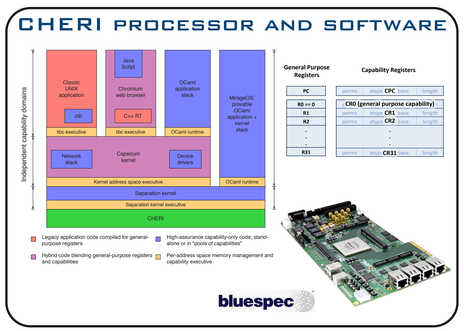
CTSRD

CRASH-worthy Trustworthy Systems Research and Development

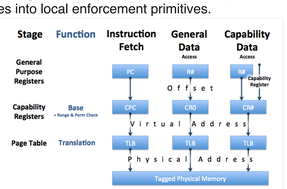
Peter G. Neumann, Robert N. M. Watson, Ross Anderson, Jonathan Anderson, Steven Hand, Patrick Lincoln, Anil Madhavapeddy, Andrew W. Moore, Simon Moore, Steven J. Murdoch, Philip Paeps, Michael Roe, John Rushby, Hassen Saidi, Jonathan Woodruff.

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.

CHERI PROCESSOR AND SOFTWARE



1. 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.

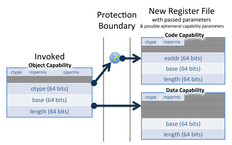


2. CHERI addresses these problems through **efficient** and **compiler-friendly** hardware primitives to support the **object-capability security model**. In CHERI, manipulation of protection properties is as natural and lightweight as code and data manipulation is in commodity CPUs today.

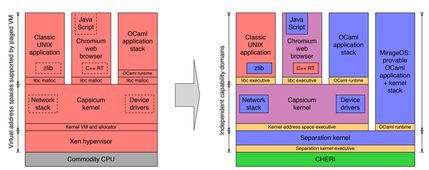
Capability registers supplement general-purpose registers, allowing protection to be managed directly by the compiler. CHERI's capability features allow a large number of simultaneous and frequently switching security domains to co-exist efficiently, utilising more scalable data and code caches in the 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 OS, programming language, and application requirements.

3. CHERI targets **low-level software TCBs**: OS kernels, language run-times, 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, transparently to other components. CHERI sandboxes unmodified RISC code by indirecting loads and stores via general-purpose registers through a reserved capability register. CHERI's composition of capabilities and the MMU places capability environments "above" the virtual address space. Subdivision of address spaces using capabilities allows compartmentalised applications to operate efficiently, but also play by single address space rules, avoiding **distributed system programming problems**.



4. 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, orders of magnitude faster than can be supported in MMU-based hardware designs.

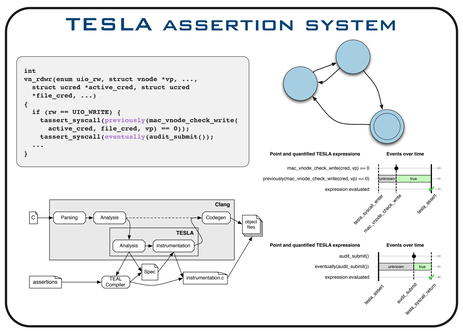


5. CHERI's 64-bit MIPS-derived CPU prototype is written in the **Bluespec hardware description language** (HDL), which facilitates rapid prototyping and design space exploration. As a starting point for our **hybrid software stack**, we have selected Cambridge's FreeBSD-derived **Capsicum hybrid capability 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**.

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 ASSERTION SYSTEM



6. TCB security policy implementations are often **artifacts of security policy** rather than its direct representation as **security policies.** Testing the correspondence between a policy and its implementation is often difficult, as **security policies are often temporal properties**, and **software assertions test only instantaneous properties.** Programmers must resort to **verbose**, **time-consuming**, and **error-prone** manual instrumentation and state management to test critical 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

```
#define tassert_syscall(exp)                    \
  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 *file_cred,
  ...)
{
  if (rw == UIO_WRITE) {
    tassert_syscall(previously(mac_vnode_check_write(
      active_cred, file_cred, vp) == 0));
    tassert_syscall(eventually(audit_submit()));
  }
  ...
}
```

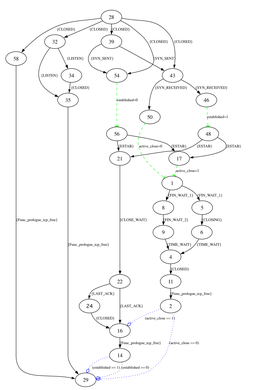
7. **Temporally Enforced Security Logic Assertions (TESLA)** employs ideas from model checking, projecting assertions into software as continuously validated automata. Programmers represent properties in a **simple temporal assertion language** or **explicit automata**:

- close to the code they describe,
- at arbitrary points in control flow,
- using similar syntax, types, and identifiers, as the program, and
- with to reference local and global program state.

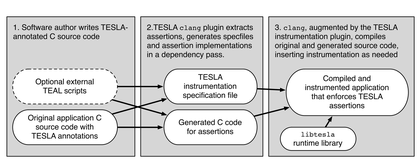
8. TESLA's simple assertion language adds temporal quantifiers **previously** and **eventually** to the C assertion syntax, allowing assertions to refer to past and future events scoped to a timeline and programmer-selected ordering (per-thread or global).

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).

```
#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;
    }
  }
}
```



9. We plan to add new assertion types checking **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.



10. 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 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 perform programmer, administrator, or user-scripted actions.

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)


Dr. Peter G Neumann


Dr. Robert N. M. Watson


Dr. Simon Moore


Mr. Jonathan Woodruff


Dr. Anil Madhavapeddy


Mr. Jonathan Anderson


Dr. Hassen Saidi


Mr. Ben Laurie


Dr. Patrick Lincoln


Mr. Rance DeLong

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.