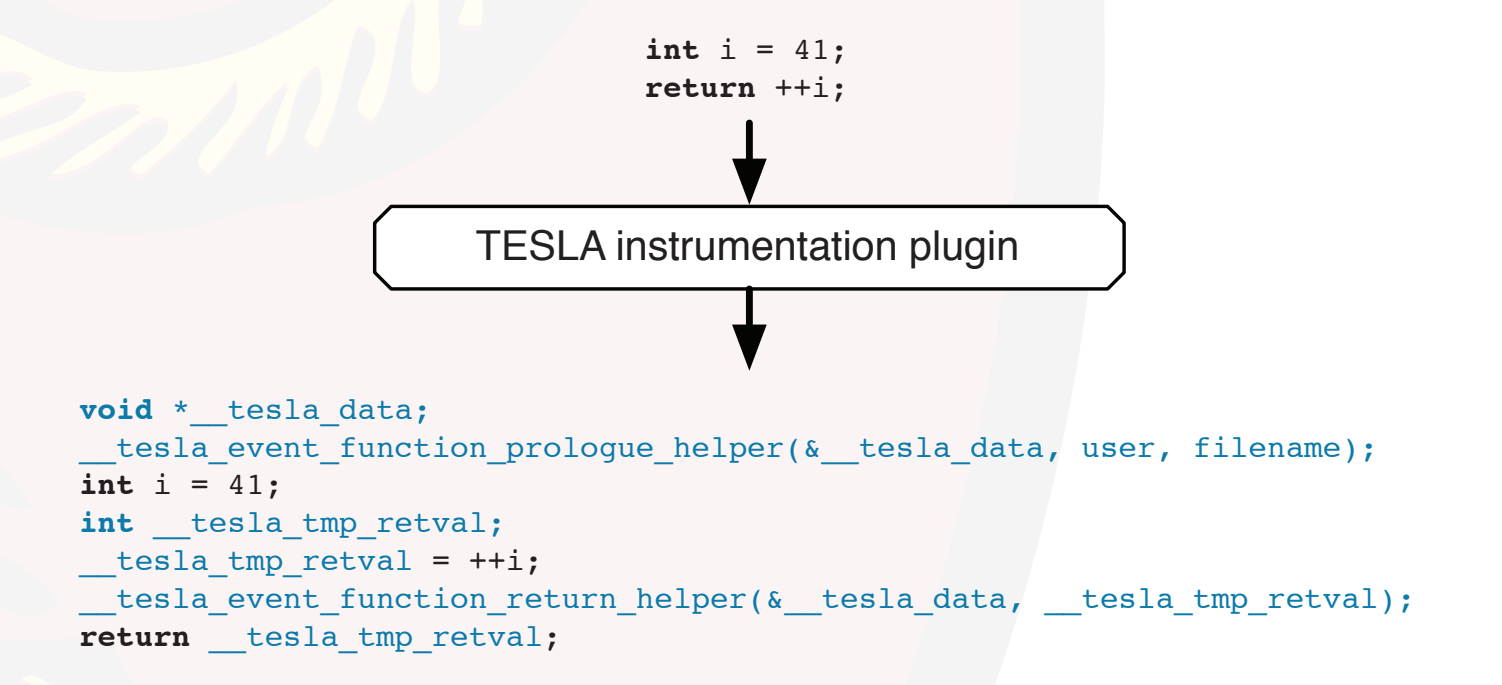
```
#include <netinet/tcp_fsm.h>

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;
    }
  }
}
```

Assertions can also be expressed as explicit automata using the **TESLA Assertion Language (TEAL)**. An example automata, representing the FreeBSD TCP implementation, is shown to the left. The TEAL automaton at the far left, which describes the expected lifecycle of a TCP connection in the FreeBSD network stack, is converted to the state machine at the near left, which can be continuously validated at runtime on a machine with a saturated 10Gb Ethernet connection with no measurable slowdown.

TESLA is implemented using a **clang-based C instrumentation framework** and the **libtesla** run-time library. Assertions are converted to C, and the clang plug-in 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 can themselves perform programmer-, administrator-, or user-scripted actions.



The instrumentation framework operates on top of the Abstract Syntax Tree (AST) layer, injecting instrumentation functions by modifying the AST. In the future, we will explore alternative ways of splitting analysis and instrumentation tasks between clang and LLVM.



```
int i = 41;
return ++i;
```

```
void *__tesla_data;
__tesla_event_function_prologue_helper(&__tesla_data, user, filename);
int i = 41;
int __tesla_tmp_retval;
__tesla_tmp_retval = ++i;
__tesla_event_function_return_helper(&__tesla_data, __tesla_tmp_retval);
return __tesla_tmp_retval;
```

The diagram above shows how instrumentation functions are invoked for function entry, assignment and function return events. In order to drive the state machines, which analyse the temporal behaviour of the system, **instrumentation functions** dispatch events to all automata interested in the specific event. When an unexpected event triggers an invalid state transition, the automaton is invalidated and a callback action is fired (configured in the runtime libtesla library).

We have used TESLA to check TCP state transitions in the FreeBSD kernel and rekeying behaviour in the OpenSSH client. We have checked security properties such as check-before-use in the FreeBSD Mandatory Access Control (MAC) framework, and plan to explore API conformance testing in OpenSSL and Cambridge's Xen hypervisor.