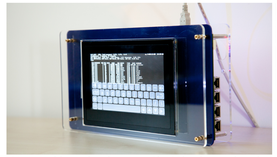
CTSRD

Peter G. Neumann, Robert N. M. Watson, and **Simon W. Moore**
Jonathan Anderson, Ross Anderson, David Chisnall, Nirav Dave, Brooks Davis, Rance DeLong, Khilan Gudka, Steven Hand, Alex Horsman, Jong Hun Han, Asif Khan, Myron King, Ben Laurie, Patrick Lincoln, Anil Madhavapeddy, Ilias Marinos, Dr Theo A. Markettos, Ed Maste, Andrew W. Moore, Alan Mujumdar, Prashanth Mundkur, Steven J. Murdoch, Robert Norton, Philip Paeps, Michael Roe, Colin Rothwell, John Rushby, Hassen Saidi, Muhammad Shahbaz, Stacey Son, Richard Uhler, Philip Withnall, Jonathan Woodruff, Bjoern A. Zeeb

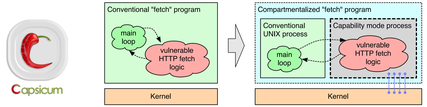


CHERI tablet and rack-mount CheriCloud array, based on Terasic's DE4 FPGA board. CHERI CPUs support fine-grained compartmentalization, mitigating broad classes of known and unknown vulnerabilities.

CTSRD is developing a principled, formally-supported, robust, programmer-friendly, high-performance and incrementally adoptable hardware/software platform designed for efficient implementation of the principle of least privilege. Hardware and software security structures and design principles are reinforced by:

- **Capability Hardware Enhanced RISC Instructions (CHERI)**
- **Security Oriented Analysis of Application Programs (SOAAP)**
- **Smten formal verification suite for Bluespec HDL**
- **Temporally Enforced Security Logic Assertions (TESLA)**

CTSRD adopts a hybrid approach, running current C-language operating systems and applications while supporting gradual adoption of novel protection features in both critical Trusted Computing Bases (TCBs) and high-risk software components.



Capsicum and the application compartmentalization motivation

Programmers are turning to **application compartmentalization** to mitigate inevitable vulnerabilities: software is decomposed into sandboxes in accordance with the **principle of least privilege.** As granularity increases, individual sandbox rights decrease. Only the rights of compromised components are leaked, forcing attackers to exploit more vulnerabilities to accomplish the same goals.

However, **compartmentalization scalability** – utilization of increasing numbers of sandboxes – is constrained by performance and programmability limitations of current hardware and software. Today's CPU instruction set architectures (ISAs) reflect a 1990s design consensus **conflating virtualization and protection**, limiting protection scalability. Compartmentalizing applications using IPC-linked processes also introduces **distributed systems programming problems** for local applications.

Capability Hardware Enhanced RISC Instructions (CHERI)

CHERI provides fine-grained protection within address spaces, complementing virtual-memory based processes to efficiently support compartmentalization:

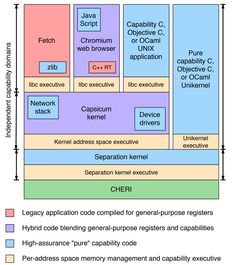
- Uses a reduced instruction set computer (RISC) approach, providing tools for compiler and operating system writers while minimizing hardware complexity.
- Targets low-level software TCBs: OS kernels, language runtimes and web browsers, as well as high-risk data processing such as video decoding.
- Allows simultaneous implementation of different security models, reflecting diverse OSs, programming languages, and application requirements.
- Implements a hybrid capability model supporting current software side-by-side with components employing fine-grained compartmentalization.

We have developed two CHERI prototypes, synthesizable to FPGA. CheriCloud allows remote access to systems running conventional and adapted software. BERI Open Systems CIC, a non-profit UK company, will open source CHERI in 2014.

CheriBSD hybrid capability OS

CheriBSD is an adaptation of the FreeBSD operating system to utilize features of the CHERI ISA. CheriBSD relies heavily on CHERI's hybridization support, allowing us to deploy in-process memory protection and sandbox incrementally. FreeBSD has been extended to:

- Maintain CHERI registers for user threads
- Selectively utilize CHERI memory protection via CHERI Clang/LLVM
- Implement CCall/CReturn fast exceptions: object-capability invocation
- Implement a CHERI "trusted stack" tracking object-capability invocation
- Recover from sandbox faults, such as memory errors, returning control to the invoking application (or sandbox)
- Provide a libcheri(3) API allowing applications to create and invoke sandboxes
- Implement monitoring extensions to the procstat(1) tool to track sandbox state
- Support CHERI debugging extensions for CHERI LLDB
- Support on-board peripherals such as Ethernet, display, flash, and SD card




```
% ssh ctsrd@cheritest.sec.cl.cam.ac.uk
Last login: Thu Nov 14 01:54:08 2013 from cl8188.aw.cl.cam.ac.uk
FreeBSD 11.0-CURRENT (CHERI_DE4_SORDOT) #4 011d9a4(master)-dirty: Mon Jan  6 13:19:44 GMT 2014
rt@cc8188:~% slogin rnw24@cheritest.sec.cl.cam.ac.uk

% procstat -RX 11180
  PID COMM     CLASS          METHOD     INVOKE FAULT   LMIN   LMAX   SMIN   SMAX  SMEAN  SMEDIAN
11180 cheritest  cheritest-helper.bin md5          5     0  11133  56396  11133  56396  22383    11831
11180 cheritest  cheritest-helper.bin abort        2     2   3048   3288   3048   3288   3168     3168
11180 cheritest  cheritest-helper.bin puts         3     0 419754 598131 419754 598131 508887   508777
11180 cheritest  cheritest-helper.bin syscall      3     0   6836   7581   6836   7581   7228     7269
11180 cheritest  cheritest-helper.bin divzero     4     4   2915   3188   2915   3188   3050     3049
```

In the coming months, we plan to implement further CheriBSD features including deploying CHERI memory protection through more of the CheriBSD userspace, and deploying additional sandboxing around the system.

Capability-enabled Clang and LLVM

We have extended the Clang/LLVM compiler to generate CHERI ISA instructions based on new C-language annotations and Internal Representation (IR) intrinsics. We add a `__capability` pointer qualifier, triggering CHERI instructions generation, inferring bounds checks and permissions from type information (e.g., arguments to allocation functions and `const`). These are dynamically enforced. MIPS and CHERI functions interleave seamlessly allowing gradual migration. Memory capabilities are the type-safety foundation for shared memory between protection domains.

Bounds checking

```
#include <capabilities.h>
...

// The compiler automatically inserts the bounds limits
capability int *buffer =
    (capability int*)malloc(size);

// Size can be computed from the capability here, or
// made explicit so the code can be compiled on a
// non-capability-aware architecture
fillArray(buffer, size);

// This overflows the bounds and so will trap at run time
int retVal = buffer[size];
```

```
# Call malloc()
ld    $25, %call16(malloc)($18)
jalr  $25
# Load the address of fillArray
ld    $25, %call16(fillArray)($18)
# Set the length of the capability
cincbase      $c1, $c0, $2
# Call fillArray
jalr  $25
# Set capability length (in branch-delay slot)
csetlen $c1, $c1, $16
# Load the value (causing capability violation trap)
clw   $16, $1, 0($c1)
```

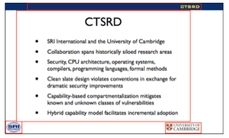
Capability tagging and permissions

```
// If this isn't a valid capability, do nothing
if (!__builtin_cheri_get_cap_tag(carray)) return;

// const is enforced with capability protections
capability const int *x = array;

// This will abort at run time:
fillArray((capability int*)x, cap_get_length(x));
```

```
# Get the tag bit
CGetTag $1, $c1
andi    $1, $1, 1
# If it's zero, skip to the return
beq $1, $zero, $081_2
nop
# Clear the write permission
ori $2, $zero, 65495
candperm    $c1, $c1, $2
# Load the address of the fillArray() function
ld  $25, %call16(fillArray)($1)
# Get the capability length for the second argument
cgetlen $2, $c1
# Call fillArray
jalr    $25
```

Application compartmentalization on CHERI

With increasingly mature CHERI processor prototypes, CHERI Clang/LLVM, and CheriBSD, we are turning our attention to deploying CHERI features around the OS and application. In November 2012, we demonstrated a bespoke presentation package, CheriPoint, able to perform granular sandboxing when rendering composed slide decks from many sources. We are now applying CHERI features in off-the-shelf applications such as tcpdump, experimenting with memory protection and fine-grained sandboxing.




```
# tcpdump -i ote0 | head -20
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on ote0, link-type EN10MB (Ethernet), capture size 65535 bytes
02:21:41.658068 IP cheritest.sec.cl.cam.ac.uk.ssh > cl0188.aw.cl.cam.ac.uk.49225: Flags [P.], seq
101973773:101973889, ack 1807744729, win 1040, options [nop,nop,TS val 1123802448 ecr 91588452], length 116
02:21:41.661524 IP cl0188.aw.cl.cam.ac.uk.49225 > cheritest.sec.cl.cam.ac.uk.ssh: Flags [.], ack 116, win 8184,
options [nop,nop,TS val 91597245 ecr 1123802448], length 0
02:21:41.688266 IP cheritest.sec.cl.cam.ac.uk.ssh > cl0188.aw.cl.cam.ac.uk.49225: Flags [P.], seq 116:232, ack 1,
win 1040, options [nop,nop,TS val 1123802478 ecr 91597245], length 116
02:21:41.691326 IP cl0188.aw.cl.cam.ac.uk.49225 > cheritest.sec.cl.cam.ac.uk.ssh: Flags [.], ack 232, win 8184,
options [nop,nop,TS val 91597275 ecr 1123802478], length 0
02:21:41.871254 IP gw-2456.route-nwest.net.private.cam.ac.uk.1985 > 224.0.0.102.1985: HSRPv1
02:21:41.929941 STP 802.1w, Rapid STP, Flags [Learn, Forward], bridge-id 83e7.00:1c:0e:50:40:00.8807, length 42
02:21:41.946293 STP 802.1w, Rapid STP, Flags [Learn, Forward], bridge-id 806c.00:1c:0e:50:40:00.8807, length 42
```

tcpdump is a widely used packet capture and interpretation tool; it is able to render and analyse packet sequences including low-level network characteristics (e.g., Ethernet/IP/TCP headers) and application-level features (e.g., NFS RPC decoding). With a large library of *protocol dissectors* able to render packets from a broad range of protocols, tcpdump is subject to a large attack surface. Further, tcpdump is also frequently used in security-sensitive environments: analysing traffic off the Internet, and when diagnosing suspicious traffic or in-progress attacks.

cheri_tcpdump is an adaptation of tcpdump using the CHERI model: memory protection, fine-grained compartmentalization, and incremental deployability. We are using tcpdump to explore *dynamic compartmentalization*: adaptive adjustment of performance investment in security responding to changing security parameters.

Security Oriented Analysis of Application Programs (SOAAP)

Experience with Capsicum shows that adapting programs for compartmentalization is difficult, leading to problems with correctness, performance, complexity, and critically, security. SOAAP is a set of techniques to assist with compartmentalization.

Compartmentalization hypotheses are explored through source-code annotations describing sandboxing strategy (e.g., sandbox creation, rights delegation, and RPC forwarding). Security goals and properties (e.g., information flow constraints and past vulnerabilities) as well as acceptable performance overhead are labeled in source code.

SOAAP engages the developer in a dialogue, identifying potential bugs (e.g., data inconsistencies), security breaches (e.g., information leaks), expected performance, and the impact of software supply-chain trojans.

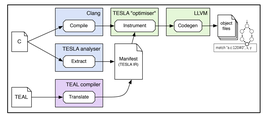
Past vulnerabilities

```
1  __soaap_sandbox_ephemeral("parser")
2  void parse(__soaap_read_fd int ifd, DOMTree* t) {
3    ...
4    if (...) {
5      __soaap_vuln_pt("CVE-2005-ABC");
6      ...
7    }
8    ...
9  }
10
11 __soaap_vuln_fn("CVE-2005-DEF")
12 void not_sandboxed() {
13   ...
14 }
```

```
$ make soaap
*** Sandboxed method "parse" has a past-vulnerability
*** annotation for "CVE-2005-ABC". Another vulnerability
*** here would leak the following:
*** Read access to file descriptor "ifd"

*** Method "not_sandboxed" had past vulnerability
*** "CVE-2005-DEF" but is not sandboxed. Another
*** vulnerability here could leak ambient authority
*** including full network and file system access.
```

Temporally Enhanced Security Logic Assertions (TESLA)

TESLA allows programmers to describe temporal properties of security-critical software and validate them at runtime. Programmers describe these properties with inline assertions or explicit finite-state automata. Both forms of TESLA specification are written in C, referencing names from surrounding scopes and exploiting the compiler's type checker. Both are converted to a TESLA intermediate representation (IR), allowing other languages to target the TESLA backend as well.



In the example below, the programmer asserts that the X.509 certificate passed to the `use_cert` function has been properly verified. In fact, however, the certificate verification code suffers from the vulnerability in CVE-2008-5077: an OpenSSL error code has been misinterpreted as success.

```
void
use_cert(X509 *cert)
{
#ifdef TESLA
  TESLA_WITHIN(main, previously(
      X509_STORE_CTX_init(ANY(ptr), ANY(ptr), cert, ANY(ptr)) == 1,
      X509_verify_cert(ANY(ptr)) == 1
  ));
#endif

  /* use the certificate ... */
}
```

```
automaton {
  identifier {
    location {
      filename: "openssl-api.c"
      line: 188
      counter: 0
    }
  }
  context: ThreadLocal
  expression {
    type: SEQUENCE
    sequence {
      expression {
        type: FUNCTION
        function {
          function {
            name: "X509_STORE_CTX_init"
          }
          direction: Exit
          context: Callee
          [...]
          argument {
            type: Variable
            index: 0
            name: "cert"
          }
          [...]
          expectedReturnValue {
            type: Constant
            value: 1
          }
        }
```

TESLA instruments the program as expressed in LLVM IR. Any language that targets LLVM can be instrumented. TESLA generates code to convert program events like function entry/return or structure field assignment into symbols that are consumed by TESLA automata.

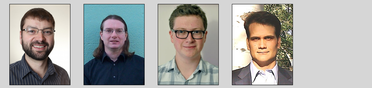
```
$ diff openssl-api.ll openssl-api.instr.ll

; Function Attrs: nounwind ssp uwtable
define i32 @main(i32 %argc, i8** %argv) #0 {
entry:
+  call void @__tesla_instrumentation_callee_enter_main(i32 %argc, i8** %argv)
   %retval = alloca i32, align 4
   %argc.addr = alloca i32, align 4
   %argv.addr = alloca i8**, align 8
@@ -147,6 +148,7 @@

 return:                                           ; preds = %if.end16, %if.then
   %20 = load i32* %retval, !dbg !649
+  call void @__tesla_instrumentation_callee_return_main(i32 %argc, i8** %argv, i32 %20)
   ret i32 %20, !dbg !649
 }

@@ -306,7 +308,9 @@
   %cert.addr = alloca %struct.x509_st*, align 8
   store %struct.x509_st* %cert, %struct.x509_st** %cert.addr, align 8
   call void @llvm.dbg.declare(metadata !{%struct.x509_st** %cert.addr}, metadata !780), !dbg !781
+  call void (i8*, i32, i32, %struct.__tesla_locality*, ...)* @__tesla_inline_assertion(i8* getelementptr
+  %instrumentation_cert = load %struct.x509_st** %cert.addr, !dbg !782
+  %0 = ptrtoint %struct.x509_st* %instrumentation_cert to i64, !dbg !782
+  call void @__tesla_instrumentation_assertion_reached_0(i64 %0), !dbg !782
   ret void, !dbg !783
 }
```

```
TESLA failure:
In automaton 'openssl-api.c:211#0':
automaton 0 {
  state 0: --(main(X,X): Entry)-->{1 <<init>>}
  state 1: --(X509_STORE_CTX_init(X,X,cert,X) == 1)-->{2}
  state 2: --(X509_verify_cert(X) == 1)-->{3}
  state 3: --(NOW)-->{4}
  state 4: --(main(X,X) == X)-->{5 <<cleanup>>}
  state 5:
}
openssl-api: No instance matched key '0x1 [ 7fda614147c0 X X X ]' for transition(s) [ (3:0x1 -> 4) ]
```

In this example, TESLA observes the `main()` entry event, so it creates an automaton instance and moves it from state 0 to state 1. However, it does not observe the `X509_STORE_CTX_init() == 1` event (this is the cause of the verification flaw), so when the `NOW` event occurs, TESLA cannot find an automaton instance named `(cert=0x7fda614147c0,*,*,*)`. The certificate has not been verified!

Dr Peter G. Neumann | Dr Robert N. M. Watson | Dr Simon W. Moore | Dr Jonathan Anderson | Dr David Chisnall

Dr Nirav Dave | Mr Brooks Davis | Mr Rance DeLong | Dr Khilan Gudka | Dr Theo A. Markettos



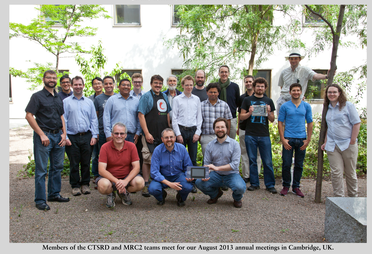
Mr Ed Maste | Dr Michael Roe | Mr Colin Rothwell | Mr Stacey Son

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)



Members of the CTSRD and MRC2 teams meet for our August 2013 annual meetings in Cambridge, UK.

Ed Maste, Prashanth Mundkur, Steven Murdoch, Jong Hun Han, Hassen Saidi, Khilan Gudka, Colin Rothwell, Peter G. Neumann, Mathe Schwarzkopf, Brooks Davis, Nirav Dave, Jonathan Woodruff, Bjoern Zeeb, Robert Norton, David Chisnall, Alan Mujumdar, Alex Horsman

Andrew Moore, Simon Moore, Robert Watson

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.