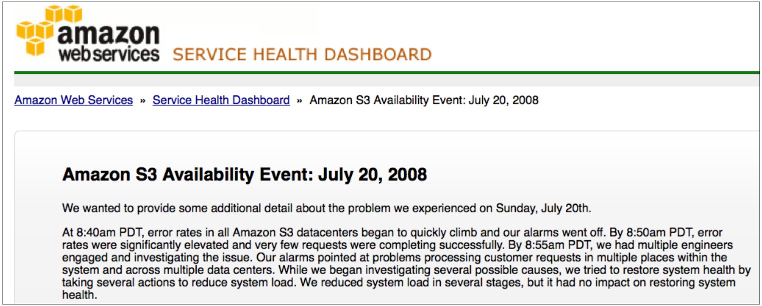
Scalable Error Isolation for Distributed Systems

Diogo Behrens, Sergei Arnautov, Christof Fetzer (TU Dresden)
Marco Serafini (Qatar Computing Research Institute)
Flavio P. Junqueira (Microsoft Research, Cambridge)

May 6, 2015

Motivation

Motivation



SERVICE HEALTH DASHBOARD

Amazon Web Services » Service Health Dashboard » Amazon S3 Availability Event: July 20, 2008

Amazon S3 Availability Event: July 20, 2008

We wanted to provide some additional detail about the problem we experienced on Sunday, July 20th.

At 8:40am PDT, error rates in all Amazon S3 datacenters began to quickly climb and our alarms went off. By 8:50am PDT, error rates were significantly elevated and very few requests were completing successfully. By 8:55am PDT, we had multiple engineers engaged and investigating the issue. Our alarms pointed at problems processing customer requests in multiple places within the system and across multiple data centers. While we began investigating several possible causes, we tried to restore system health by taking several actions to reduce system load. We reduced system load in several stages, but it had no impact on restoring system health.

Motivation

SERVICE HEALTH DASHBOARD

Amazo

Mesa: Geo-Replicated, Near Real-Time, Scalable Data Warehousing

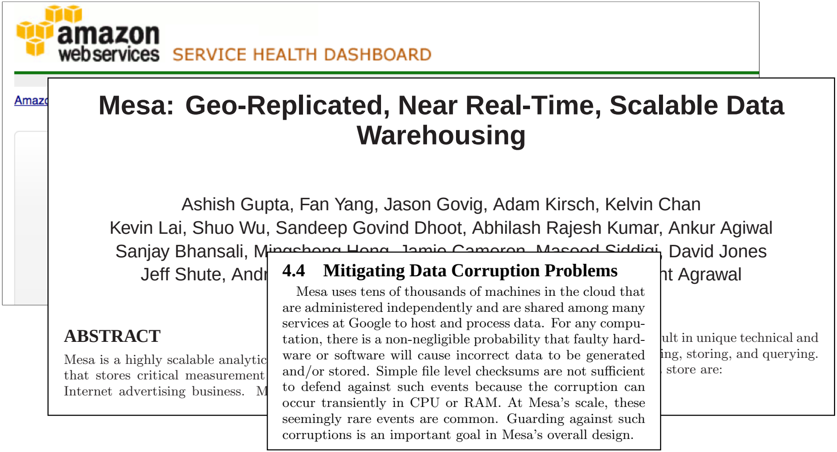
Ashish Gupta, Fan Yang, Jason Govig, Adam Kirsch, Kelvin Chan
Kevin Lai, Shuo Wu, Sandeep Govind Dhoot, Abhilash Rajesh Kumar, Ankur Agiwal
Sanjay Bhansali, Mingsheng Hong, Jamie Cameron, Masood Siddiqi, David Jones
Jeff Shute, Andrey Gubarev, Shivakumar Venkataraman, Divyakant Agrawal
Google, Inc.

ABSTRACT

Mesa is a highly scalable analytic data warehousing system that stores critical measurement data related to Google's Internet advertising business. Mesa is designed to satisfy

ness critical nature of this data result in unique technical and operational challenges for processing, storing, and querying. The requirements for such a data store are:

Motivation



SERVICE HEALTH DASHBOARD

Amazo

Mesa: Geo-Replicated, Near Real-Time, Scalable Data Warehousing

Ashish Gupta, Fan Yang, Jason Govig, Adam Kirsch, Kelvin Chan
Kevin Lai, Shuo Wu, Sandeep Govind Dhoot, Abhilash Rajesh Kumar, Ankur Agiwal
Sanjay Bhansali, Mingsheng Hong, Jamie Cameron, Masood Siddiqi, David Jones
Jeff Shute, Andr
nt Agrawal

ABSTRACT

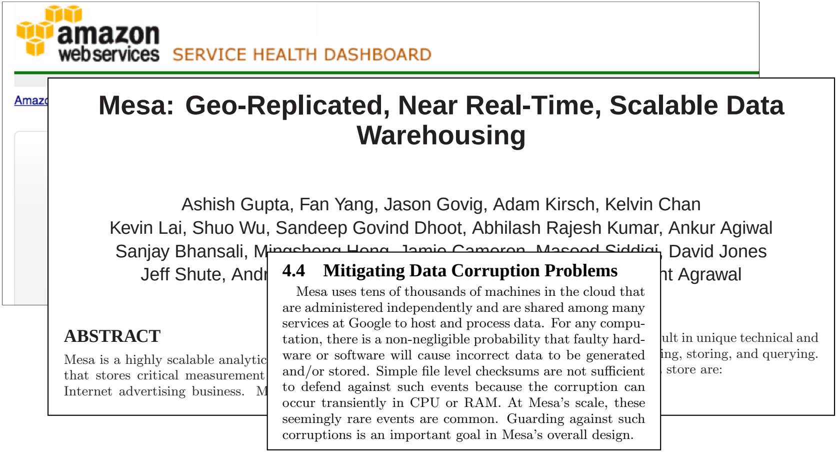
Mesa is a highly scalable analytic
that stores critical measurement
Internet advertising business. M

ult in unique technical and
ing, storing, and querying.
store are:

4.4 Mitigating Data Corruption Problems

Mesa uses tens of thousands of machines in the cloud that are administered independently and are shared among many services at Google to host and process data. For any computation, there is a non-negligible probability that faulty hardware or software will cause incorrect data to be generated and/or stored. Simple file level checksums are not sufficient to defend against such events because the corruption can occur transiently in CPU or RAM. At Mesa's scale, these seemingly rare events are common. Guarding against such corruptions is an important goal in Mesa's overall design.

Motivation



amazon webservices SERVICE HEALTH DASHBOARD

Amazo

Mesa: Geo-Replicated, Near Real-Time, Scalable Data Warehousing

Ashish Gupta, Fan Yang, Jason Govig, Adam Kirsch, Kelvin Chan
Kevin Lai, Shuo Wu, Sandeep Govind Dhoot, Abhilash Rajesh Kumar, Ankur Agiwal
Sanjay Bhansali, Mingsheng Hong, Jamie Cameron, Masood Siddiqi, David Jones
Jeff Shute, Andr nt Agrawal

ABSTRACT

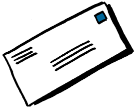
Mesa is a highly scalable analytic
that stores critical measurement
Internet advertising business. M

ult in unique technical and
ing, storing, and querying.
store are:

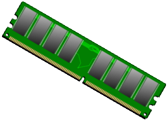
4.4 Mitigating Data Corruption Problems

Mesa uses tens of thousands of machines in the cloud that are administered independently and are shared among many services at Google to host and process data. For any computation, there is a non-negligible probability that faulty hardware or software will cause incorrect data to be generated and/or stored. Simple file level checksums are not sufficient to defend against such events because the corruption can occur transiently in CPU or RAM. At Mesa's scale, these seemingly rare events are common. Guarding against such corruptions is an important goal in Mesa's overall design.

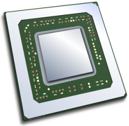
CRC for messages!



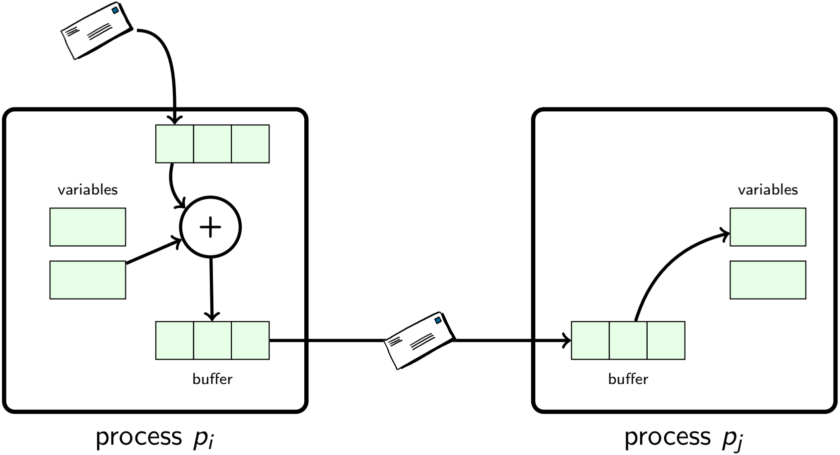
ECC for memory!



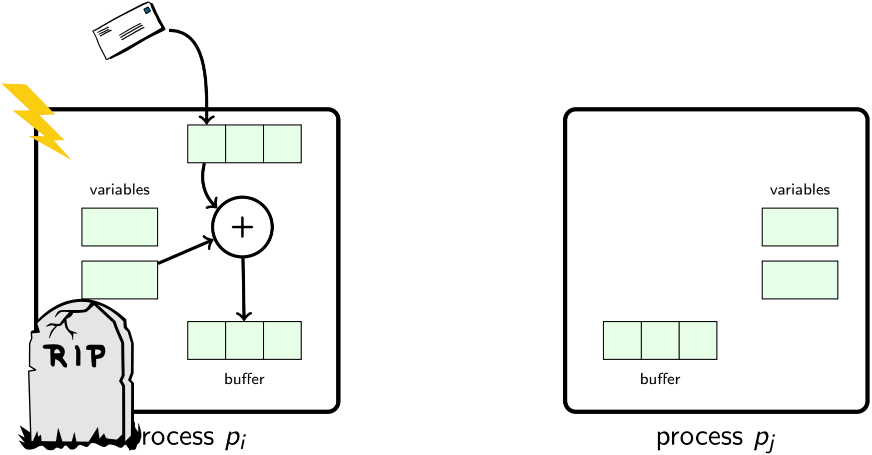
CPU??



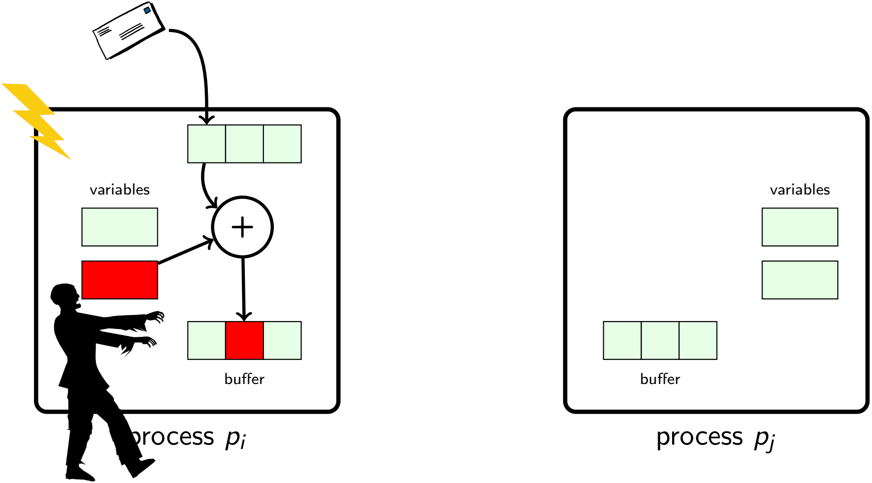
From data corruption to service disruption



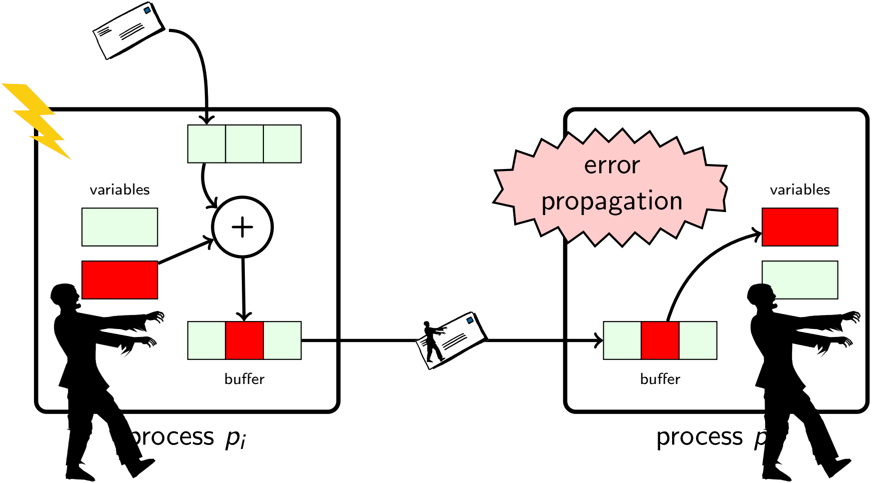
From data corruption to service disruption



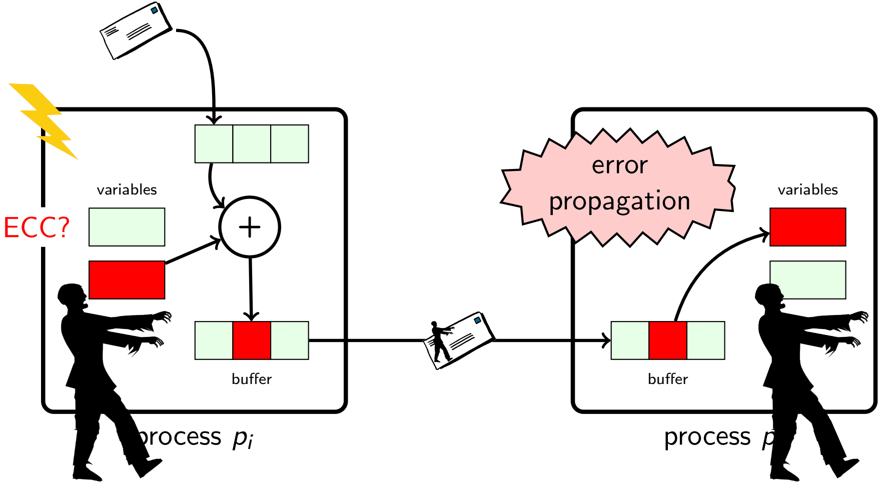
From data corruption to service disruption



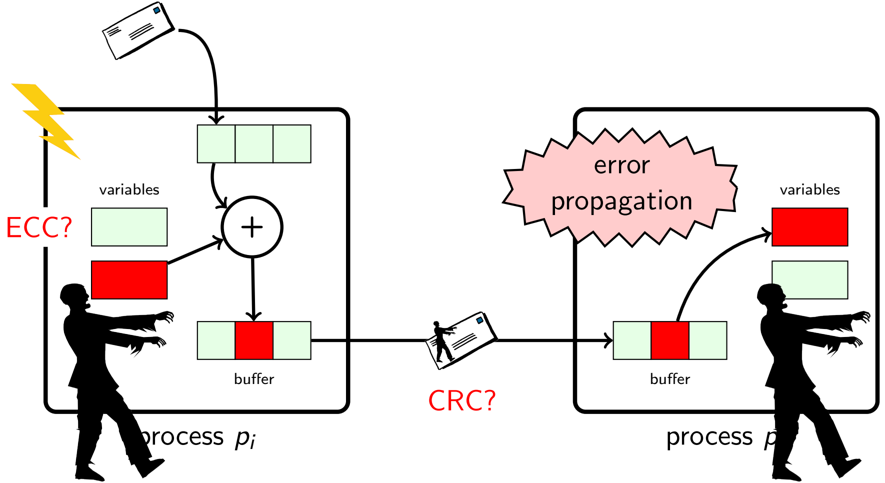
From data corruption to service disruption



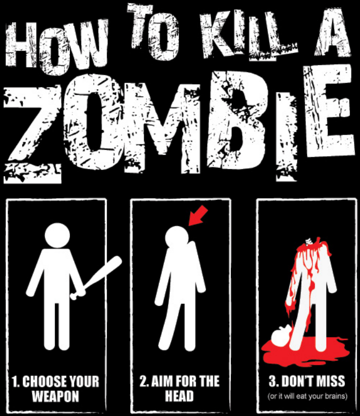
From data corruption to service disruption



From data corruption to service disruption

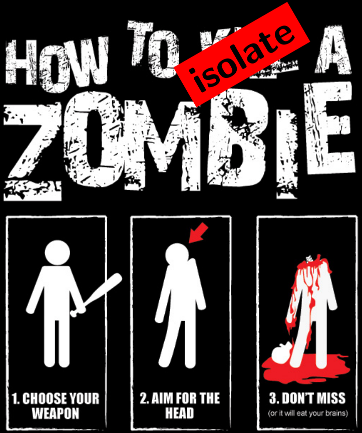


This talk is about. . .



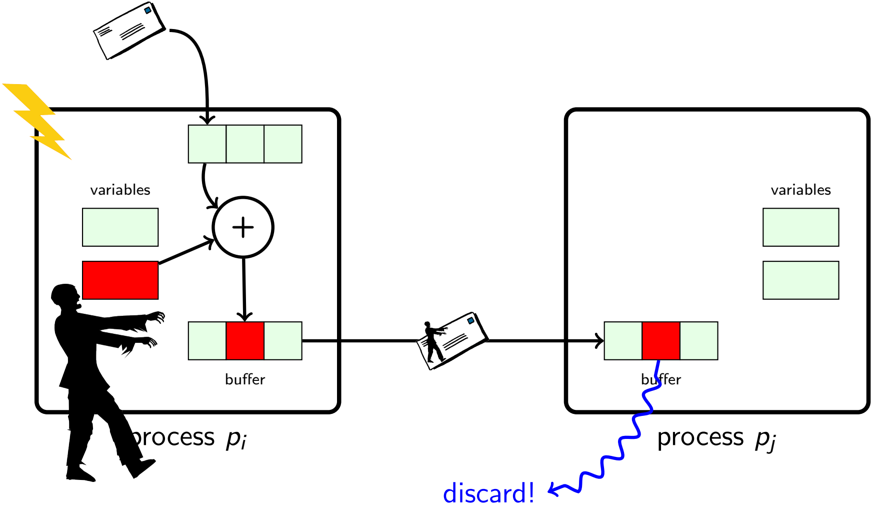
http://micha81.deviantart.com/art/How-To-Kill-a-Zombie-T-Shirt-108157364

This talk is about...

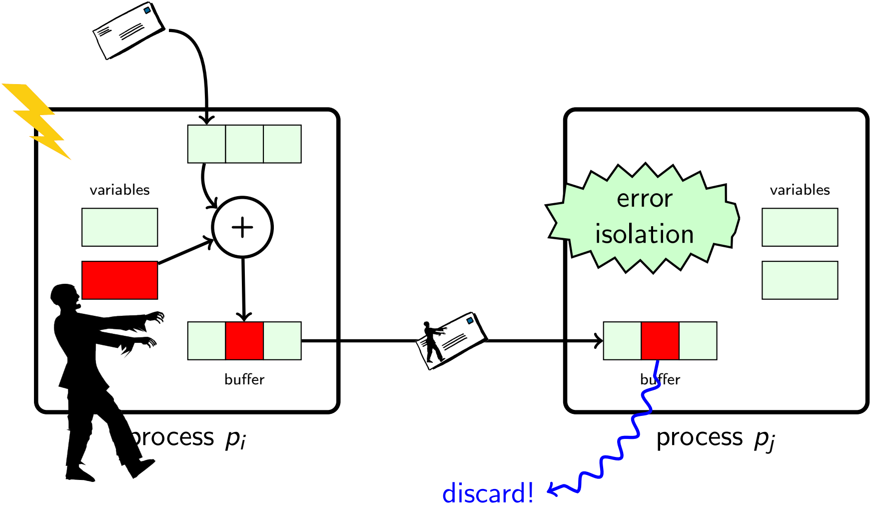


http://micha81.deviantart.com/art/How-To-Kill-a-Zombie-T-Shirt-108157364

Goal: error isolation



Goal: error isolation



How to deal with data corruptions?

DIY: Ad hoc software checks

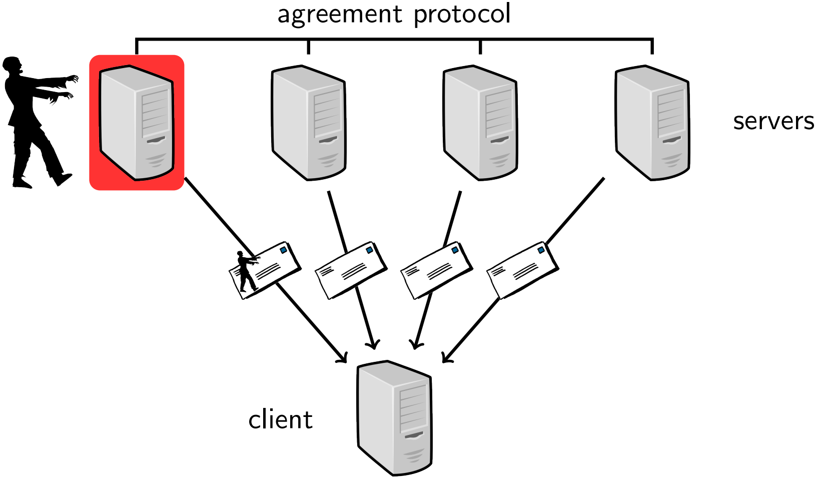
```
while(1) {
    switch(state) {
        case INIT: {
            // create socket, bind and listen
            fd = socket(AF_INET, SOCK_STREAM, 0);
            if (fd < 0) {
                perror("socket");
                return EXIT_FAILURE;
            }

            // this is important, so that if a process restarts, it can
            // quickly reuse the same port
            int on = 1;
            if (setsockopt(fd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on)) < 0) {
                perror("setsockopt");
                return EXIT_FAILURE;
            }

            bzero(&addr, sizeof(addr));
            addr.sin_family      = AF_INET;
            addr.sin_addr.s_addr = htonl(INADDR_ANY);
            addr.sin_port        = htons(port);

            if (bind(fd, (struct sockaddr*) &addr, s     (addr)) < 0) {
                perror("bind");
                return EXIT_FAILURE;
            }

            if (listen(fd, 2) < 0) {
                perror("listen");
                return EXIT_FAILURE;
            }

            // initialize ukv service
            ukv = ukv_init();

            state = ACCT;
            break;
        }
        case RECV: {
            // once a connection is accepted, read      the connection
            // until it is closed
            read = recvfrom(cfd, buffer, BUFSIZE, 0,      t sockaddr*) &caddr, &len);
            if (read <= 0) {
                perror("recvfrom");
                close(cfd);

                state = ACCT;
                break;
            }

            buffer[read] = '\0';
            msg = buffer;

            state = PROC;
            break;
        }
        case PROC: {

            r = ukv_recv(ukv, msg);
            if (!r) state = FINI;
            else state = SEND;
            break;
        }
        case SEND: {
            sendto(cfd, r, strlen(r), 0,
                    (struct sockaddr*) &caddr, sizeof(caddr));
            ukv_done(ukv, r);
            state = RECV;
            break;
        }
```

DIY: Ad hoc software checks

```
while(1) {
    switch(state) {
        case INIT: {
            // create socket, bind and listen
            fd = socket(AF_INET, SOCK_STREAM, 0);
            if (fd < 0) {
                perror("socket");
                return EXIT_FAILURE;
            }

            // this is important, so that if a process restarts, it can
            // quickly reuse the same port
            int on = 1;
            if (setsockopt(fd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on)) < 0) {
                perror("setsockopt");
                return EXIT_FAILURE;
            }

            bzero(&addr, sizeof(addr));
            addr.sin_family      = AF_INET;
            addr.sin_addr.s_addr = htonl(INADDR_ANY);
            addr.sin_port        = htons(port);

            if (bind(fd, (struct sockaddr*) &addr, s     (addr)) < 0) {
                perror("bind");
                return EXIT_FAILURE;
            }

            if (listen(fd, 2) < 0) {
                perror("listen");
                return EXIT_FAILURE;
            }

            // initialize ukv service
            ukv = ukv_init();

            state = ACCT;
            break;
        }
        case RECV: {
            // once a connection is accepted, read      the connection
            // until it is closed
            read = recvfrom(cfd, buffer, BUFSIZE, 0,      t sockaddr*) &caddr, &len);
            if (read <= 0) {
                perror("recvfrom");
                close(cfd);

                state = ACCT;
                break;
            }

            buffer[read] = '\0';
            msg = buffer;

            state = PROC;
            break;
        }
        case PROC: {

            r = ukv_recv(ukv, msg);
            if (!r) state = FINI;
            else state = SEND;
            break;
        }
        case SEND: {
            sendto(cfd, r, strlen(r), 0,
                   (struct sockaddr*) &caddr, sizeof(caddr));
            ukv_done(ukv, r);
            state = RECV;
            break;
        }
```

- Complex, time consuming
- Which errors to consider?
- No principled approach

Principled approach: Byzantine fault tolerance

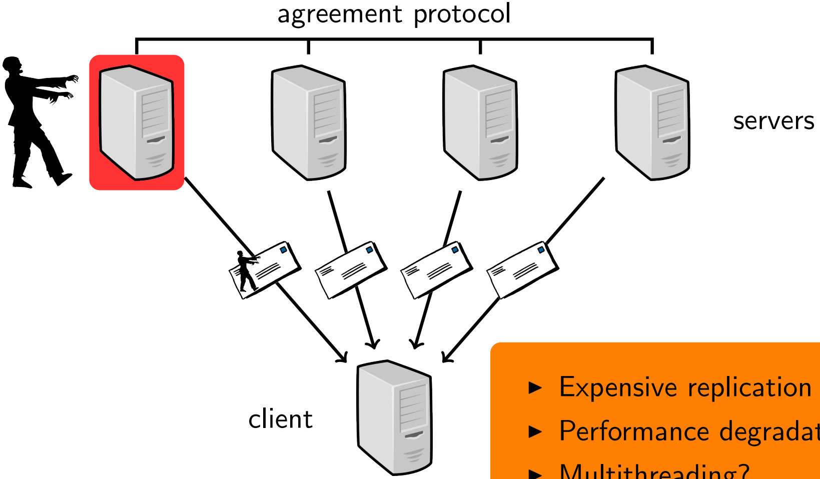


agreement protocol

servers

client

Principled approach: Byzantine fault tolerance



- Expensive replication
- Performance degradation
- Multithreading?

Principled approach: Local hardening

Principled approach: Local hardening

- **Instruction duplication (SWIFT, CGO’05)**
  - \+ compiler technique
  - − not designed for distributed systems
  - − last-mile faults (no error isolation)

Principled approach: Local hardening

- **Instruction duplication (SWIFT, CGO'05)**
  - \+ compiler technique
  - – not designed for distributed systems
  - – last-mile faults (no error isolation)
- **PASC (ATC'12)**
  - \+ achieves error isolation
  - – large memory overhead (2x)
  - – no support for multithreading

Scalable Error Isolation

A new approach

Scalable Error Isolation (SEI)

Local hardening

No additional messages exchanged;
Local redundancy in space and time

Scalable Error Isolation (SEI)

Local hardening

No additional messages exchanged;
Local redundancy in space and time

End-to-end

CRCs for communication and **computation**

Scalable Error Isolation (SEI)

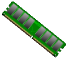
Local hardening
No additional messages exchanged;
Local redundancy in space and time

End-to-end
CRCs for communication and **computation**

Formal guarantees
Fault model and correctness proof

Scalability dimensions

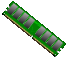
Scalability dimensions



Memory scalability

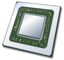
Small footprint (ECC or other error codes)

Scalability dimensions



Memory scalability

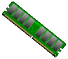
Small footprint (ECC or other error codes)



Thread scalability

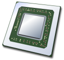
Support for multithreaded applications

Scalability dimensions



Memory scalability

Small footprint (ECC or other error codes)



Thread scalability

Support for multithreaded applications

`main()`

Codebase scalability

Compiler technique reduces developer work

Outline

Overview of SEI

Requirements, fault model, and algorithm

Challenges

Support for multithreaded applications

`foo()`

Implementation: `libsei`

Library for C-based programs

Evaluation

Fault coverage and performance overhead

Outline

Overview of SEI
Requirements, fault model, and algorithm

Challenges
Support for multithreaded applications

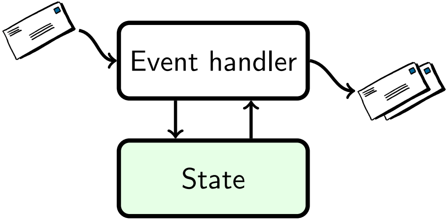
`foo()`

Implementation: `libsei`
Library for C-based programs

Evaluation
Fault coverage and performance overhead

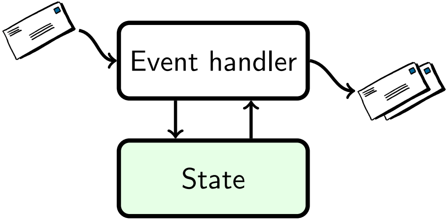
Target applications

- **Event based – message passing**



Target applications

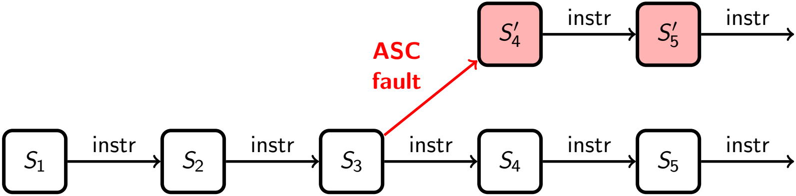
- **Event based – message passing**



- **Multithreaded applications**
  - Critical sections to access shared variables
  - Hierarchical locking (consistent order) to avoid deadlocks

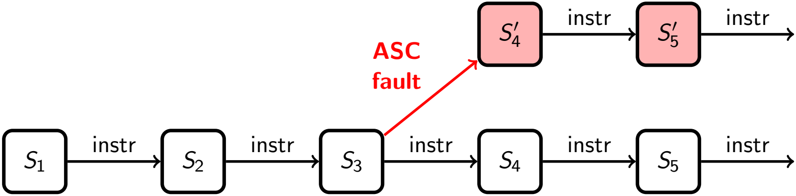
Arbitrary State Corruption (ASC) Model

Faults corrupt any number of variables



Arbitrary State Corruption (ASC) Model

Faults corrupt any number of variables

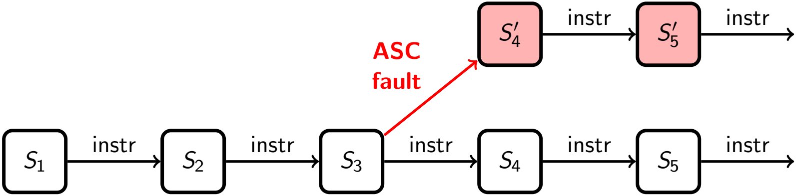


Assumptions:

- **Fault frequency**:
  at most one 1 fault per
  event handler

Arbitrary State Corruption (ASC) Model

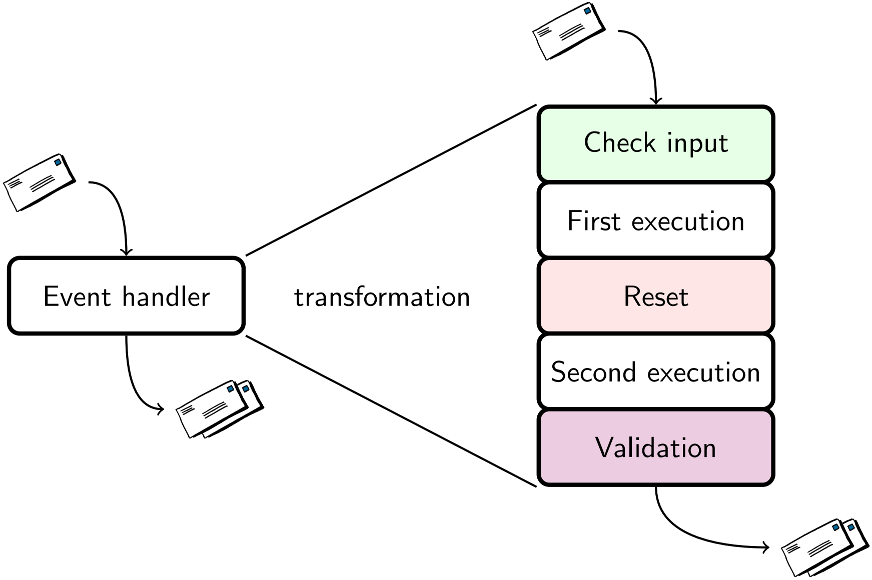
Faults corrupt any number of variables



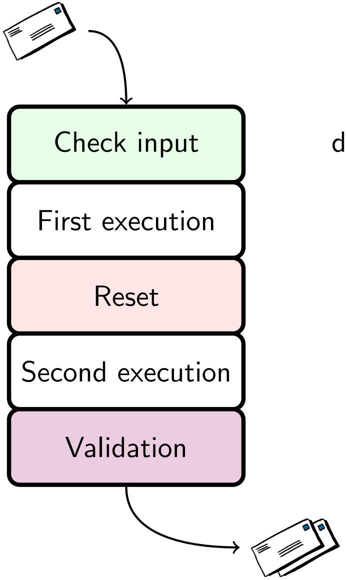
Assumptions:

- **Fault frequency**: at most one 1 fault per event handler
- **Corruption coverage**: detection codes work, e.g., ECC, CRC

Hardening with SEI



Hardening with SEI



Check input

First execution

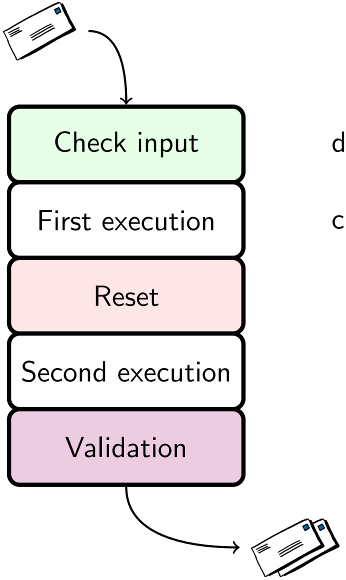
Reset

Second execution

Validation

discard message if CRC invalid

Hardening with SEI



Check input

First execution

Reset

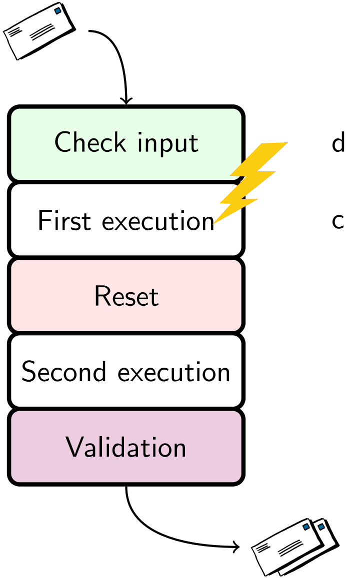
Second execution

Validation

discard message if CRC invalid

check read variables

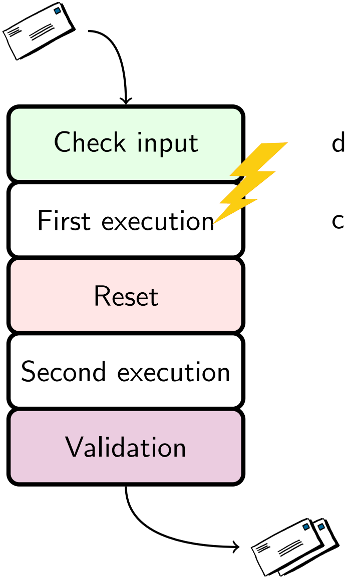
Hardening with SEI



discard message if CRC invalid

check read variables

Hardening with SEI



Check input

discard message if CRC invalid

First execution

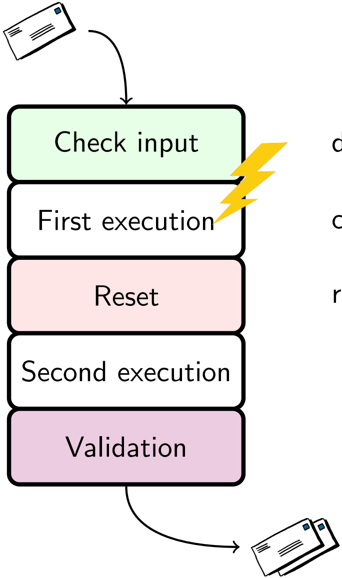
check read variables, snapshot original state

Reset

Second execution

Validation

Hardening with SEI



Check input

First execution

Reset

Second execution

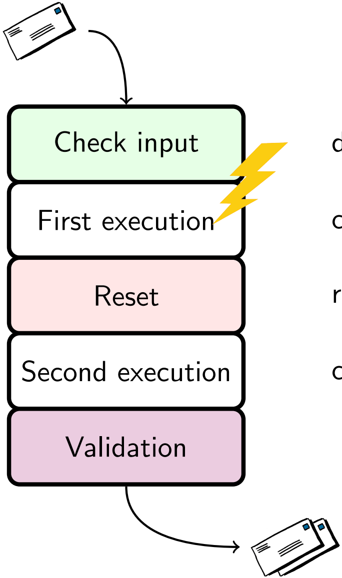
Validation

discard message if CRC invalid

check read variables, snapshot original state

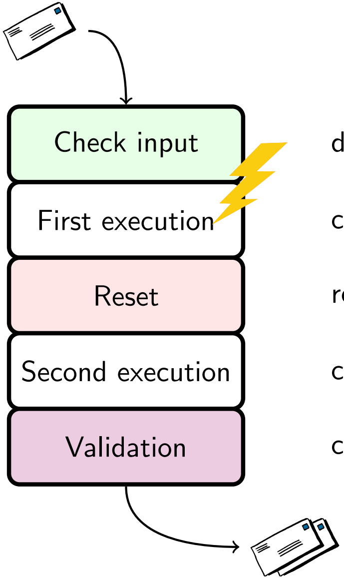
restore snapshot, record state modifications

Hardening with SEI



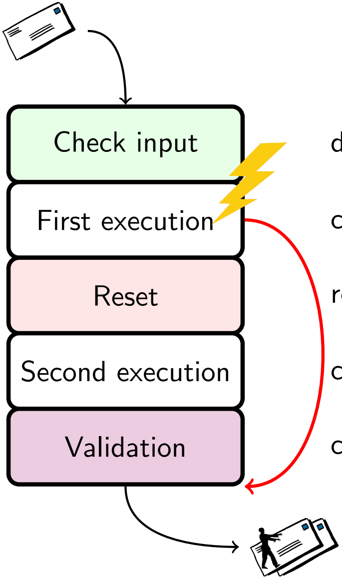
Check input	discard message if CRC invalid
First execution	check read variables, snapshot original state
Reset	restore snapshot, record state modifications
Second execution	check read variables, record modified variables
Validation	

Hardening with SEI



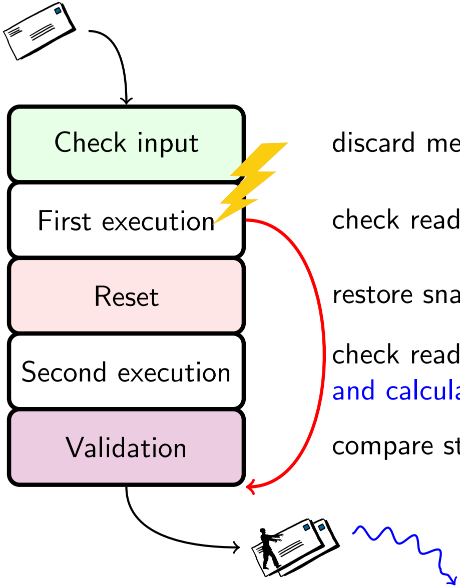
Check input	discard message if CRC invalid
First execution	check read variables, snapshot original state
Reset	restore snapshot, record state modifications
Second execution	check read variables, record modified variables
Validation	compare state modifications

Hardening with SEI



Check input	discard message if CRC invalid
First execution	check read variables, snapshot original state
Reset	restore snapshot, record state modifications
Second execution	check read variables, record modified variables
Validation	compare state modifications

Hardening with SEI



Check input	discard message if CRC invalid
First execution	check read variables, snapshot original state
Reset	restore snapshot, record state modifications
Second execution	check read variables, record modified variables, and calculates CRC
Validation	compare state modifications

invalid CRC, discard!

Outline

Overview of SEI

Requirements, fault model, and algorithm

Challenges

Support for multithreaded applications

`foo()`

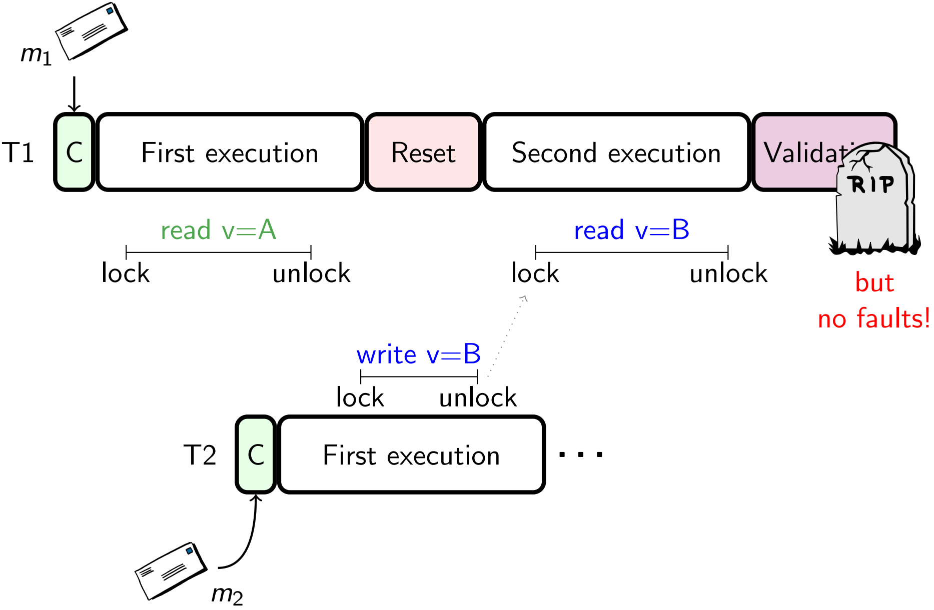
Implementation: `libsei`

Library for C-based programs

Evaluation

Fault coverage and performance overhead

Deterministic event handling with multithreading



m_1

T1 C First execution Reset Second execution Validati

read v=A

lock unlock

read v=B

lock unlock

but
no faults!

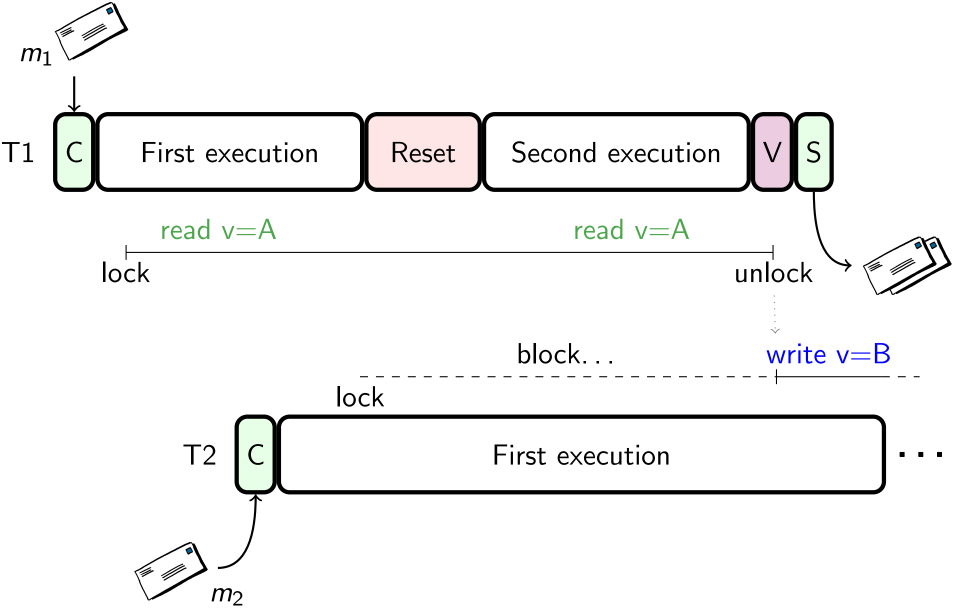
write v=B

lock unlock

T2 C First execution . . .

m_2

Deterministic event handling with multithreading



m_1

T1

C

First execution

Reset

Second execution

V

S

read v=A

read v=A

lock

unlock

block...

write v=B

lock

T2

C

First execution

...

m_2

Outline

Overview of SEI

Requirements, fault model, and algorithm

Challenges

Support for multithreaded applications

`foo()`

Implementation: `libsei`

Library for C-based programs

Evaluation

Fault coverage and performance overhead

libsei: SEI implementation in C

bitbucket.org/db7/libsei

```
while(1) {
  ilen = recv_msg(imsg, &crc);

    do_something_here(msg);
    omsg = create_message(&olen);




  send_msg(omsg, olen, CRC(omsg, olen));
}
```

- **Annotate event handler**
  with `__begin` and `__end`
- **Annotate messages**
  `__output*` for annotation
  `__crc_pop` for next CRC

libsei: SEI implementation in C

bitbucket.org/db7/libsei

```
while(1) {
  ilen = recv_msg(imsg, &crc);
  if (__begin(imsg, ilen, crc)) {
    do_something_here(msg);
    omsg = create_message(&olen);


    __end();        // end of handler
  } else continue; //discard invalid

  send_msg(omsg, olen, CRC(omsg, olen));
}
```

- **Annotate event handler**
  with `__begin` and `__end`
- **Annotate messages**
  `__output*` for annotation
  `__crc_pop` for next CRC

libsei: SEI implementation in C

`bitbucket.org/db7/libsei`

```
while(1) {
  ilen = recv_msg(imsg, &crc);
  if (__begin(imsg, ilen, crc)) {
    do_something_here(msg);
    omsg = create_message(&olen);
    __output_append(omsg, olen);
    __output_done();
    __end();        // end of handler
  } else continue; //discard invalid

  send_msg(omsg, olen, CRC(omsg, olen));
}
```

- **Annotate event handler** with `__begin` and `__end`
- **Annotate messages** `__output*` for annotation `__crc_pop` for next CRC

`libsei`: SEI implementation in C

`bitbucket.org/db7/libsei`

```
while(1) {
  ilen = recv_msg(imsg, &crc);
  if (__begin(imsg, ilen, crc)) {
    do_something_here(msg);
    omsg = create_message(&olen);
    __output_append(omsg, olen);
    __output_done();
    __end();        // end of handler
  } else continue; //discard invalid

  send_msg(omsg, olen, __crc_pop());
}
```

- **Annotate event handler**
  with `__begin` and `__end`
- **Annotate messages**
  `__output*` for annotation
  `__crc_pop` for next CRC

`libsei`: SEI implementation in C

`bitbucket.org/db7/libsei`

```
while(1) {
  ilen = recv_msg(imsg, &crc);
  if (__begin(imsg, ilen, crc)) {
    do_something_here(msg);
    omsg = create_message(&olen);
    __output_append(omsg, olen);
    __output_done();
    __end();       // end of handler
  } else continue; //discard invalid

  send_msg(omsg, olen, __crc_pop());
}
```

- **Annotate event handler**
  with `__begin` and `__end`
- **Annotate messages**
  `__output*` for annotation
  `__crc_pop` for next CRC
- **Compile with GCC $\geq$ 4.7**
  state updates within handler
  instrumented with TM pass

`libsei`: SEI implementation in C

`bitbucket.org/db7/libsei`

```
while(1) {
  ilen = recv_msg(imsg, &crc);
  if (__begin(imsg, ilen, crc)) {
    do_something_here(msg);
    omsg = create_message(&olen);
    __output_append(omsg, olen);
    __output_done();
    __end();        // end of handler
  } else continue; //discard invalid

  send_msg(omsg, olen, __crc_pop());
}
```

- **Annotate event handler**
  with `__begin` and `__end`
- **Annotate messages**
  `__output*` for annotation
  `__crc_pop` for next CRC
- **Compile with GCC ≥ 4.7**
  state updates within handler instrumented with TM pass
- `libsei` **does the rest**
  - executes handler twice
  - calculates CRCs
  - validates state updates

Outline

Overview of SEI
Requirements, fault model, and algorithm

Challenges
Support for multithreaded applications

`foo()`

Implementation: `libsei`
Library for C-based programs

Evaluation
Fault coverage and performance overhead

Use cases

- `memcached`
  - Key-value store
  - Widely used as cache for databases
  - Internally, a huge hash table with eviction queues
  - Multithreaded

Use cases

- `memcached`
    - Key-value store
    - Widely used as cache for databases
    - Internally, a huge hash table with eviction queues
    - Multithreaded
- Deadwood
    - DNS recursive server
    - Single-threaded
    - See paper for results

Fault coverage: targeted software fault injection

Fault coverage: targeted software fault injection

Fault group	**Variant**	**Undetected**	**SEI-detected**	**Crash/other**
Control flow	native	9.66%	-	90.34%
	SEI	0.06%	14.70%	85.23%
Data flow	native	44.18%	-	55.82%
	SEI	0.15%	57.55%	42.29%

Fault coverage: targeted software fault injection

Fault group	**Variant**	**Undetected**	**SEI-detected**	**Crash/other**
Control flow	native	9.66%	-	90.34%
	SEI	0.06%	14.70%	85.23%
Data flow	native	44.18%	-	55.82%
	SEI	0.15%	57.55%	42.29%

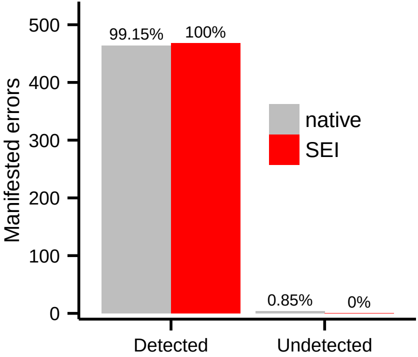
Fault coverage: targeted software fault injection

Fault group	**Variant**	**Undetected**	**SEI-detected**	**Crash/other**
Control flow	native	9.66%	-	90.34%
	SEI	0.06%	14.70%	85.23%
Data flow	native	44.18%	-	55.82%
	SEI	0.15%	57.55%	42.29%

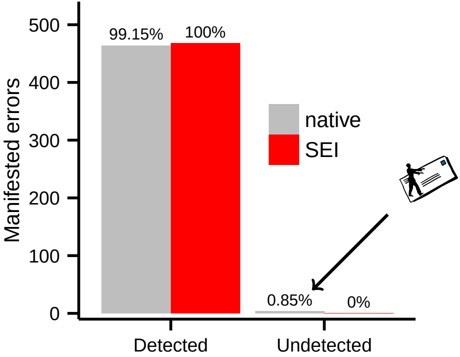
SEI: two orders of magnitude fewer undetected errors

Fault coverage: CPU undervolting

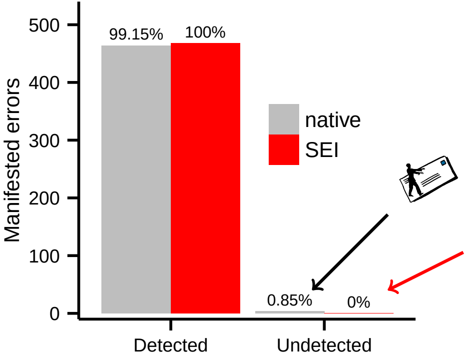
Fault coverage: CPU undervolting



Fault coverage: CPU undervolting



Fault coverage: CPU undervolting

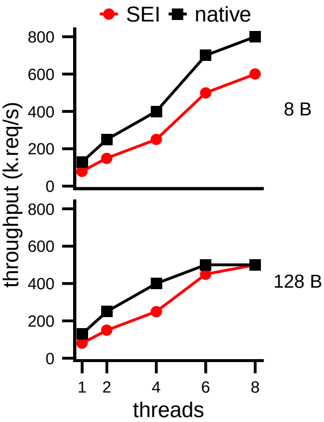


error isolation

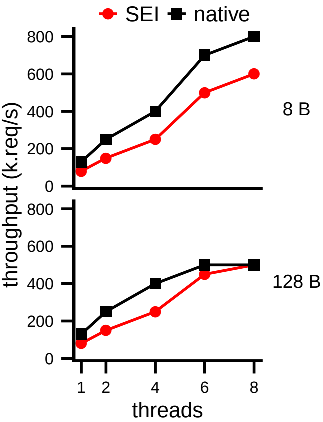
SEI: no undetected errors

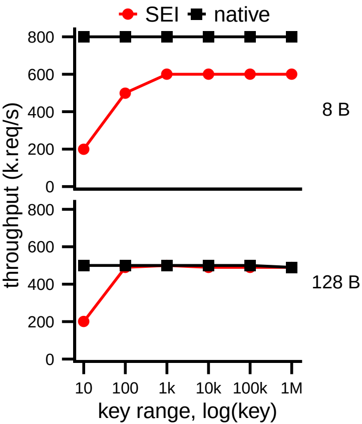
`memcached` performance: threads and key range

`memcached` performance: threads and key range

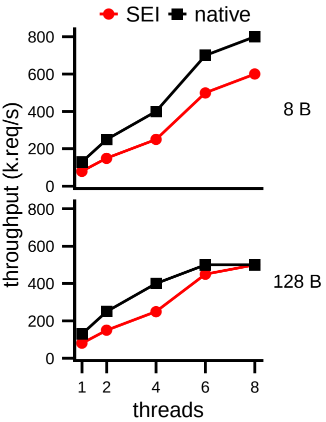


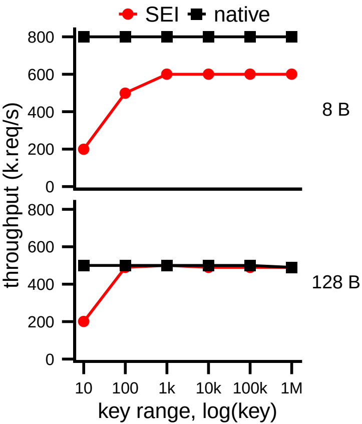
`memcached` performance: threads and key range





`memcached` performance: threads and key range





SEI: little overhead with $\geq$ 128 B and ranges of $\geq$ 100 keys

Conclusion

- **Algorithm: Scalable Error Isolation (SEI)**
  - Local and end-to-end
  - Effective against data corruptions
- **Implementation:** `libsei`
  - No memory overhead with ECC
  - Little performance overhead with non-CPU intensive applications
  - Implementation is open source

Thank you! Questions?

Source code and technical report:
`http://bitbucket.org/db7/libsei`