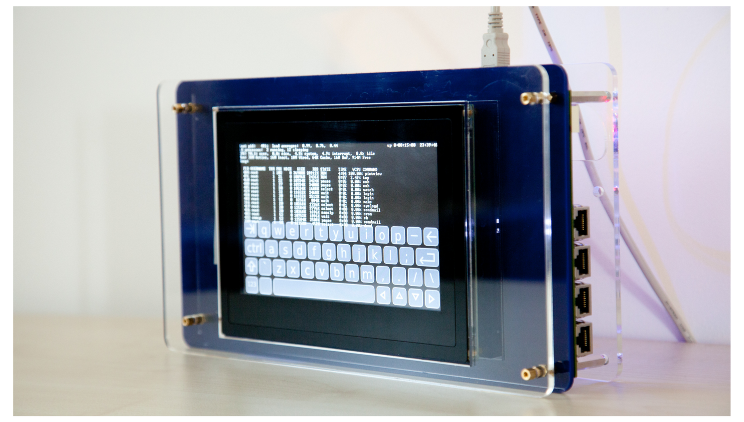
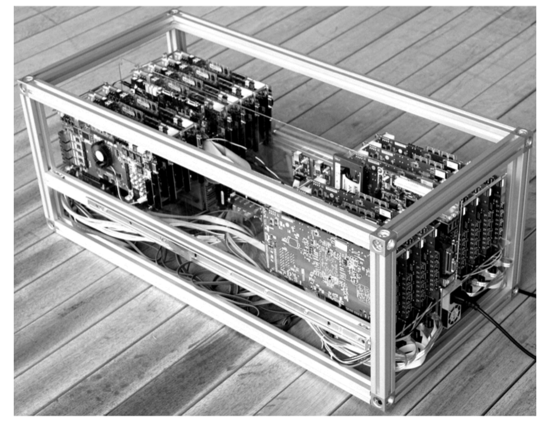
CTSRD

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, Asif Khan, Myron King, Ben Laurie, Patrick Lincoln, Anil Madhavapeddy, Ilias Marinos, Andrew W. Moore, Alan Mujumdar, Prashanth Mundkur, Steven J. Murdoch, Robert Norton, Philip Paeps, Michael Roe, John Rushby, Hassen Saidi, Muhammad Shahbaz, Stacey Son, Richard Uhler, Philip Withnall, Jonathan Woodruff, Bjoern A. Zeeb

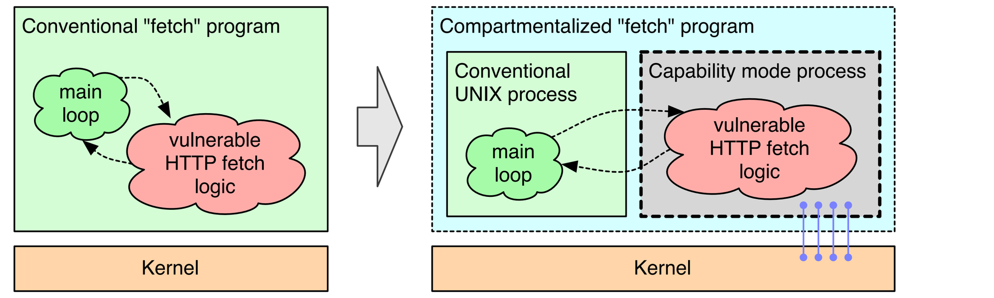



CHERI tablet and rack-mount BlueHive array, based on Terasic's DE4 FPGA board. CHERI CPUs support fine-grained compartmentalization, mitigating broad classes of known and unknown vulnerabilities.

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, able to run existing C-language operating systems and application software while supporting gradual adoption of its novel protection features for critical Trusted Computing Bases (TCBs) and high-risk components. CTSRD allows programmers to wipe the slate clean incrementally.



Capsicum and the application compartmentalization motivation

Programmers are turning to **application compartmentalization** to mitigate inevitable vulnerabilities: software is decomposed into sandboxed components, each with only the rights it requires to function. This approach employs the **principle of least privilege**: as granularity increases, rights delegated to individual sandboxes decrease. As vulnerabilities are exploited, only the rights of the affected component are leaked, forcing attackers to exploit many more vulnerabilities to accomplish the same goals.

The **Capsicum hybrid capability model** developed by Cambridge and Google blends contemporary OS design with capability system security, addressing semantic mismatches between OS features and application compartmentalization. Rights are delegated using **capabilities**, unforgeable tokens of authority. **Capability-mode** processes have access only to explicitly delegated rights as the use of OS global namespaces is denied.

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.

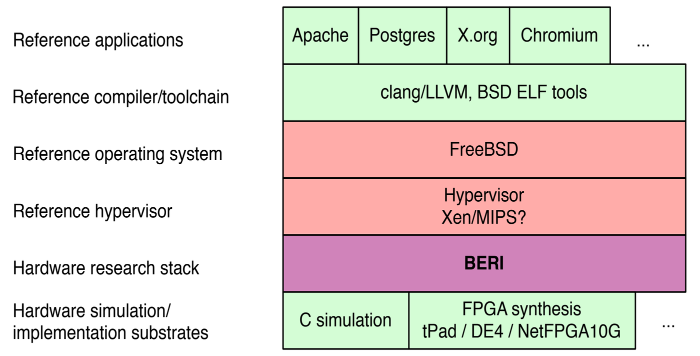
Current systems are exposed to greater threats, demanding dramatically increased use of compartmentalization, placing strain on protection and programming models designed for less risky workloads. CTSRD is developing clean-slate technologies to support large-scale deployment of compartmentalization for the first time.

Capability Hardware Enhanced RISC Instructions (CHERI)

CHERI provides a fine-grained protection model within address spaces, complementing existing virtual-memory based process models by providing efficient and more programmable support for application 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 64-bit CHERI prototypes, synthesizable for Terasic DE-4 and NetFPGA 10G platforms. Our CheriCloud facility allows remote access to CHERI systems able to run both conventional and adapted software. CheriCloud will be available to early adopters in the DARPA CRASH and MRC communities.



Capability-enabled Clang and LLVM

We have extended the Clang/LLVM compiler suite to generate CHERI ISA instructions based on C-language annotations and Internal Representation (IR) intrinsics. We add a `__capability` qualifier for pointers, causing CHERI instructions to be generated, inferring bounds checks and permissions from type information such as arguments to allocation functions and const. These are then dynamically enforced. MIPS and CHERI-aware functions may be interleaved seamlessly allowing a gradual migration. Memory capabilities form the type-safety foundation for shared memory between protection domains.

Bounds checking

```
// 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)
```

The CHERI-enabled C compiler will automatically set size information for capabilities constructed from dynamic and static allocations and allow automatic dynamic range checking.

Capability tagging and permissions

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

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

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

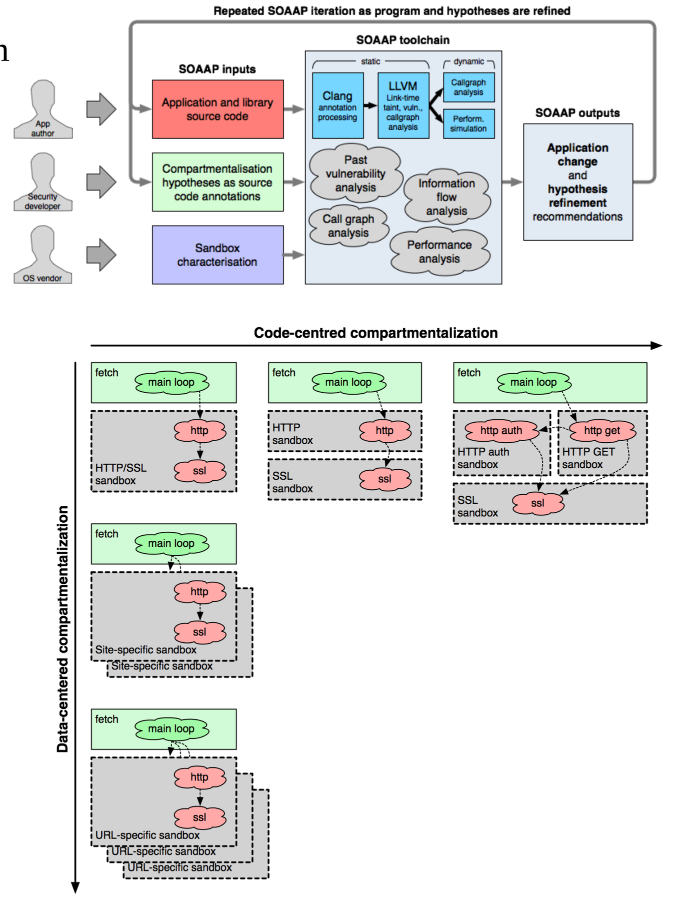
```
# Get the tag bit
CGetTag $1, $c1
andi    $1, $1, 1
# If it's zero, skip to the return
beq $1, $zero, $BB1_2
nop
# Clear the write permission
ori $2, $zero, 65495
candperm    $c1, $c1, $2
# Load the address of the fillArray() function
ld  $25, %call16(fillArray)($1)
# Get the capability length for the second argument
cgetlen $2, $c1
# Call fillArray
jalr    $25
```

We provide a set of builtin functions for manipulating capabilities directly in C and enforce properties at run time. This example inspects the length and passes that as an argument, and has a quick-exit path if handed an invalid capability.

http://www.cl.cam.ac.uk/research/security/ctsrd/cheri.html

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 semi-automated techniques to assist programmers 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 (such as information flow constraints and past vulnerabilities) as well as acceptable performance overhead can also be labeled in program source code.

SOAAP uses static and dynamic analysis to engage developers in interactive dialogue, identifying potential correctness bugs (e.g., data inconsistencies), security breaches (e.g., information leaks), expected performance and the impact of software supply-chain trojans. SOAAP builds on Clang and LLVM.

Data Consistency

```
1  __soaap_var_read("compress") int some_flag = 0;
2  int some_other_flag = 1;
3
4  void main() {
5    __soaap_create_sandbox("compress")
     ...
23   some_flag = get_option_from_cmd_line();
24   gz_compress(in, out);
25 }
26
27 __soaap_sandbox_persistent("compress")
28 void gz_compress(int ifd, int ofd) {
29   if (some_flag && some_other_flag) {
30     ...
31   }
32 }
```

SOAAP can find unintended accesses and missing data synchronisation.

```
$ make soaap
*** Write to shared global variable "some_flag" in method
*** "main" will not be seen by sandbox "compress".
*** Synchronization is needed to propagate this update to
*** the sandbox.
+++ Line 23 of file compress.c

*** Sandboxed method "gz_compress" read global variable
*** "some_other_flag" but it is not allowed to. If the
*** access is intended, the variable needs to be annotated
*** with __soaaap_read_var.
+++ Line 29 of file compress.c
```

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");
       ...
10   }
     ...
13 }
14
15 __soaap_vuln_fn("CVE-2005-DEF")
16 void not_sandboxed() {
17   ...
18 }
```

SOAAP can evaluate effectiveness in mitigating past vulnerabilities.

```
$ 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.
```

Confidentiality

```
1  __soaap_classify("secret") char* server_key;
2
3  void main() {
     ...
20   while (...) {
21     accept_connection();
22   }
23 }
24
25 __soaap_sandbox_persistent("session")
26 void accept_connection() {
27   __soaap_private char session_key[1024];
28   compute_session_key(session_key, server_key);
29 }
```

SOAAP can detect data leaking into and out of sandboxes.

```
$ make soaap
*** Sandbox "session" accessed data classified "secret"
*** but does not have necessary clearance to do so.
+++ Line 100 of file session.c

*** Persistent sandbox "session" contains private data
*** that may leak when the sandbox is reused. Consider
*** using an ephemeral sandbox instead or scrub the memory
*** region before control returns.
+++ Line 27 of file session.c
```

Performance

```
1  __soaap_sandbox_ephemeral("decoder")
2  __soaap_sandbox_overhead(12)
3  int jpg_decode(char* buf,
                  __soaap_data_in int len) {
     ...
25 }
```

SOAAP can simulate expected performance and evaluate overhead.

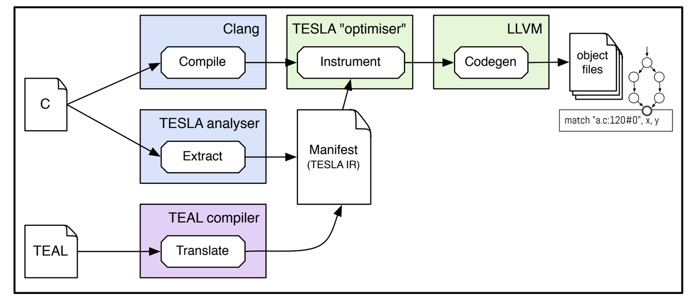
```
$ ./jpg_decode mypic.jpg
*** Sandboxing overhead 69.998658% (Threshold: 12%)
```

http://www.cl.cam.ac.uk/research/security/ctsrd/soaap.html

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.

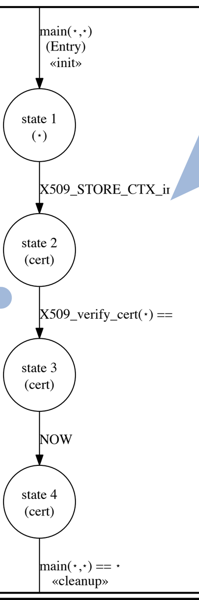


In the example below, the programmer asserts that the X.509 certificate passed to the use_cert function has been properly verified. In fact, however, the certificate verification code suffers from the vulnerability in CVE-2008-5077: an OpenSSL error code has been misinterpreted as success.

```
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 +168,7 @@

 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
 }

@@ -386,7 +408,9 @@
   %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
+  %intrumentation_cert = load %struct.x509_st** %cert.addr, !dbg !782
+  %0 = ptrtoint %struct.x509_st* %intrumentation_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!

http://www.cl.cam.ac.uk/research/security/ctsrd/tesla.html

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)

Many Cambridge members of the CTSRD team meeting for an August 2012 project photo in Cambridge, UK

Jonathan Woodruff, Richard Clayton, Jonathon Anderson, Michael Roe, Ross Anderson, David Chisnall, Robert Watson

Khilan Gudka, Robert Norton, Simon Moore

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.