PoWER Never Corrupts: Tool-Agnostic Verification of Crash Consistency and Corruption Detection

Hayley LeBlanc, Jacob R. Lorch, Chris Hawblitzel, Cheng Huang, Yiheng Tao, Nickolai Zeldovich, Vijay Chidambaram

🏆 ***Distinguished Artifact Award***

TEXAS The University of Texas at Austin
Microsoft
MIT CSAIL

Motivation: interest from Azure Storage in a **verified persistent memory key-value store**

Motivation: interest from Azure Storage in a verified **persistent memory key-value store**

Motivation: interest from Azure Storage in a **verified** **persistent memory** **key-value store**

Reasoning about persistent memory

Reasoning about persistent memory

Low-latency, byte-addressable storage

Reasoning about persistent memory

Low-latency, byte-addressable storage

Intel Optane DC PMM or battery-backed DRAM

Reasoning about persistent memory

Low-latency, byte-addressable storage

Intel Optane DC PMM or battery-backed DRAM

No earlier verified PM systems

Reasoning about persistent memory

Low-latency, byte-addressable storage

Intel Optane DC PMM or battery-backed DRAM

No earlier verified PM systems

Crash consistency and corruption detection on PM are hard!

- Small (8-byte) atomic writes
- Interaction between crashes and corruption

Reasoning about persistent memory

Low-latency, byte-addressable storage

Intel Optane DC PMM or battery-backed DRAM

No earlier verified PM systems

Crash consistency and corruption detection on PM are hard!

- Small (8-byte) atomic writes
- Interaction between crashes and corruption

Goal 1: new techniques to verify PM systems and beyond

Building a practical verified system

Building a practical verified system

Needs to be integrated with an unverified Rust codebase

Building a practical verified system

Needs to be integrated with an unverified Rust codebase

Target verification tool: **Verus** (Lattuada OOPSLA ‘23, SOSP ‘24)

Building a practical verified system

Needs to be integrated with an unverified Rust codebase

Target verification tool: **Verus** (Lattuada OOPSLA '23, SOSP '24)

Verification tool	Easy integration with Rust?	Targets low-level systems?	Fast verification times?
Verus	✅	✅	✅

Building a practical verified system

Needs to be integrated with an unverified Rust codebase

Target verification tool: **Verus** (Lattuada OOPSLA '23, SOSP '24)

Verification tool	Easy integration with Rust?	Targets low-level systems?	Fast verification times?
Verus	✅	✅	✅
Rocq/Coq (FSCQ)	❌	❌	❌
Perennial	❌	✅	❌

Building a practical verified system

Needs to be integrated with an unverified Rust codebase

Target verification tool: **Verus** (Lattuada OOPSLA '23, SOSP '24)

Verification tool	Easy integration with Rust?	Targets low-level systems?	Fast verification times?	Built-in crash safety reasoning?
Verus	✅	✅	✅	❌
Rocq/Coq (FSCQ)	❌	❌	❌	✅
Perennial	❌	✅	❌	✅

Building a practical verified system

Needs to be integrated with an unverified Rust codebase

Target verification tool: **Verus** (Lattuada OOPSLA '23, SOSP '24)

Verification tool	Easy integration with Rust?	Targets low-level systems?	Fast verification times?	Built-in crash safety reasoning?
Verus	✅	✅	✅	❌
Rocq/Coq (FSCQ)	❌	❌	❌	✅
Perennial	❌	✅	❌	✅

Goal 2: verify crash consistency without built-in language support

Building a practical verified system

Needs to be integrated with an unverified Rust codebase

Target verification tool: **Verus** (Lattuada OOPSLA ‘23, SOSP ‘24)

Verification tool	Easy integration with Rust?	Targets low-level systems?	Fast verification times?	Built-in crash safety reasoning?
Verus	✅	✅	✅	❌
Rocq/Coq (FSCQ)	❌	❌	❌	✅
Perennial	❌	✅	❌	✅

Goal 2: verify crash consistency without built-in language support

i.e., *tool-agnostic*

Benefits of a tool-agnostic technique

Compatible with nearly all current verification tools

Developers can choose a tool best suited to their system

New storage systems can take advantage of powerful new verification tools

Contributions

Contributions

First verified PM storage systems: **CapybaraKV** (Verus) and CapybaraNS (Dafny)

Contributions

First verified PM storage systems: **CapybaraKV** (Verus) and CapybaraNS (Dafny)

CapybaraKV (~25KLOC) verifies in <1 minute and achieves performance competitive with *unverified* PM KV stores

Contributions

First verified PM storage systems: **CapybaraKV** (Verus) and CapybaraNS (Dafny)

CapybaraKV (~25KLOC) verifies in <1 minute and achieves performance competitive with *unverified* PM KV stores

PoWER: crash-consistency verification approach compatible with nearly all verification tools

Contributions

First verified PM storage systems: **CapybaraKV** (Verus) and CapybaraNS (Dafny)

CapybaraKV (~25KLOC) verifies in <1 minute and achieves performance competitive with *unverified* PM KV stores

PoWER: crash-consistency verification approach compatible with nearly all verification tools

Corruption-detecting Boolean: primitive for atomic checksum updates

Contributions

First verified PM storage systems: **CapybaraKV** (Verus) and CapybaraNS (Dafny)

CapybaraKV (~25KLOC) verifies in <1 minute and achieves performance competitive with *unverified* PM KV stores

PoWER: crash-consistency verification approach compatible with nearly all verification tools

Corruption-detecting Boolean: primitive for atomic checksum updates

github.com/microsoft/verified-storage

Setting up the problem

Setting up the problem

Main robustness properties to verify:

Setting up the problem

Main robustness properties to verify:

1. Crash consistency

Setting up the problem

Main robustness properties to verify:

1. Crash consistency
2. Data corruption detection

Setting up the problem

Main robustness properties to verify:

1. Crash consistency
2. Data corruption detection

Storage systems use *cyclic redundancy checks* (CRCs) to detect corruption

Setting up the problem

Main robustness properties to verify:

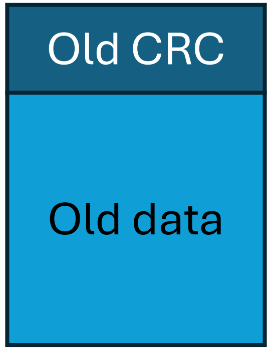
1. Crash consistency
2. Data corruption detection

Storage systems use *cyclic redundancy checks* (CRCs) to detect corruption

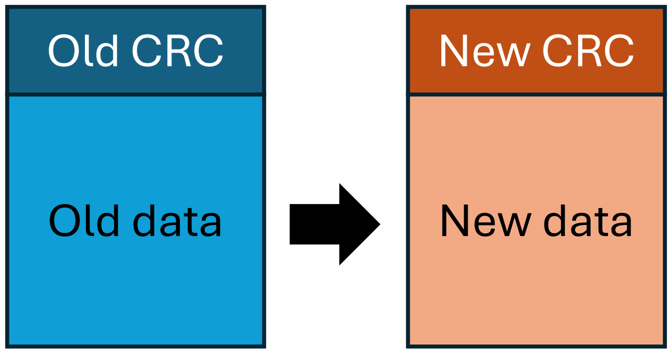
Challenging interaction with crashes!

Running example

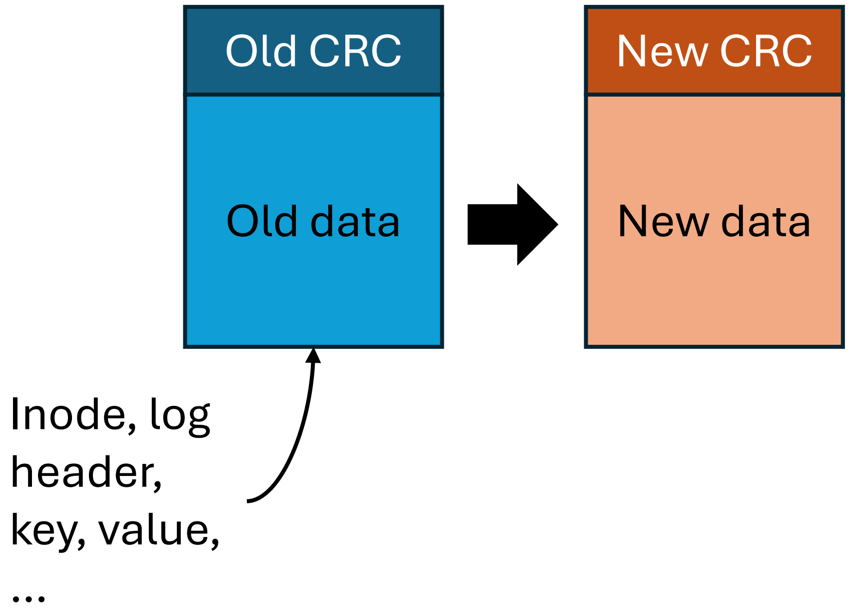
Running example



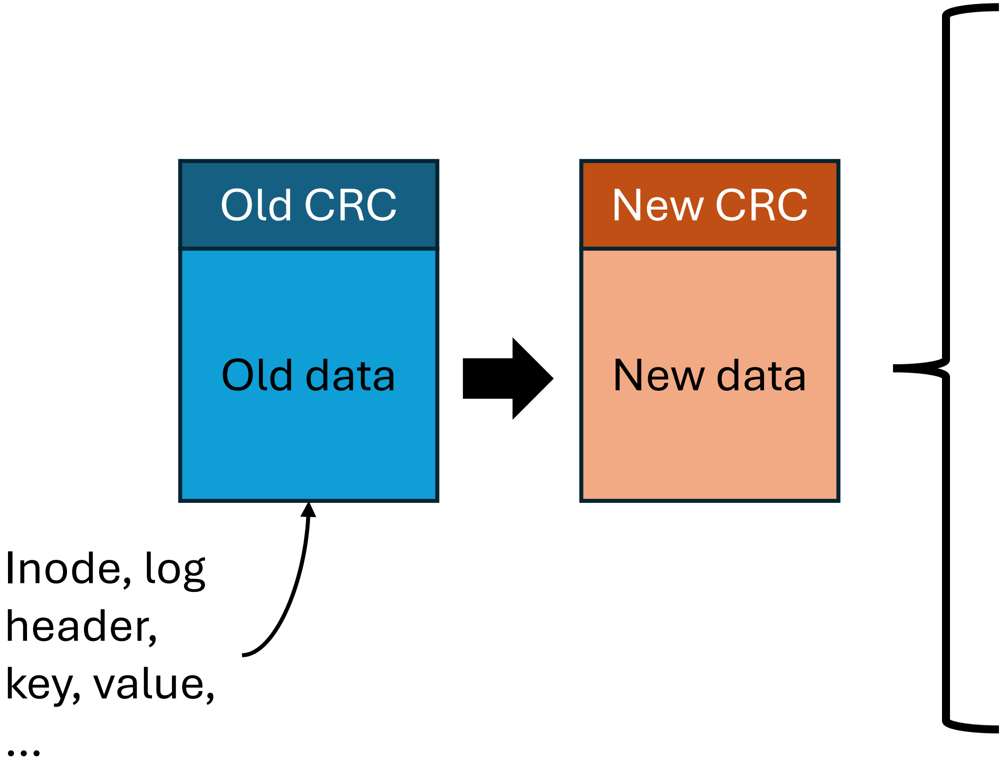
Running example



Running example

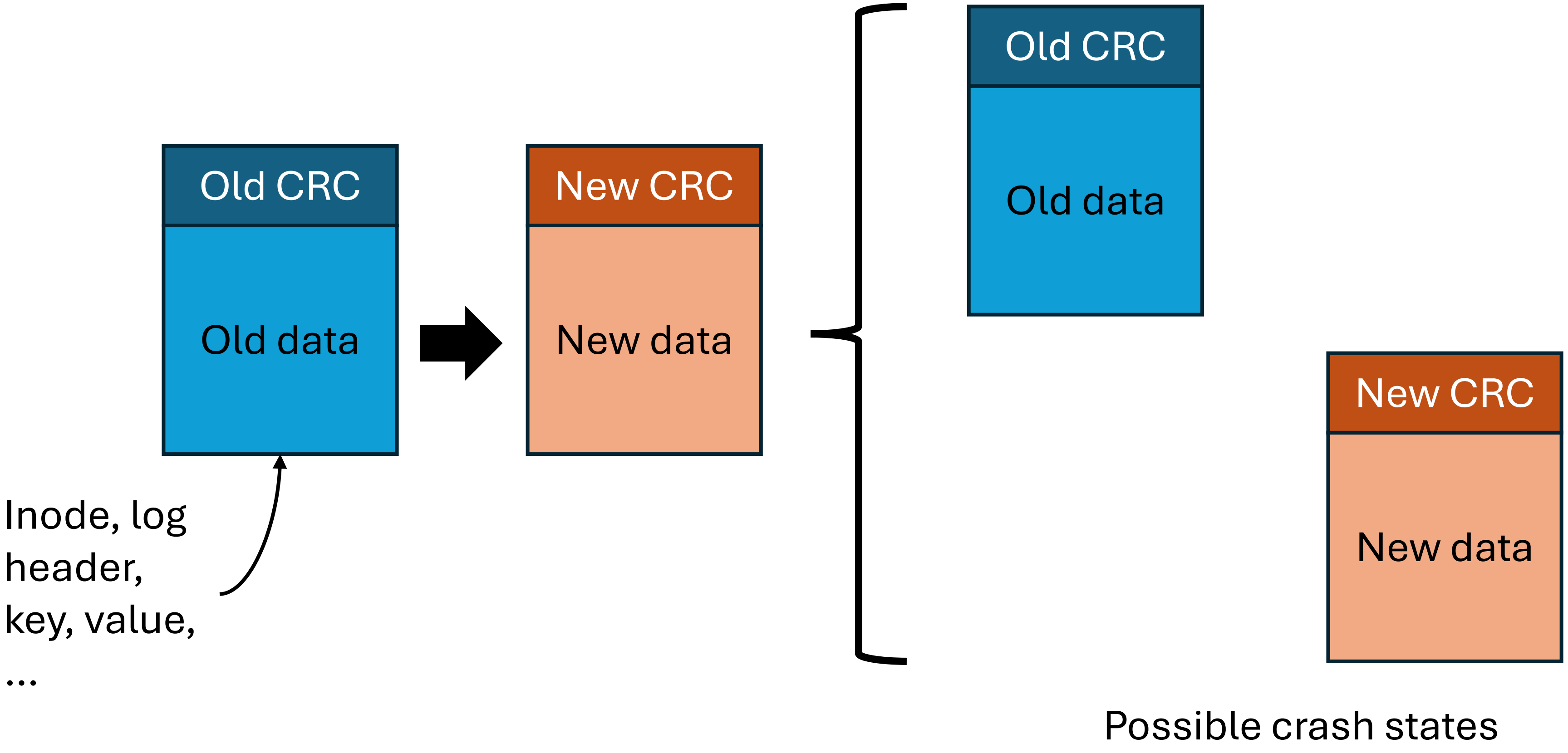


Running example

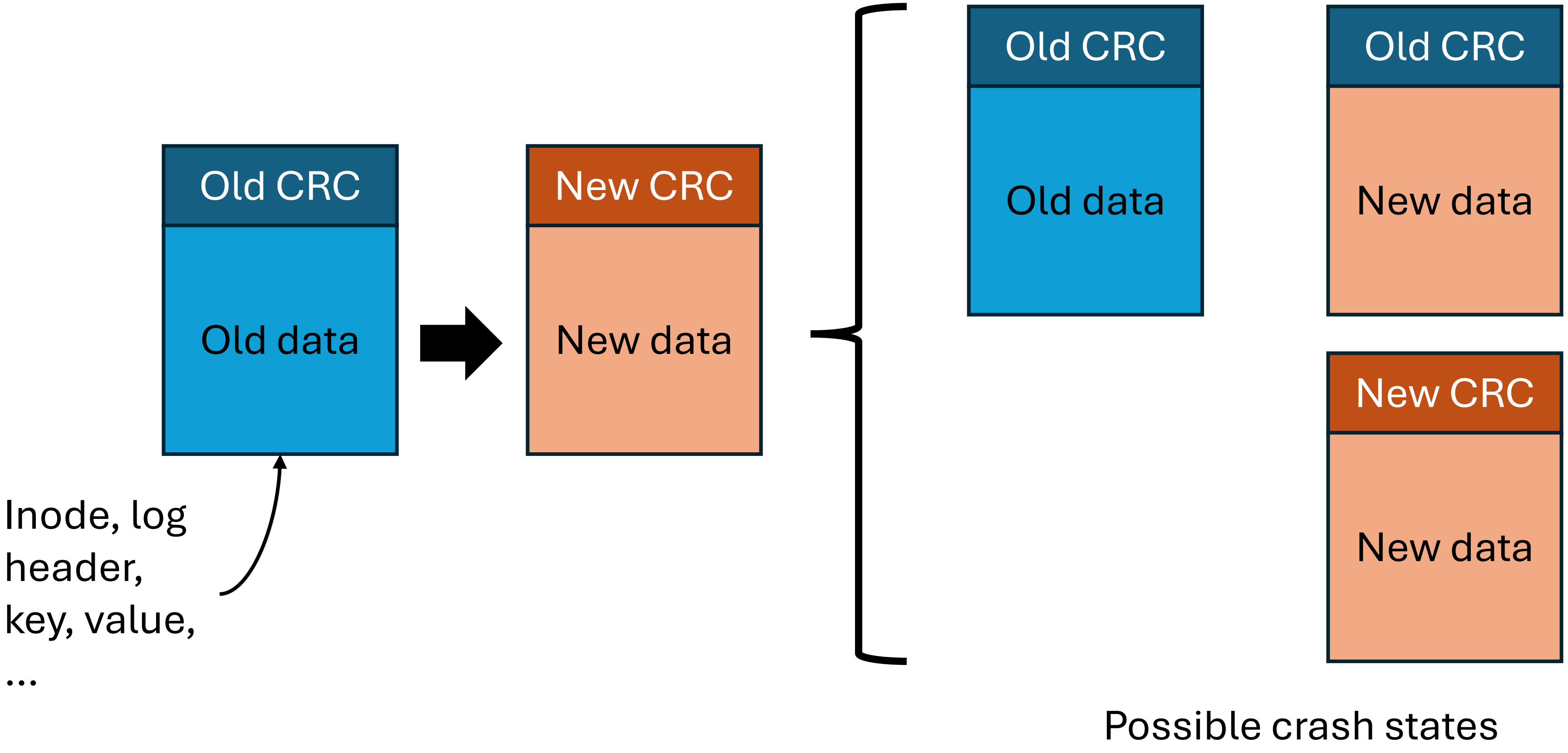


Possible crash states

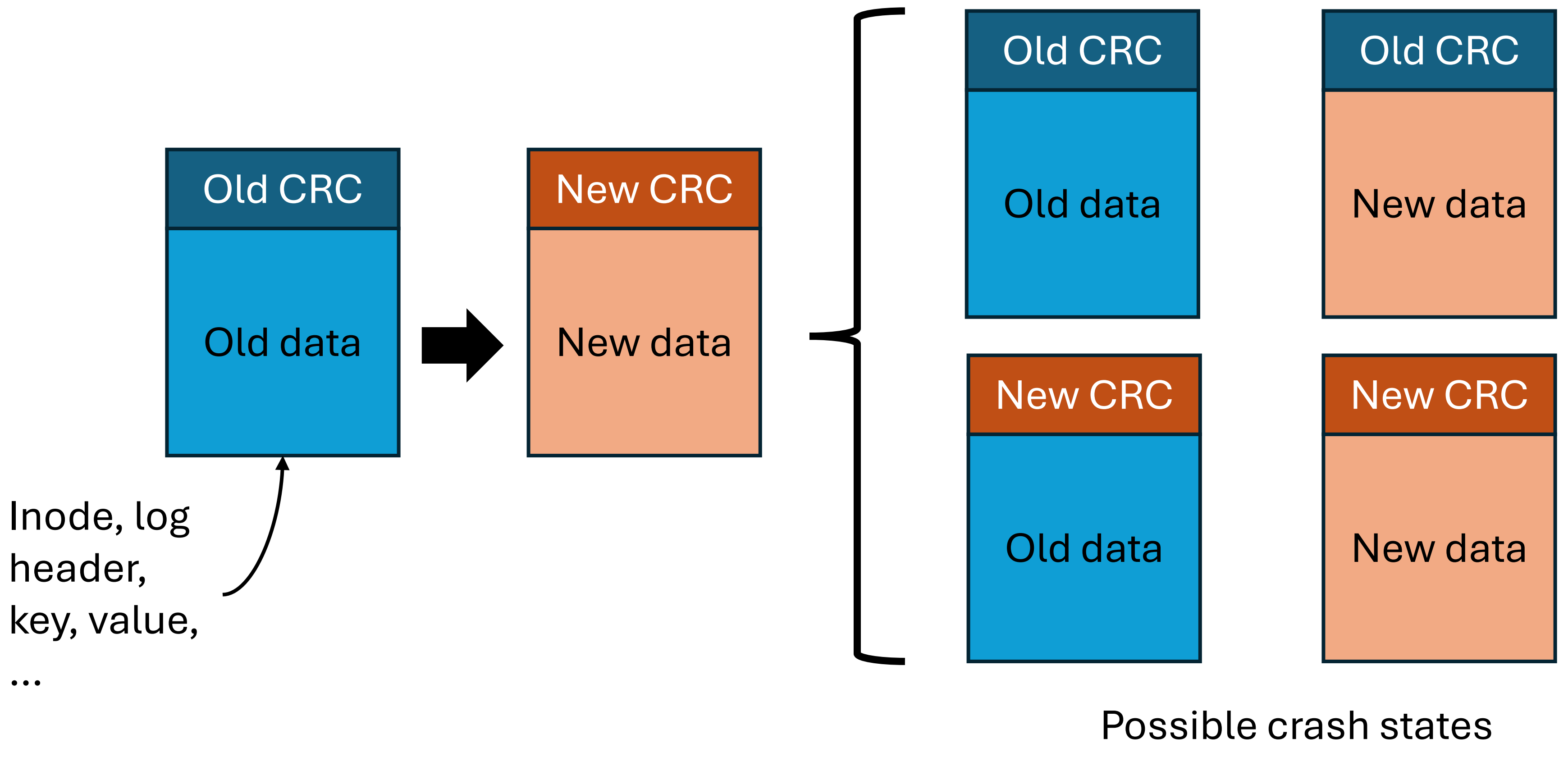
Running example



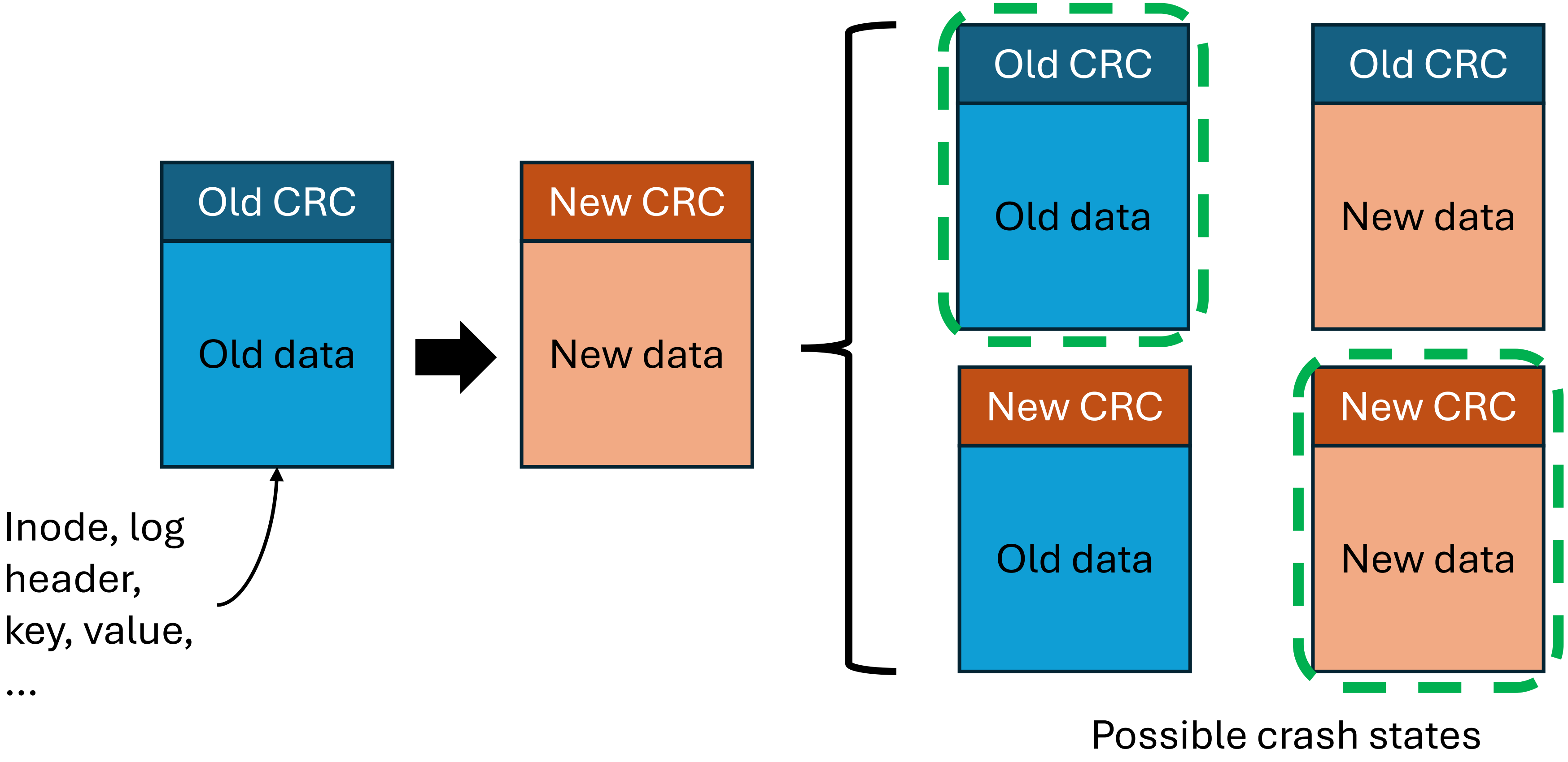
Running example



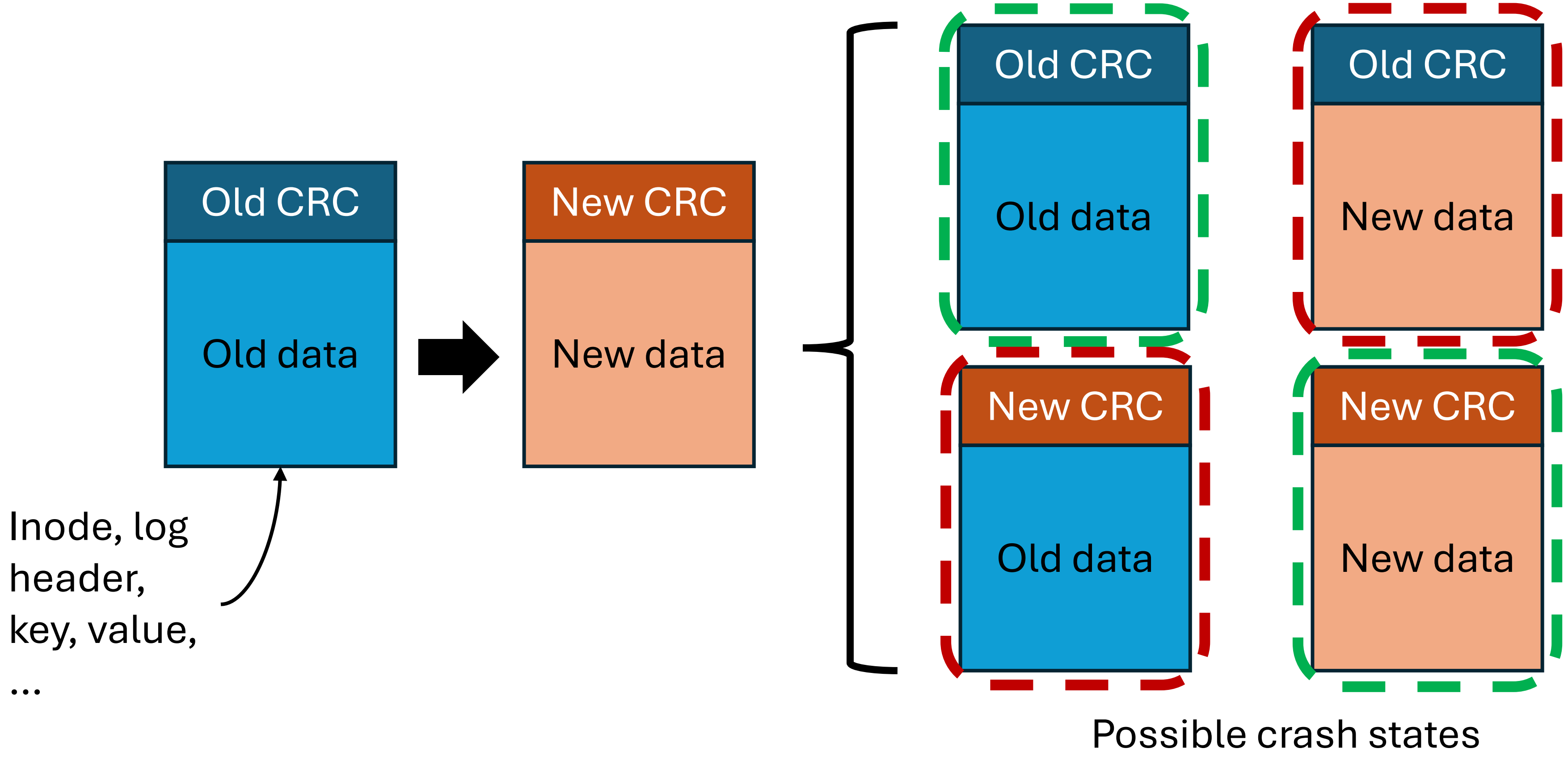
Running example



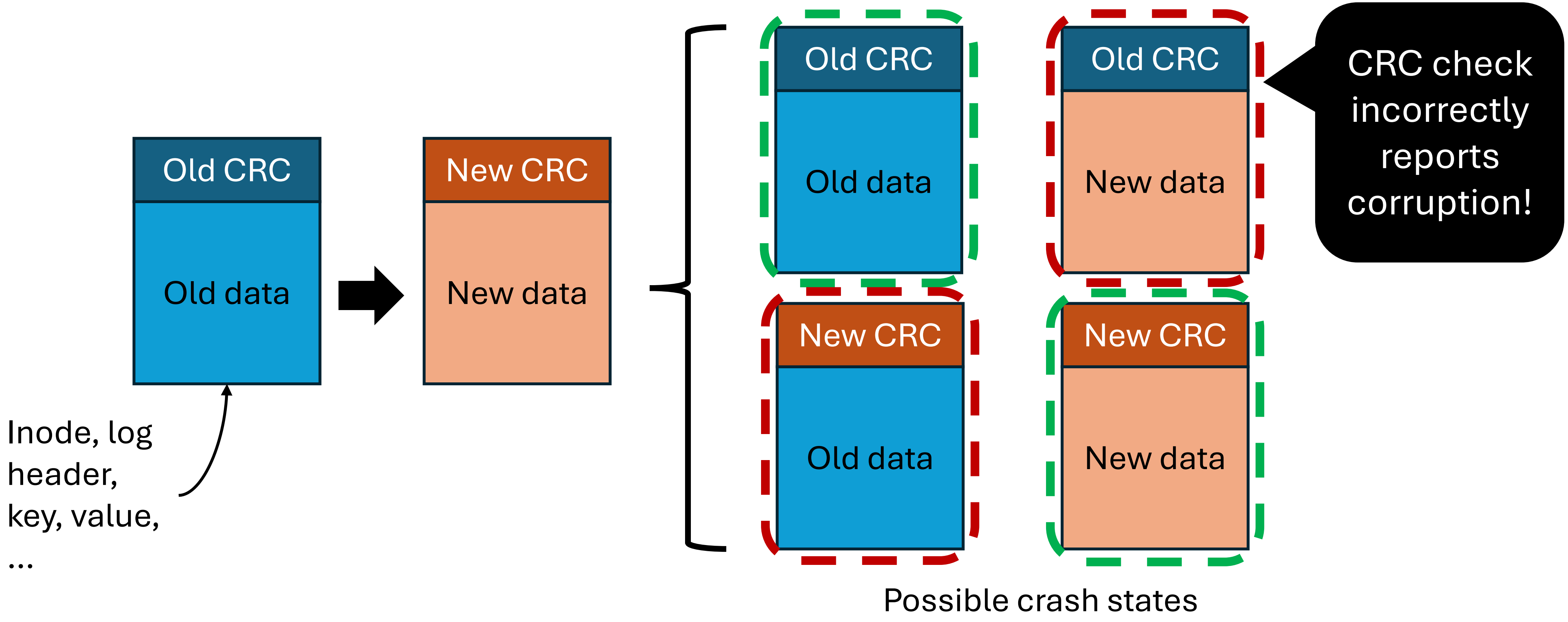
Running example



Running example



Running example



Ensuring crash consistency

Prior testing work: construct and check possible crash states

- eXplode (OSDI ‘06), CrashMonkey (OSDI ‘16), Hydra (SOSP ‘19), Yat (ATC ‘14), Vinter (ATC ‘22), Chipmunk (EuroSys ‘23), …

Ensuring crash consistency

Prior testing work: construct and check possible crash states

- eXplode (OSDI '06), CrashMonkey (OSDI '16), Hydra (SOSP '19), Yat (ATC '14), Vinter (ATC '22), Chipmunk (EuroSys '23), …

Downside of testing: **incompleteness**

Ensuring crash consistency

Prior testing work: construct and check possible crash states

- eXplode (OSDI '06), CrashMonkey (OSDI '16), Hydra (SOSP '19), Yat (ATC '14), Vinter (ATC '22), Chipmunk (EuroSys '23), …

Cannot check all workloads

Downside of testing: **incompleteness**

Ensuring crash consistency

Prior testing work: construct and check possible crash states

- eXplode (OSDI ‘06), CrashMonkey (OSDI ‘16), Hydra (SOSP ‘19), Yat (ATC ‘14), Vinter (ATC ‘22), Chipmunk (EuroSys ‘23), …

Downside of testing: **incompleteness**

Cannot check all workloads

Often intractable to check *all* crash states

Ensuring crash consistency

Prior testing work: construct and check possible crash states

- eXplode (OSDI '06), CrashMonkey (OSDI '16), Hydra (SOSP '19), Yat (ATC '14), Vinter (ATC '22), Chipmunk (EuroSys '23), …

Cannot check all workloads

Downside of testing: **incompleteness**

Often intractable to check *all* crash states

We can **statically prove** ALL crash states consistent via verification!

Verifying via pre/postconditions

Common technique supported by most verification tools

Verifying via pre/postconditions

Common technique supported by most verification tools

```
fn update(&mut self, new_data: &[u8], new_crc: u64)
      requires crc(new_data) == new_crc, ...
      ensures self.data == new_data && self.crc == new_crc, ...
```

Verifying via pre/postconditions

Common technique supported by most verification tools

Precondition must be true when the function is called

```
fn update(&mut self, new_data: &[u8], new_
        requires crc(new_data) == new_crc, ...
        ensures self.data == new_data && self.crc == new_crc, ...
```

Verifying via pre/postconditions

Common technique supported by most verification tools

Precondition must be true when the function is called

```
fn update(&mut self, new_data: &[u8], new_
        requires crc(new_data) == new_crc, ...
        ensures self.data == new_data && self.crc == new_crc, ...
```

Postcondition must be true when the function returns

Verifying via pre/postconditions

Common technique supported by most verification tools

Precondition must be true when the function is called

```
fn update(&mut self, new_data: &[u8], new_
        requires crc(new_data) == new_crc, ...
        ensures self.data == new_data && self.crc == new_crc, ...
{
        // naïve implementation
        write_to_storage(..., new_data);
        write_to_storage(..., new_crc);
}
```

Postcondition must be true when the function returns

Verifying via pre/postconditions

Common technique supported by most verification tools

```
fn update(&mut self, new_data: &[u8], new_
        requires crc(new_data) == new_crc, ...
        ensures self.data == new_data && self.crc == new_crc, ...
{
        // naïve implementation
        write_to_storage(..., new_data);
        write_to_storage(..., new_crc);
}
```

Precondition must be true when the function is called

Postcondition must be true when the function returns

How to prove intermediate crash states consistent?

Prior work: crash conditions

```
fn update(&mut self, new_data: &[u8], new_crc: u64)
      requires crc(new_data) == new_crc, ...
      ensures self.data == new_data && self.crc == new_crc, ...
      crash self.data == new_data && self.crc == new_crc ||
            (self.data == old(self).data && self.crc == old(self).crc)
{
      // naïve implementation
      write_to_storage(..., new_data);
      write_to_storage(..., new_crc);
}
```

Prior work: crash conditions

```
fn update(&mut self, new_data: &[u8], new_crc: u64)
      requires crc(new_data) == new_crc, ...
      ensures self.data == new_data && self.crc == new_crc, ...
      crash self.data == new_data && self.crc == new_crc ||
            (self.data == old(self).data && self.crc == old(self).crc)
{
      // naïve implementation
      write_to_storage(..., new_data);
      write_to_storage(..., new_crc);
}
```

Prior work: crash conditions

```
fn update(&mut self, new_data: &[u8], new_crc: u64)
      requires crc(new_data) == new_crc, ...
      ensures self.data == new_data && self.crc == new_crc, ...
      crash self.data == new_data && self.crc == new_crc ||
            (self.data == old(self).data && self.crc == old(self).crc)
{
      // naïve implementation
      write_to_storage(..., new_data);
      write_to_storage(..., new_crc);
}
```

Crash conditions abstractly describe legal crash states (FSCQ, SOSP '15)

Prior work: crash conditions

```
fn update(&mut self, new_data: &[u8], new_crc: u64)
        requires crc(new_data) == new_crc, ...
        ensures self.data == new_data && self.crc == new_crc, ...
        crash self.data == new_data && self.crc == new_crc ||
              (self.data == old(self).data && self.crc == old(self).crc)
{
        // naï
        write_
        write_to_storage(..., new_crc);
}
```

Crash conditions are not supported by most verification tools

Crash conditions abstractly describe legal crash states (FSCQ, SOSP '15)

Proving crash consistency

Goal: verify crash consistency using only pre/postconditions

Proving crash consistency

Goal: verify crash consistency using only pre/postconditions

Observations:

1. Each durable write introduces a set of new crash states
2. These crash states can be described *before* the write is invoked

Proving crash consistency

Goal: verify crash consistency using only pre/postconditions

Observations:

1. Each durable write introduces a set of new crash states
2. These crash states can be described *before* the write is invoked

Key insight: crash-consistency proof requirements can be written as **preconditions**!

Proving crash consistency

Goal: verify crash consistency using only pre/postconditions

Observations:

1. Each durable write introduces a set of new crash states
2. These crash states can be described *before* the write is invoked

Key insight: crash-consistency proof requirements can be written as **preconditions**!

Can be done in nearly any verification tool!

Preconditions **o**n **W**rites **E**nforcing **R**ecoverability (**PoWER**)

```
write_to_storage(..., new_data);
```

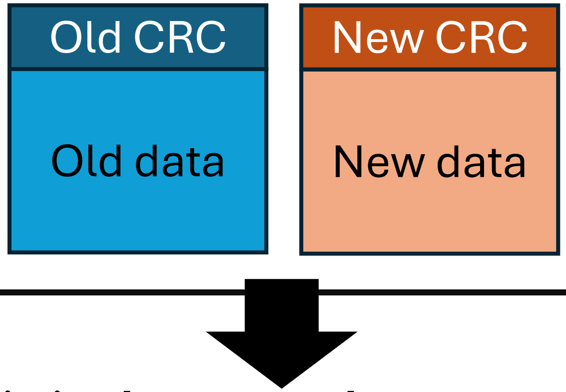
Preconditions **o**n **W**rites **E**nforcing **R**ecoverability (**PoWER**)

```
write_to_storage(..., new_data);
```

```
fn write_to_storage(..., bytes: &[u8])
      requires for all new crash states s, recover(s) is consistent
      ensures bytes written to storage device
```

Preconditions **o**n **W**rites **E**nforcing **R**ecoverability (**PoWER**)

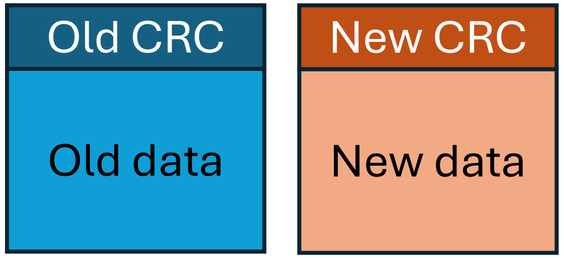
```
write_to_storage(..., new_data);
```




```
fn write_to_storage(..., bytes: &[u8])
      requires for all new crash states s, recover(s) is consistent
      ensures bytes written to storage device
```

Preconditions **o**n **W**rites **E**nforcing **R**ecoverability (**PoWER**)

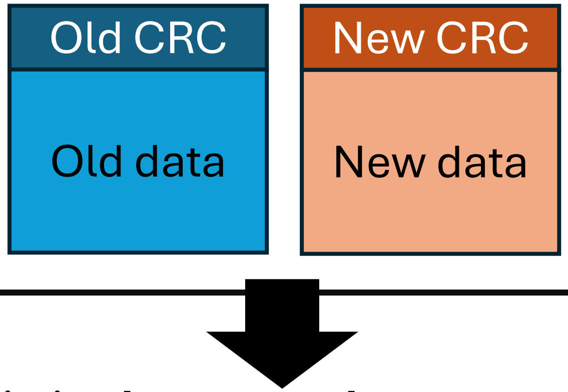
```
lemma_data_update_consistent(...);
write_to_storage(..., new_data);
```




```
fn write_to_storage(..., bytes: &[u8])
      requires for all new crash states s, recover(s) is consistent
      ensures bytes written to storage device
```

Preconditions on Writes Enforcing Recoverability (PoWER)

```
lemma_data_update_consistent(...);
write_to_storage(..., new_data);
```



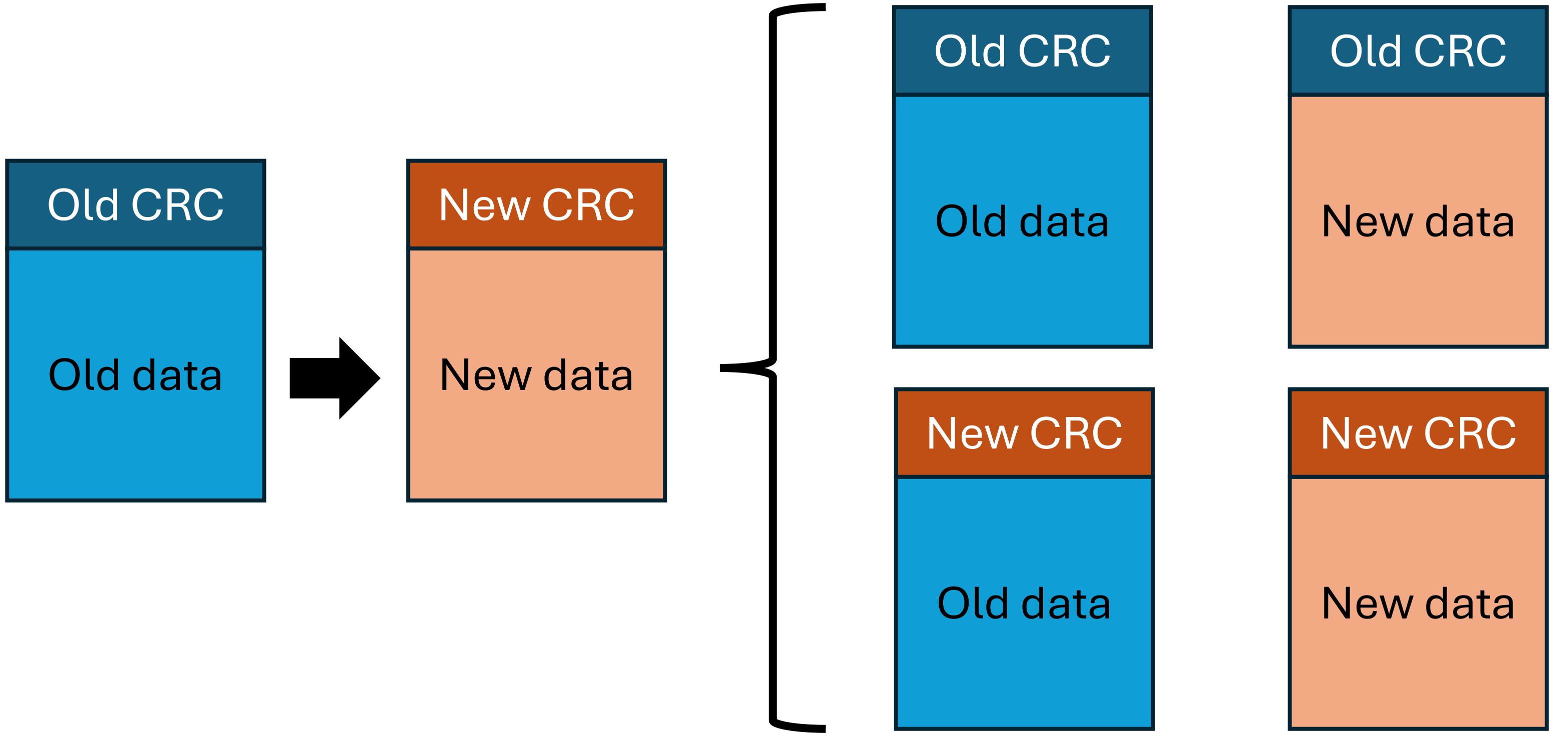

```
fn write_to_storage(..., bytes: &[u8])
      requires for all new crash states s, recover(s) is consistent
      ensures bytes written to storage device
```

Satisfy precondition ==> prove crash consistency!

See paper for…

- Detailed description of PoWER technique
- Strategies for writing crash-consistency proofs
- Discussion of proofs that PoWER is sound
- PoWER and concurrency

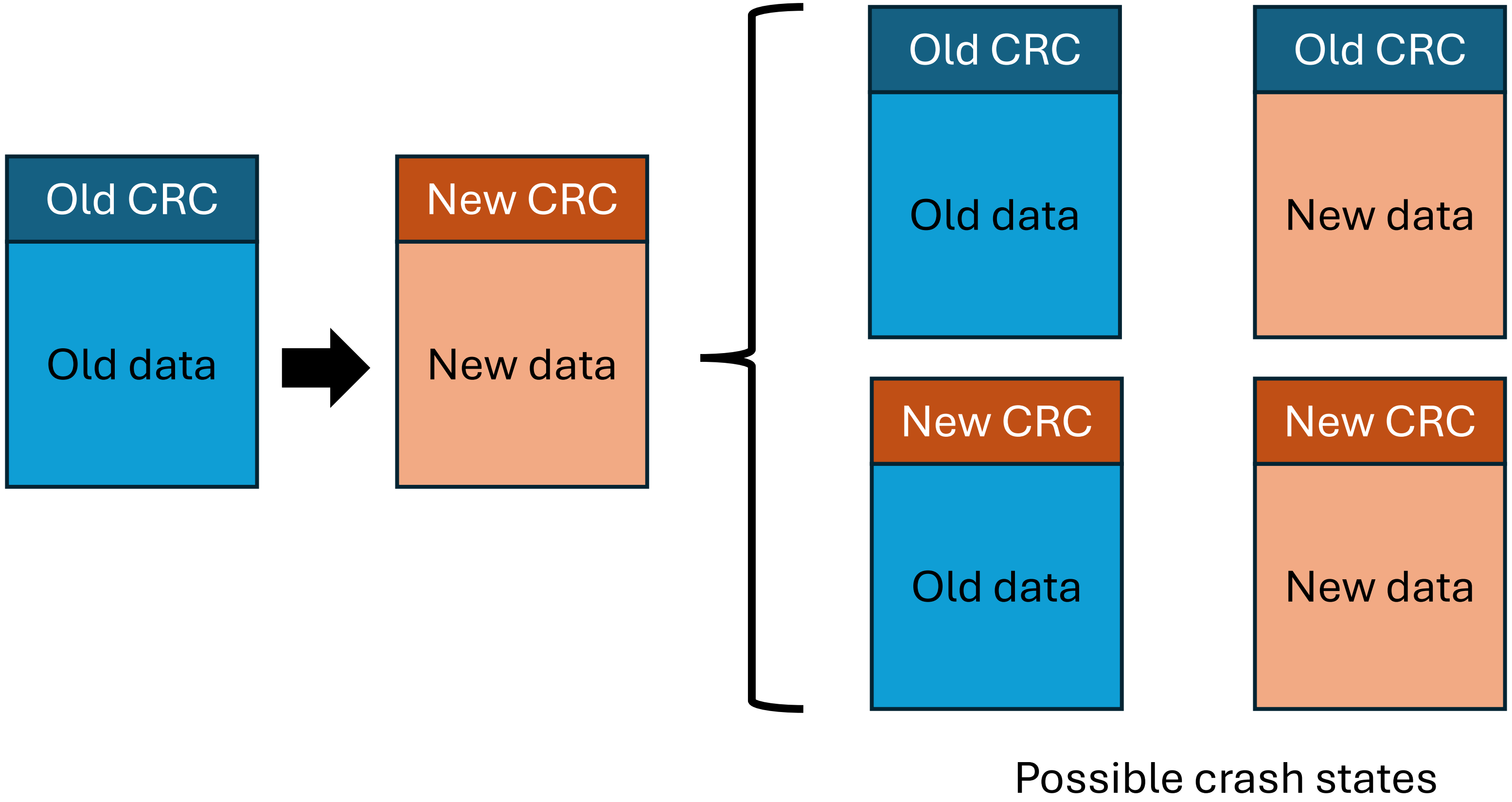
Back to our example



Possible crash states

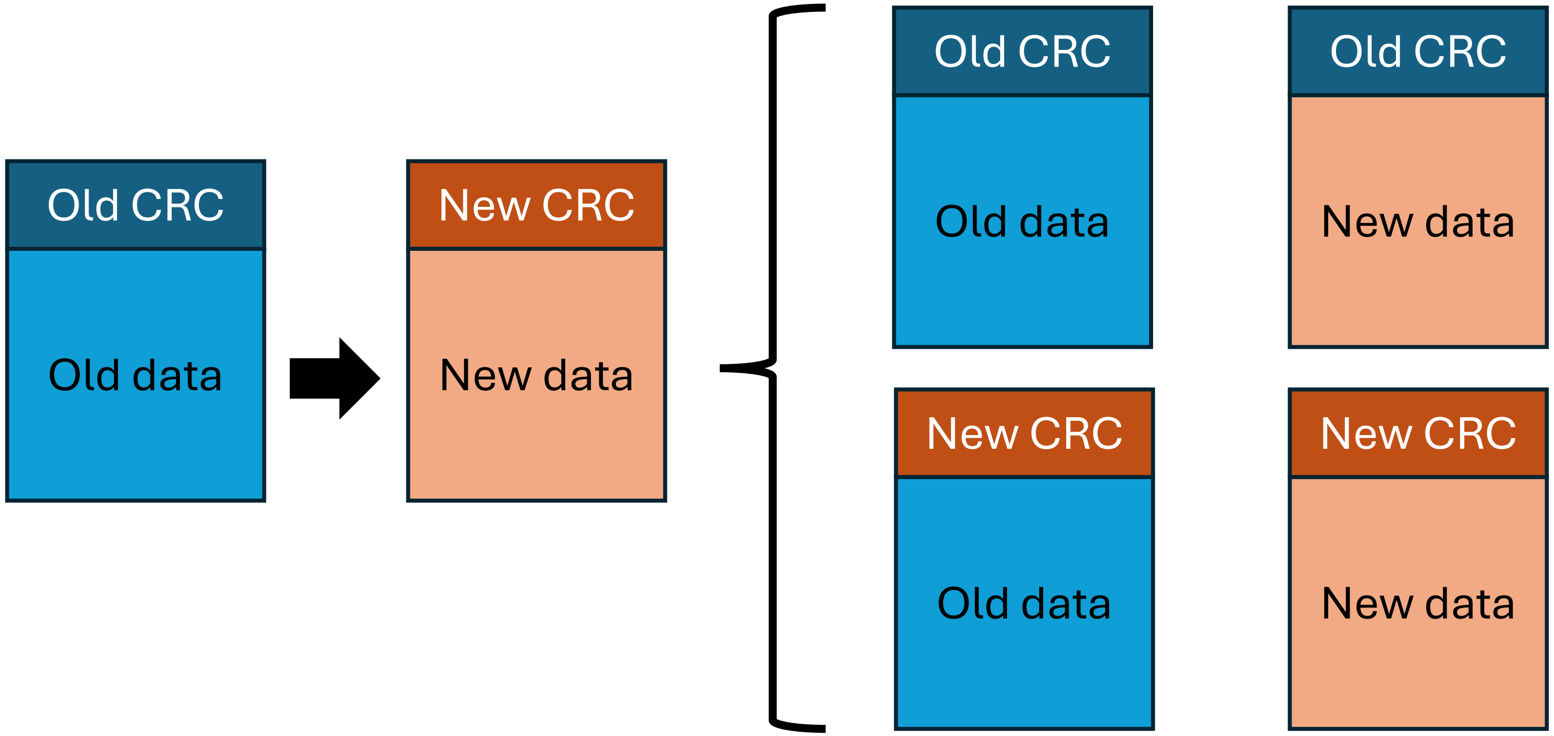
Back to our example

How do we implement this operation in a crash-consistent way?



Block-based systems

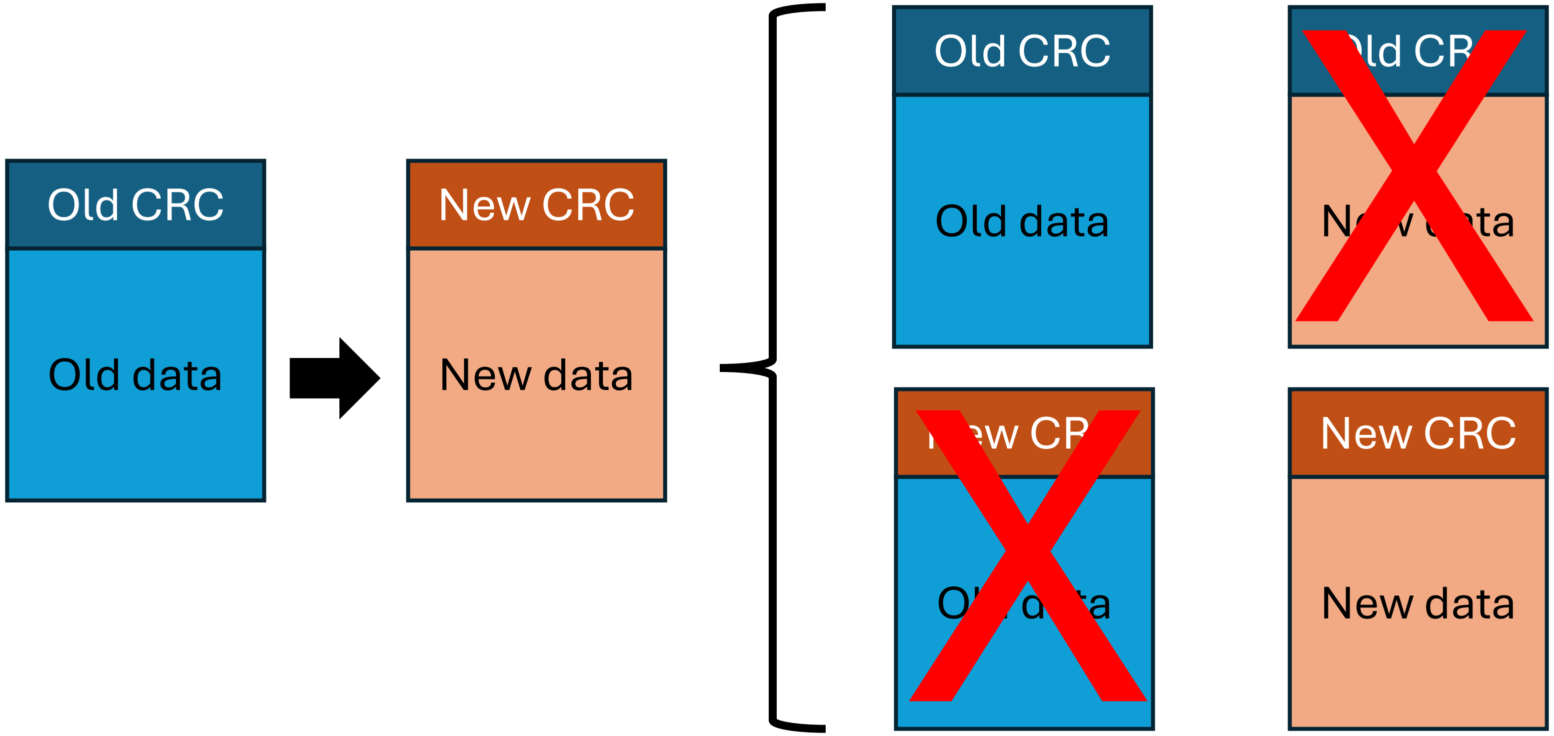
Atomic block-sized writes → 1 CRC per block



Possible crash states

Block-based systems

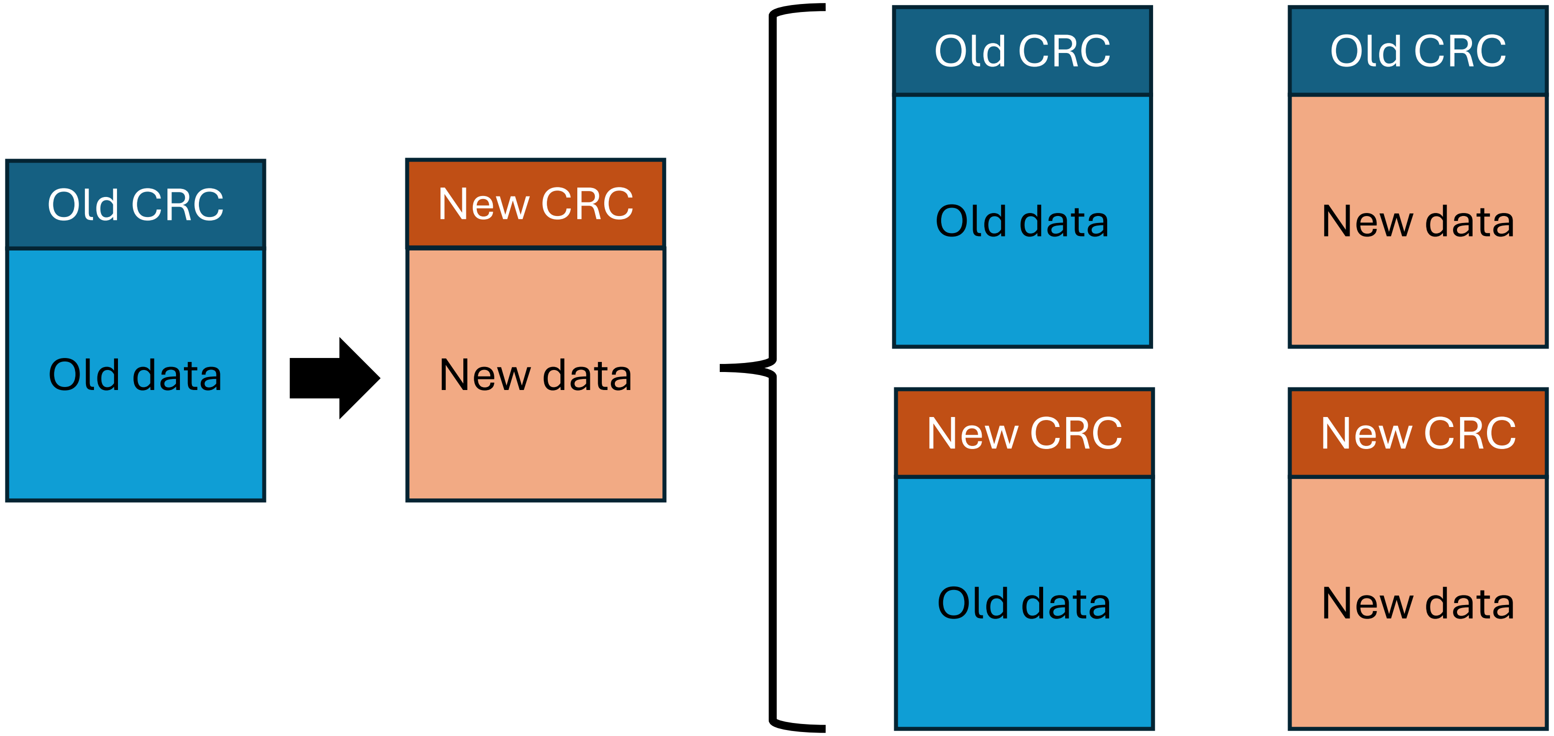
Atomic block-sized writes → 1 CRC per block



Possible crash states

Persistent memory systems

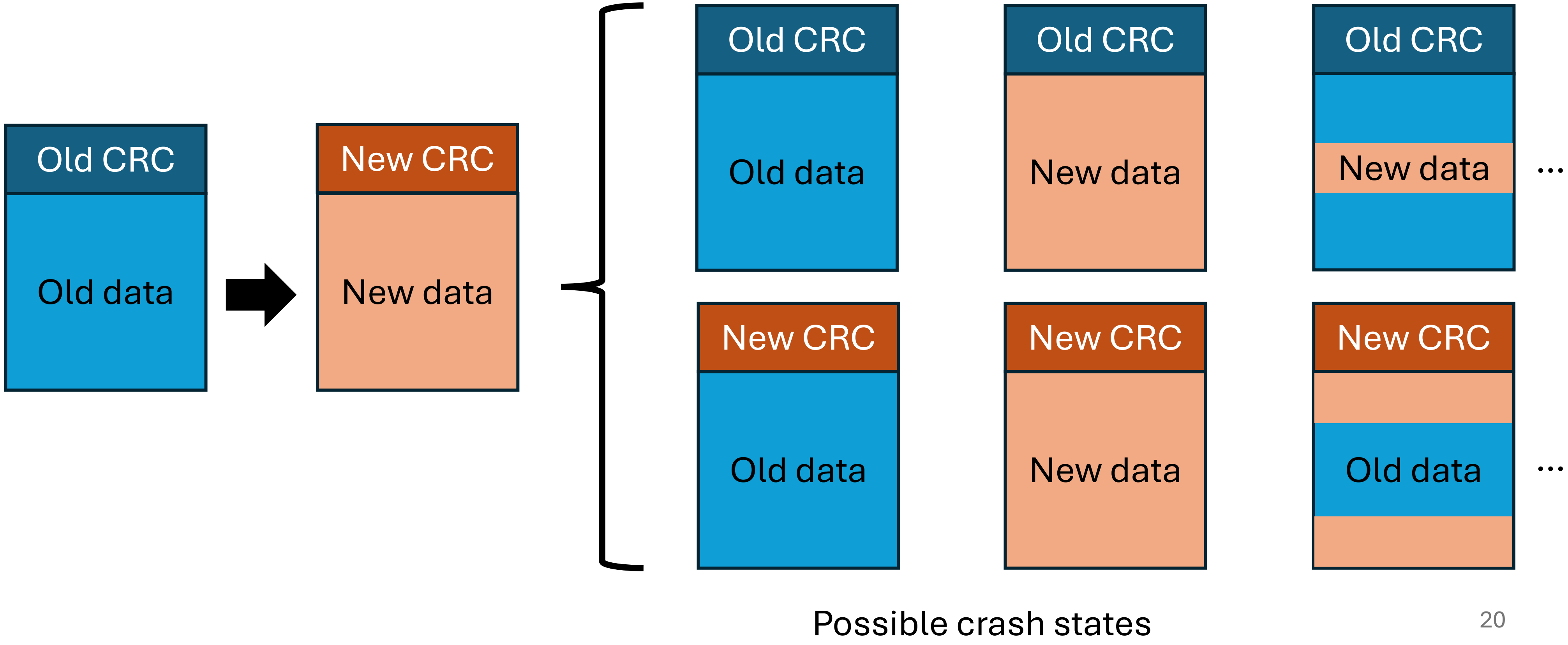
8-byte atomic writes are more challenging!



Possible crash states

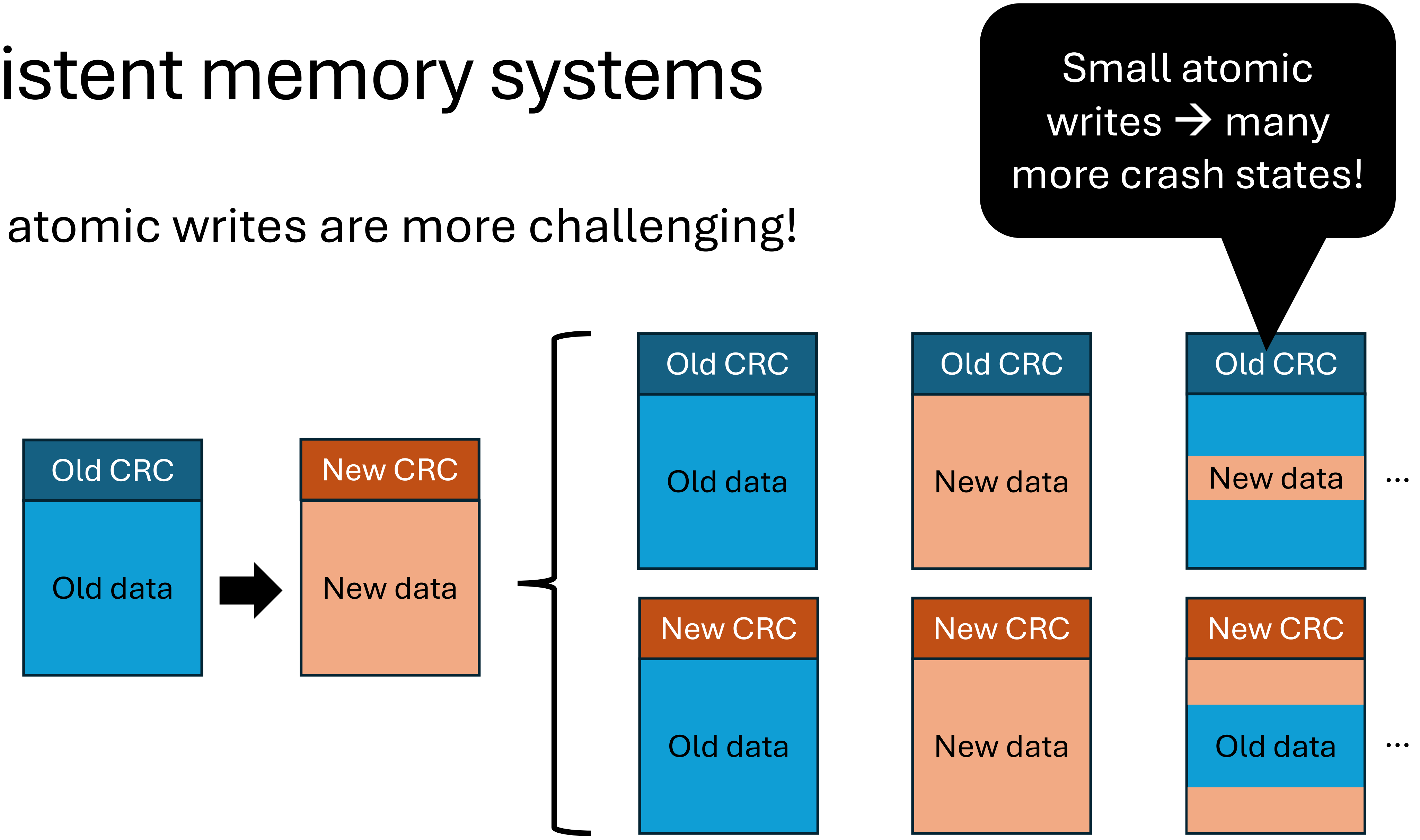
Persistent memory systems

8-byte atomic writes are more challenging!



Persistent memory systems

8-byte atomic writes are more challenging!



Possible crash states

Prior work: Tick-Tock algorithm

Prior work: Tick-Tock algorithm

Introduced by NOVA-Fortis file system (SOSP ‘17)

Prior work: Tick-Tock algorithm

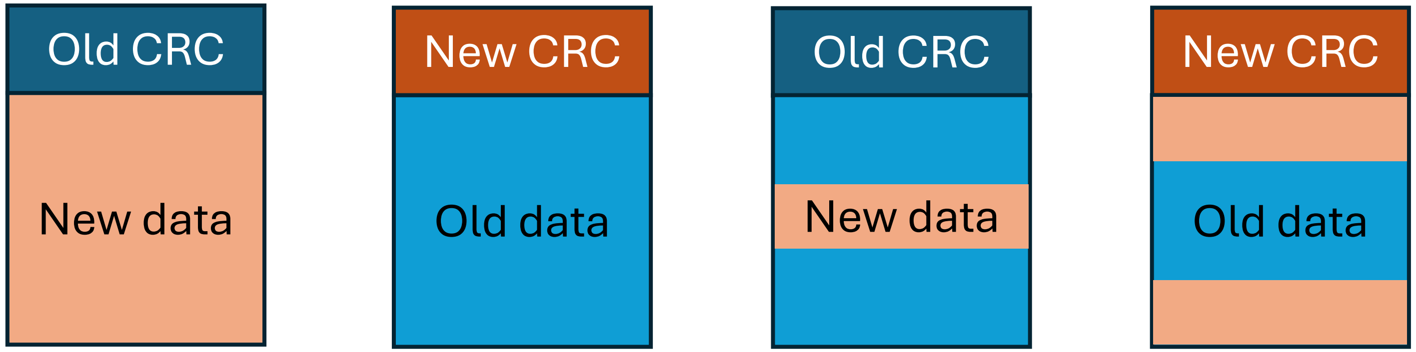
Introduced by NOVA-Fortis file system (SOSP ‘17)

Uses CRCs to detect bit flips *and* crash inconsistencies

Prior work: Tick-Tock algorithm

Introduced by NOVA-Fortis file system (SOSP '17)

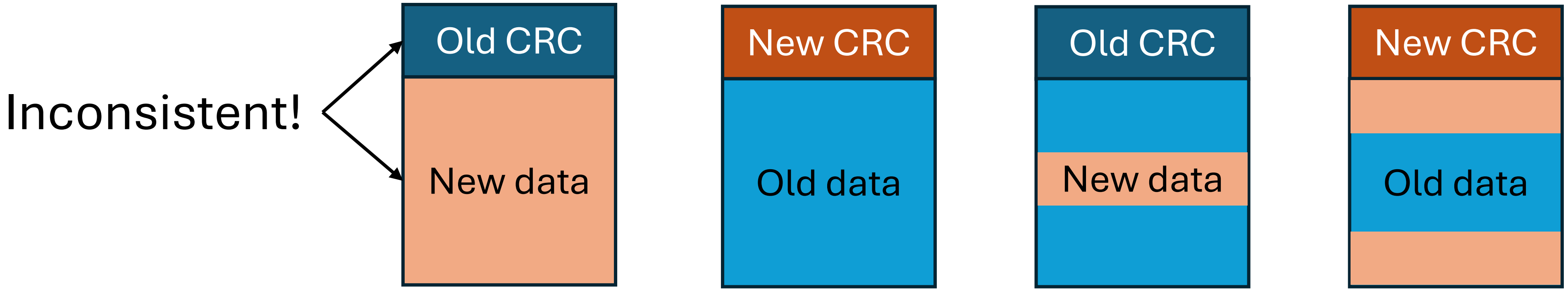
Uses CRCs to detect bit flips *and* crash inconsistencies



Prior work: Tick-Tock algorithm

Introduced by NOVA-Fortis file system (SOSP ‘17)

Uses CRCs to detect bit flips *and* crash inconsistencies

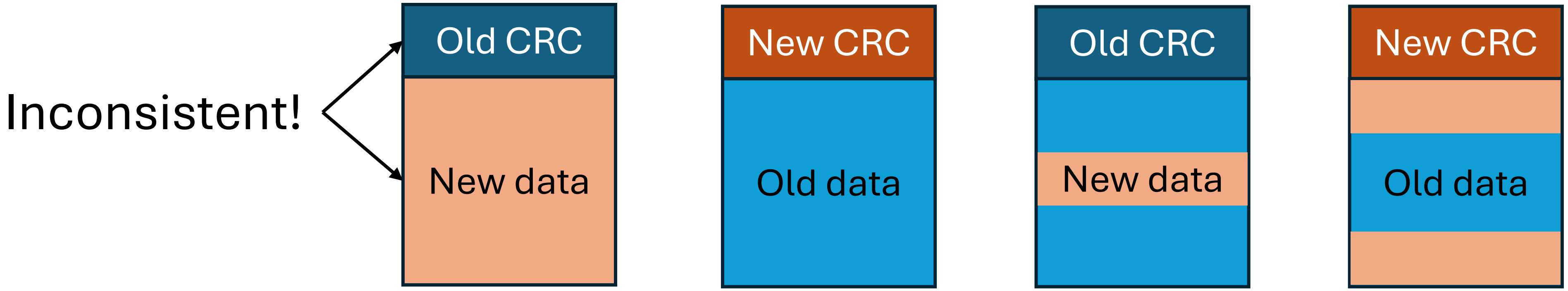


Prior work: Tick-Tock algorithm

Prior work found CRC atomicity bugs (LeBlanc EuroSys '23)

Introduced by NOVA-Fortis file system (SOSP '17)

Uses CRCs to detect bit flips *and* crash inconsistencies



Challenges

Challenges

CRCs are designed to detect a few bit flips

- Basis of our model of corruption (see paper)

Challenges

CRCs are designed to detect a few bit flips

- Basis of our model of corruption (see paper)

Crashes can result in many arbitrarily different states

Challenges

CRCs are designed to detect a few bit flips

- Basis of our model of corruption (see paper)

Crashes can result in many arbitrarily different states

CRCs cannot detect crashes as reliably as corruption

Challenges

CRCs are designed to detect a few bit flips

- Basis of our model of corruption (see paper)

Crashes can result in many arbitrarily different states

CRCs cannot detect crashes as reliably as corruption

Verified code should work even in rare, worst-case scenarios

Solution: corruption-detecting Boolean (CDB)

Solution: corruption-detecting Boolean (CDB)

Two possible 8-byte values: CRC(0), CRC(1)

Solution: corruption-detecting Boolean (CDB)

Two possible 8-byte values: CRC(0), CRC(1)

Store Boolean value and checksum in same 8 bytes

Solution: corruption-detecting Boolean (CDB)

Two possible 8-byte values: CRC(0), CRC(1)

Store Boolean value and checksum in same 8 bytes

Supports crash-atomic updates!

Solution: corruption-detecting Boolean (CDB)

Two possible 8-byte values: CRC(0), CRC(1)

Store Boolean value and checksum in same 8 bytes

Supports crash-atomic updates!

Capybara systems use CDBs:

- As a validity “bit”
- For atomic data+CRC updates via CoW-like technique

Solution: corruption-detecting Boolean (CDB)

Two possible 8-byte values: CRC(0), CRC(1)

Store Boolean value and checksum in same 8 bytes

Supports crash-atomic updates!

Capybara systems use CDBs:

- As a validity “bit”

- For atomic data+CRC updates via CoW-like technique

Solution: corruption-detecting Boolean (CDB)

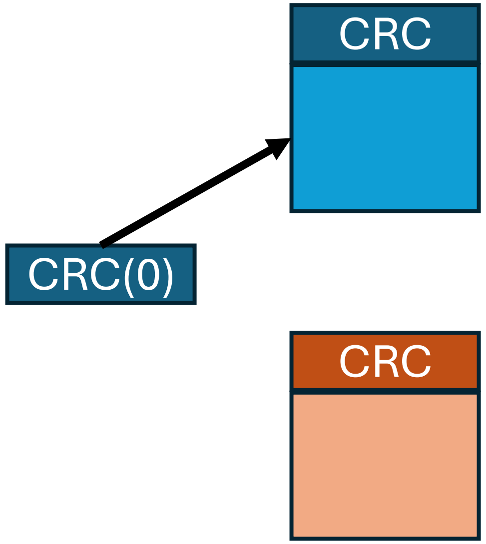
Two possible 8-byte values: CRC(0), CRC(1)

Store Boolean value and checksum in same 8 bytes

Supports crash-atomic updates!

Capybara systems use CDBs:

- As a validity "bit"
- For atomic data+CRC updates via CoW-like technique



Solution: corruption-detecting Boolean (CDB)

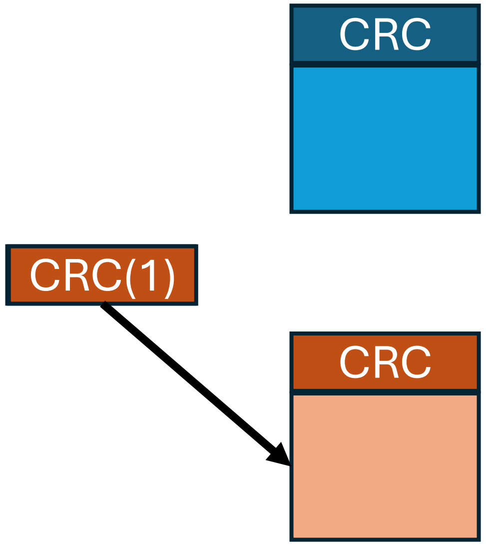
Two possible 8-byte values: CRC(0), CRC(1)

Store Boolean value and checksum in same 8 bytes

Supports crash-atomic updates!

Capybara systems use CDBs:

- As a validity “bit”
- For atomic data+CRC updates via CoW-like technique



Solution: corruption-detecting Boolean (CDB)

Two possible 8-byte values: CRC(0), CRC(1)

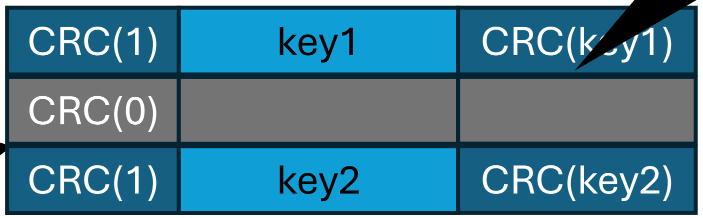
Store Boolean value and checksum in same 8 bytes

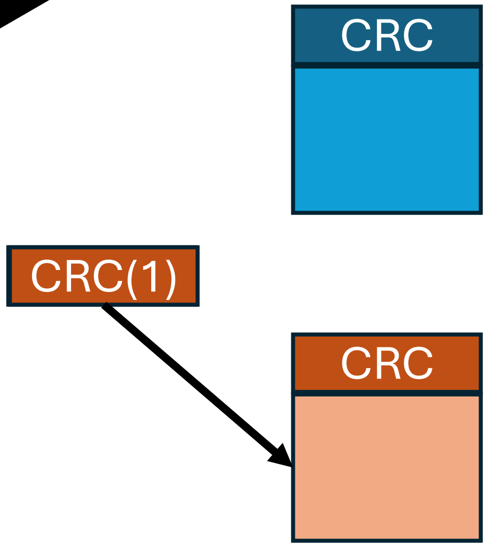
Supports crash-atomic updates!

Capybara systems use CDBs:

- As a validity "bit"
- For atomic data+CRC updates via CoW-like technique

Broadly useful primitive developed *because* of verification!





CapybaraKV

CapybaraKV

Designed for Azure Storage use case

CapybaraKV

Designed for Azure Storage use case

Persistent-memory key-value store written in Verus

CapybaraKV

Designed for Azure Storage use case

Persistent-memory key-value store written in Verus

Concurrent, crash-atomic operations on fixed-size item and list values

CapybaraKV

Designed for Azure Storage use case

Persistent-memory key-value store written in Verus

Concurrent, crash-atomic operations on fixed-size item and list values

~25KLOC (15K proof)

CapybaraKV

Designed for Azure Storage use case

Persistent-memory key-value store written in Verus

Concurrent, crash-atomic operations on fixed-size item and list values

~25KLOC (15K proof)

Verifies in <1 min on most machines

CapybaraKV

Designed for Azure Storage use case

Persistent-memory key-value store written in Verus

Concurrent, crash-atomic operations on fixed-size item and list values

~25KLOC (15K proof)

Verifies in <1 min on most machines

Similar or better performance to unverified systems pmem-Redis, pmem-RocksDB, Viper (see paper)

Conclusion

github.com/microsoft/verified-storage

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

🏆 ***Distinguished Artifact Award***

Conclusion

Contributions

github.com/microsoft/verified-storage

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

🏆 ***Distinguished Artifact Award***

Conclusion

Contributions

- First formally verified PM storage systems

github.com/microsoft/verified-storage

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

🏆 ***Distinguished Artifact Award***

Conclusion

Contributions

- First formally verified PM storage systems
- Useful new techniques for building robust verified systems

github.com/microsoft/verified-storage

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

Distinguished Artifact Award

Conclusion

Contributions

- First formally verified PM storage systems
- Useful new techniques for building robust verified systems

Lessons learned

github.com/microsoft/verified-storage

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

🏆 ***Distinguished Artifact Award***

Conclusion

Contributions

- First formally verified PM storage systems
- Useful new techniques for building robust verified systems

Lessons learned

- Crashes and corruption impact data differently

github.com/microsoft/verified-storage

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

🏆 ***Distinguished Artifact Award***

Conclusion

Contributions

- First formally verified PM storage systems
- Useful new techniques for building robust verified systems

Lessons learned

- Crashes and corruption impact data differently
- Rigor of verification can help develop broadly useful techniques

github.com/microsoft/verified-storage

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

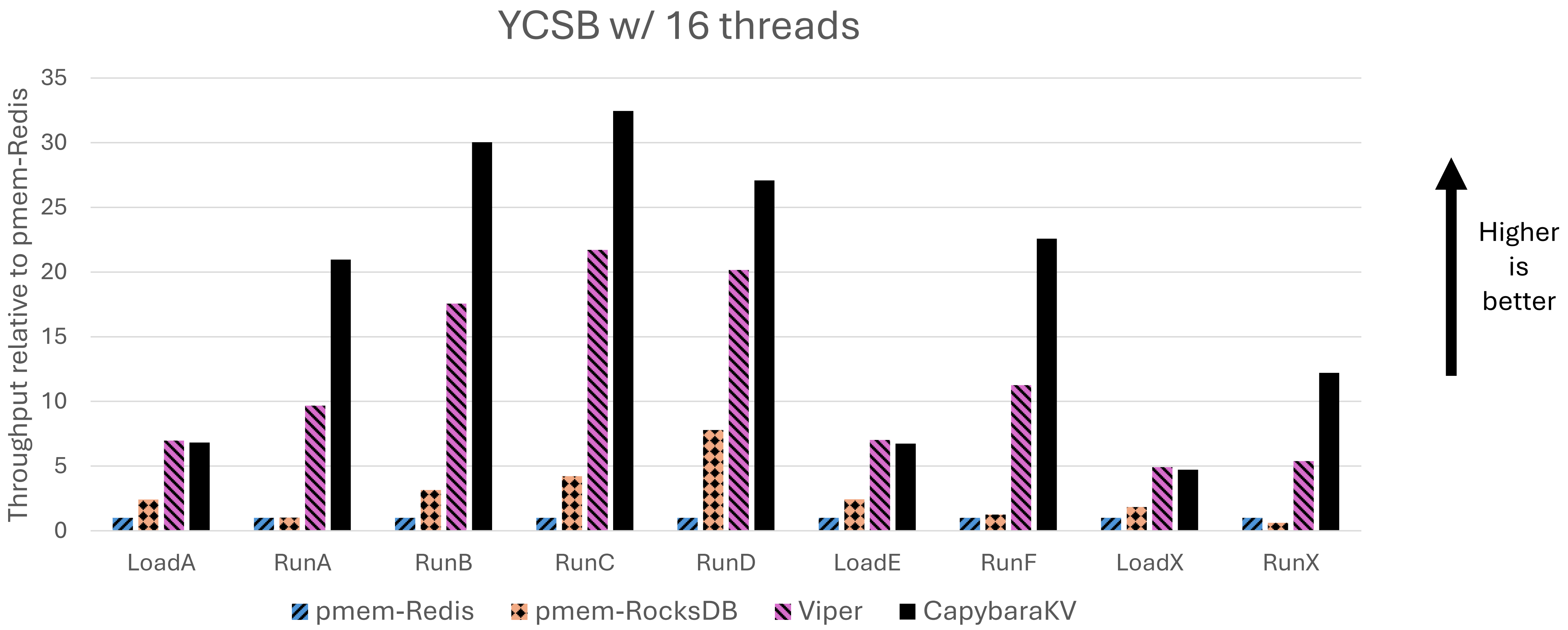
ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

Distinguished Artifact Award

Additional slides

Evaluation: YCSB



CapybaraNS

PM notary service written in Dafny

Demonstrates that PoWER works w/ tools besides Verus

Built and verified in ~3 person days

~1.5KLOC (673 proof)

PoWER limitations

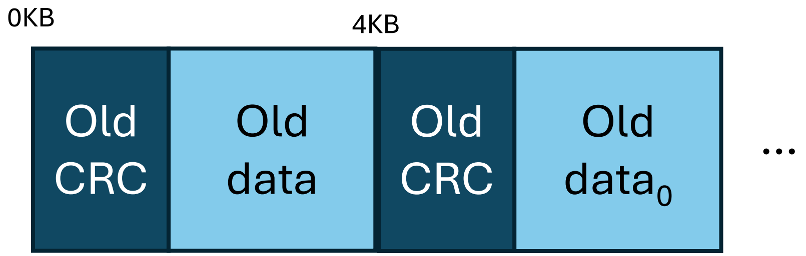
- Not *all* verifiers support the required standard features
  - PoWER also requires quantifiers and ghost variables
  - Push-button verifiers like TPot or Yggdrasil may not support PoWER
- Cannot support arbitrary fine-grained concurrent writes to shared storage regions
- Correctness depends on specifications and correctness of verifier/compiler

CapybaraKV limitations

- Requires storage space to be statically allocated at initialization
  - Other evaluated systems can grow/shrink dynamically
  - We configure Viper to allocate sufficient space at init for fair comparison
  - Not fundamental
- Keeps all keys in memory -- increases memory footprint and startup time
  - Pmem-Redis and Viper also keep all keys in memory
  - Not fundamental
- Sharded concurrency approach does not allow concurrent writes to different records

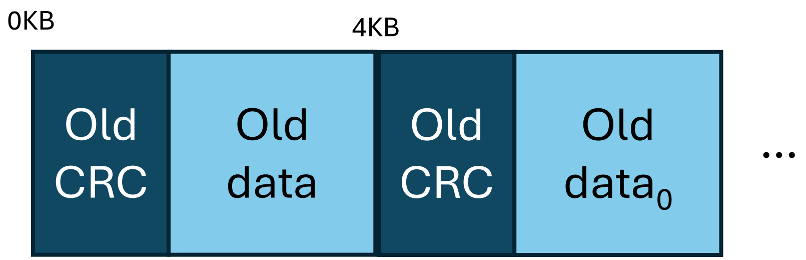
Crash-consistent CRCs

Block-sized atomic writes: one CRC per block



Crash-consistent CRCs

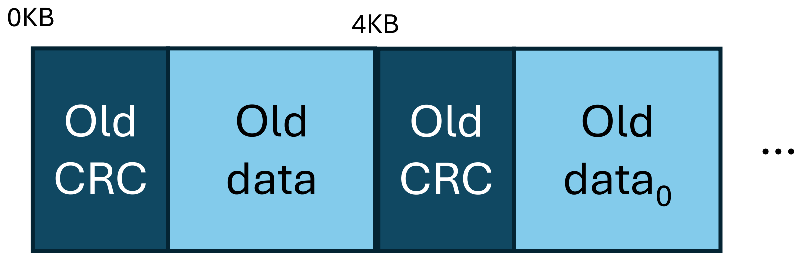
Block-sized atomic writes: one CRC per block



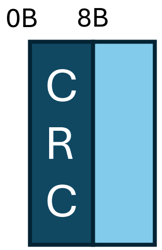
8-byte atomic writes are more challenging!

Crash-consistent CRCs

Block-sized atomic writes: one CRC per block

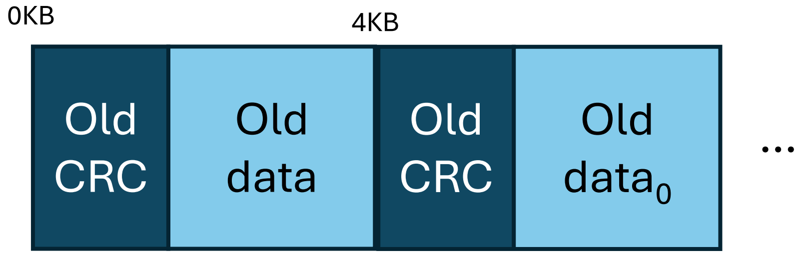


8-byte atomic writes are more challenging!

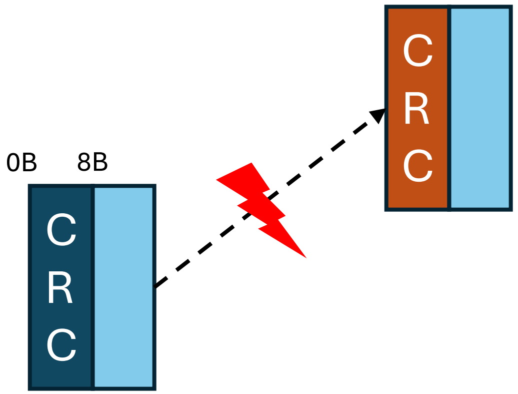


Crash-consistent CRCs

Block-sized atomic writes: one CRC per block

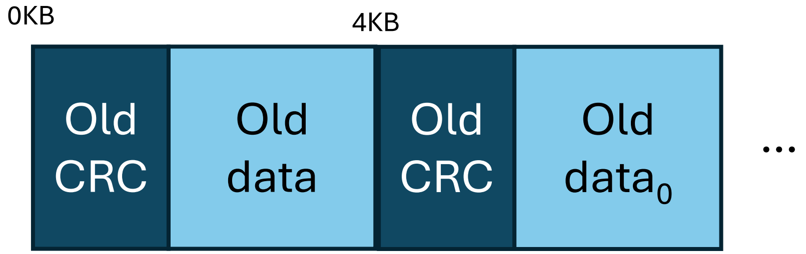


8-byte atomic writes are more challenging!

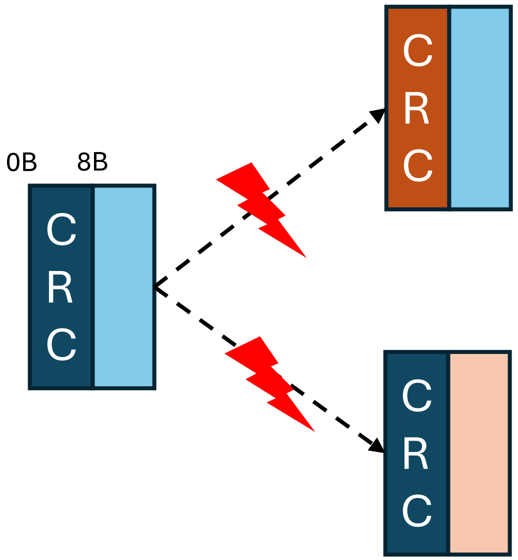


Crash-consistent CRCs

Block-sized atomic writes: one CRC per block

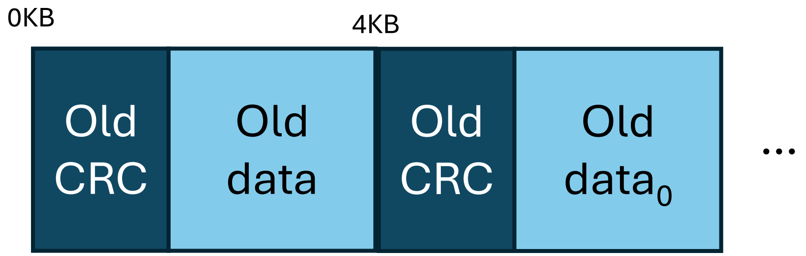


8-byte atomic writes are more challenging!

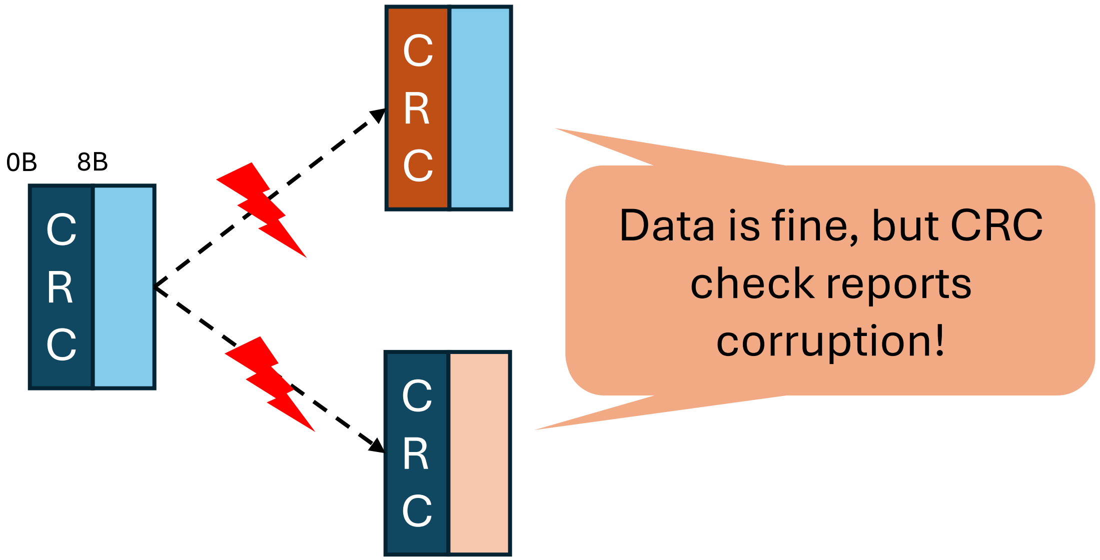


Crash-consistent CRCs

Block-sized atomic writes: one CRC per block

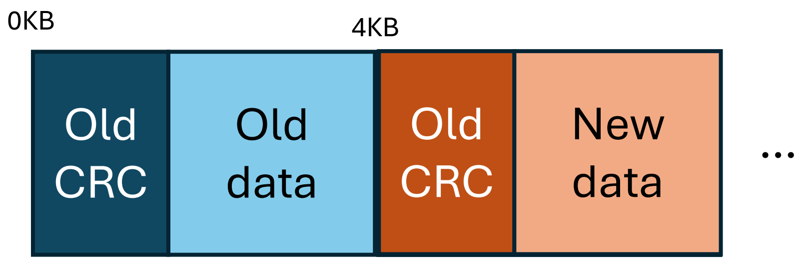


8-byte atomic writes are more challenging!

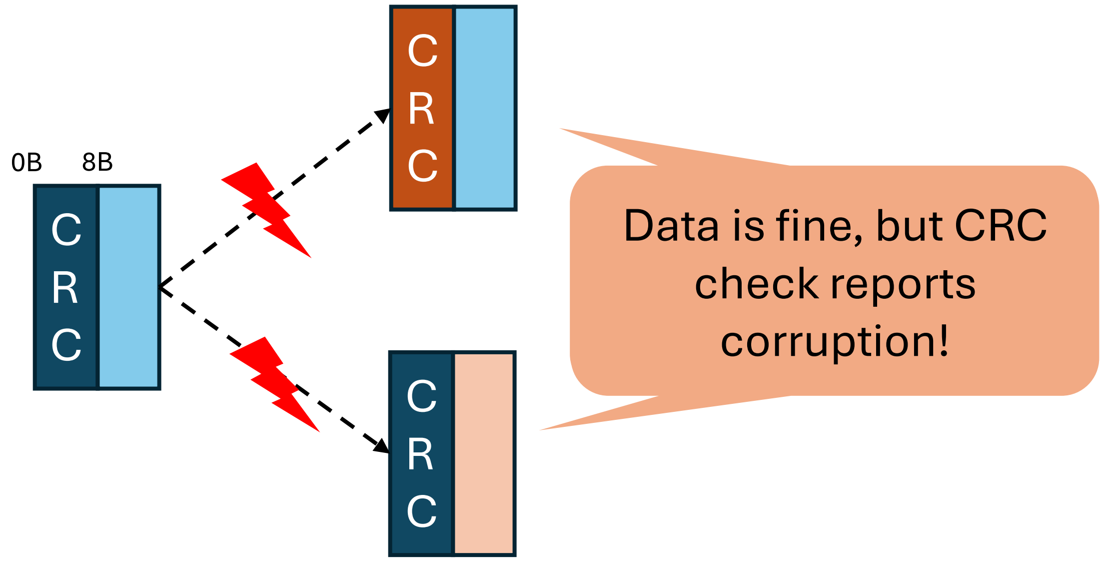


Crash-consistent CRCs

Block-sized atomic writes: one CRC per block

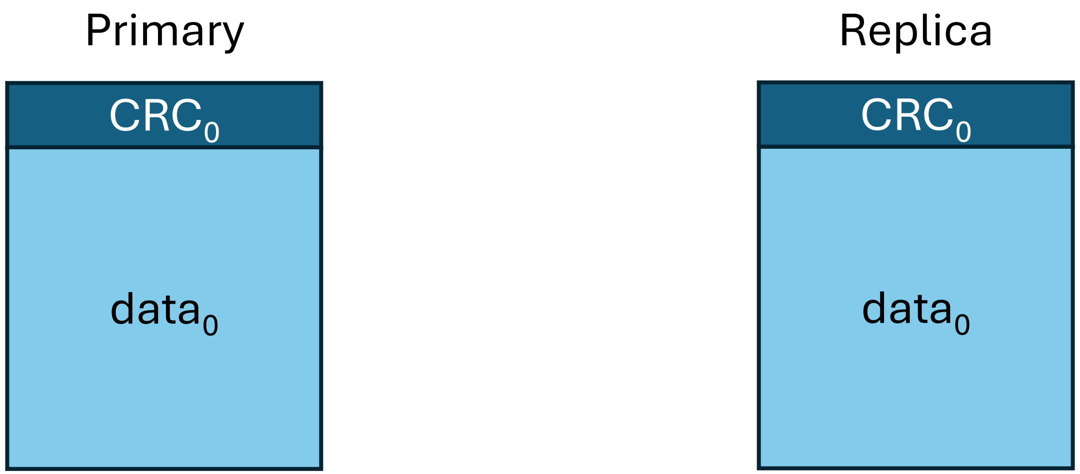


8-byte atomic writes are more challenging!



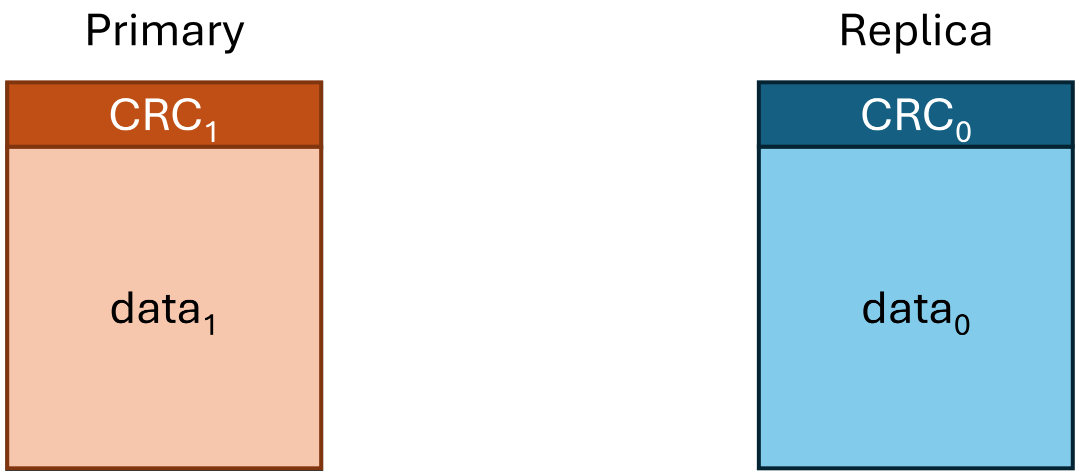
Prior work: Tick-Tock algorithm

Introduced in NOVA-Fortis file system (Xu SOSP ’17)



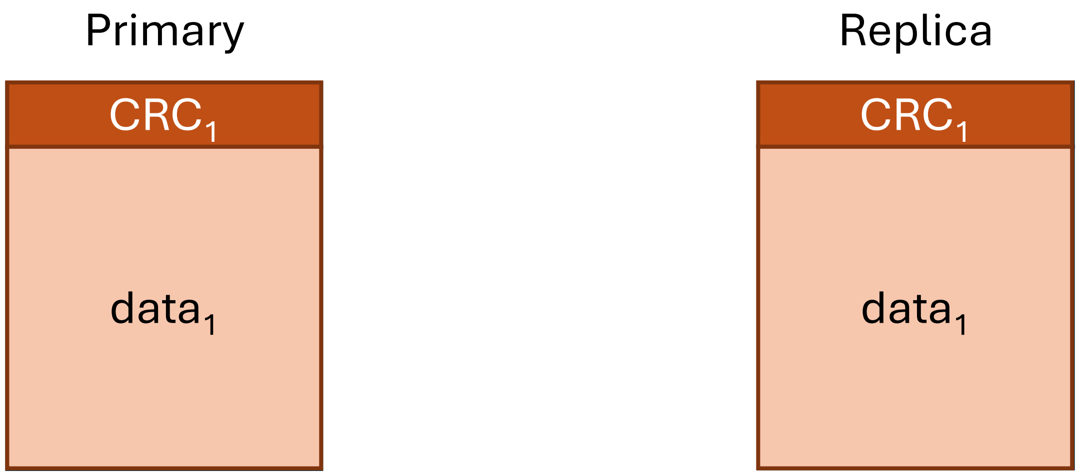
Prior work: Tick-Tock algorithm

Introduced in NOVA-Fortis file system (Xu SOSP ’17)



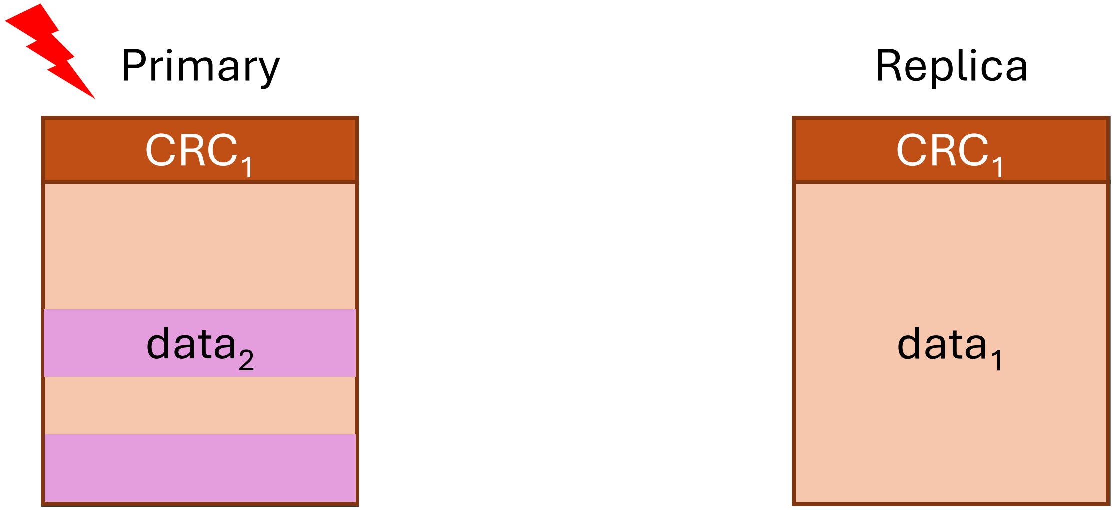
Prior work: Tick-Tock algorithm

Introduced in NOVA-Fortis file system (Xu SOSP '17)



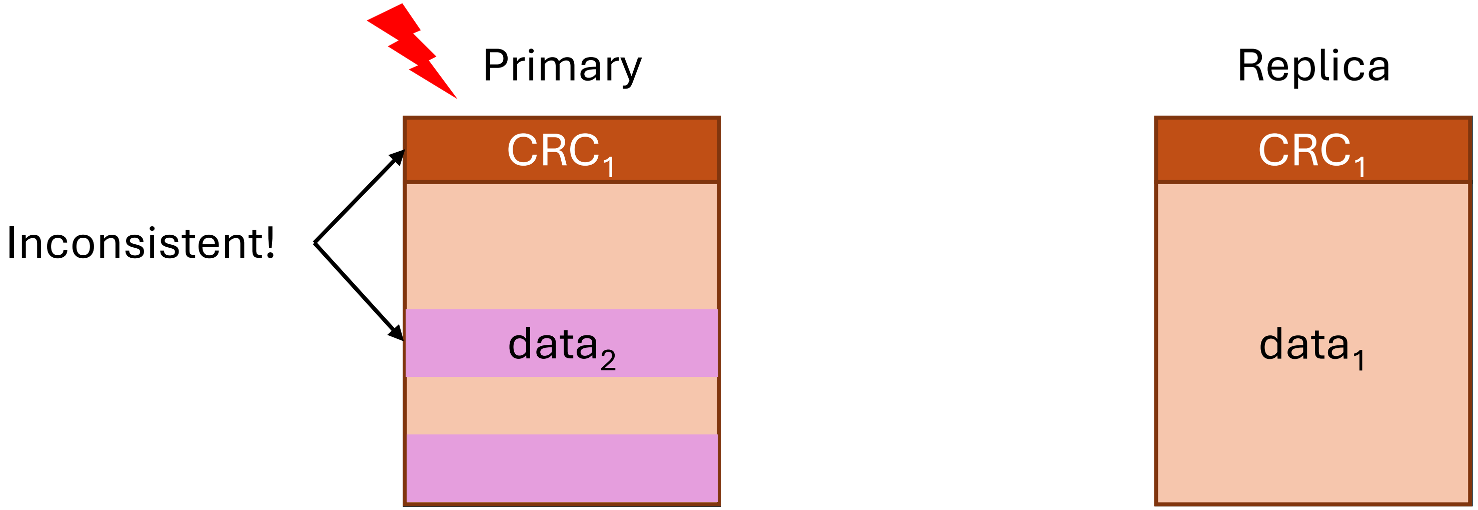
Prior work: Tick-Tock algorithm

Introduced in NOVA-Fortis file system (Xu SOSP ’17)



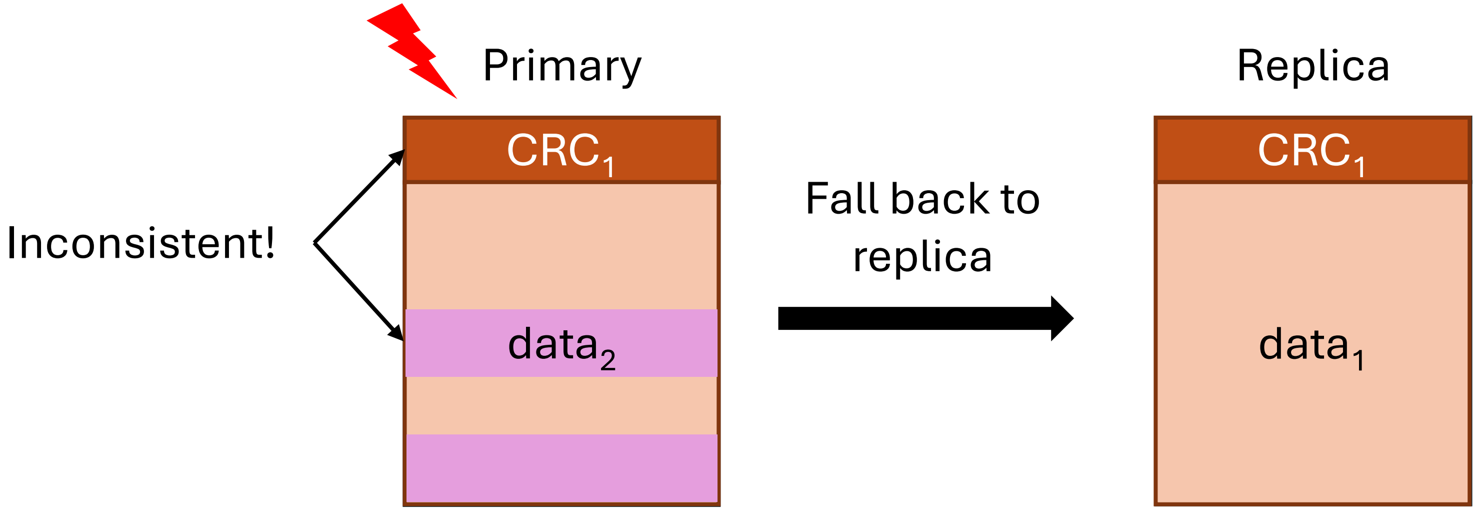
Prior work: Tick-Tock algorithm

Introduced in NOVA-Fortis file system (Xu SOSP '17)



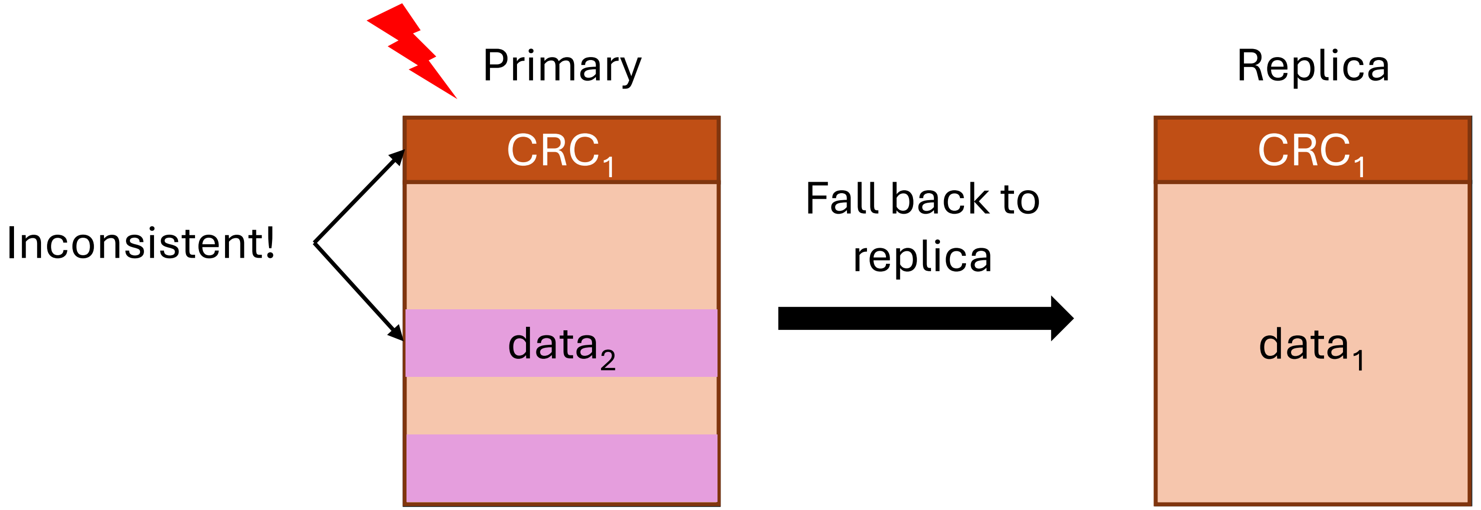
Prior work: Tick-Tock algorithm

Introduced in NOVA-Fortis file system (Xu SOSP ’17)



Prior work: Tick-Tock algorithm

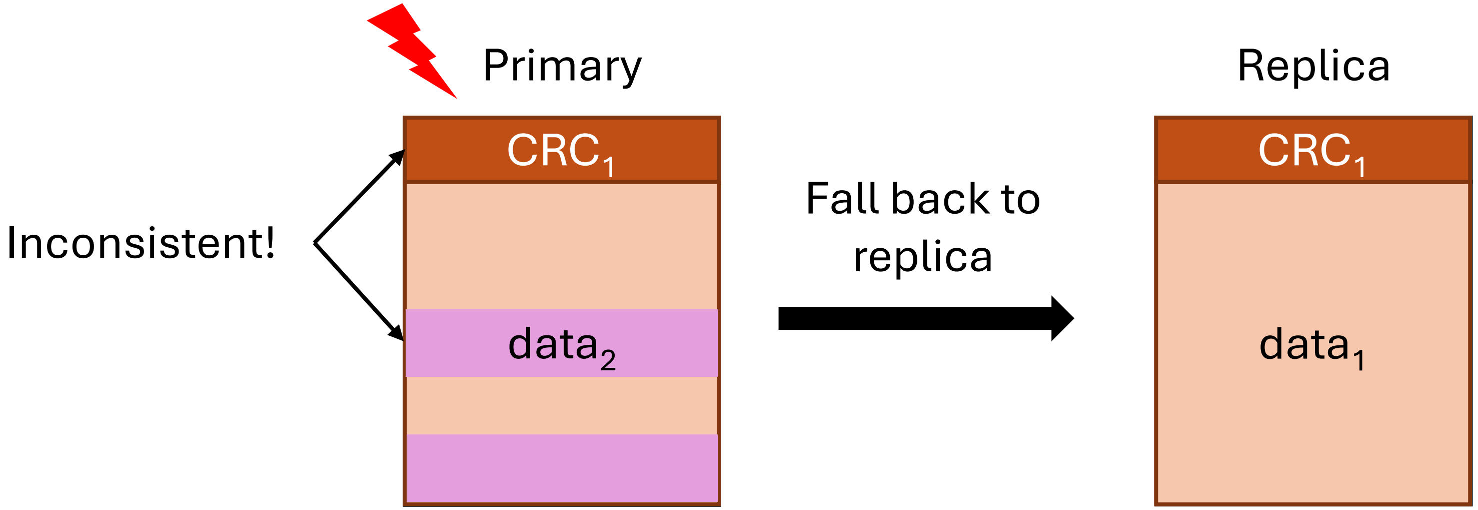
Introduced in NOVA-Fortis file system (Xu SOSP '17)



Our prior work found CRC atomicity bugs in NOVA-Fortis (LeBlanc EuroSys '23)

Prior work: Tick-Tock algorithm

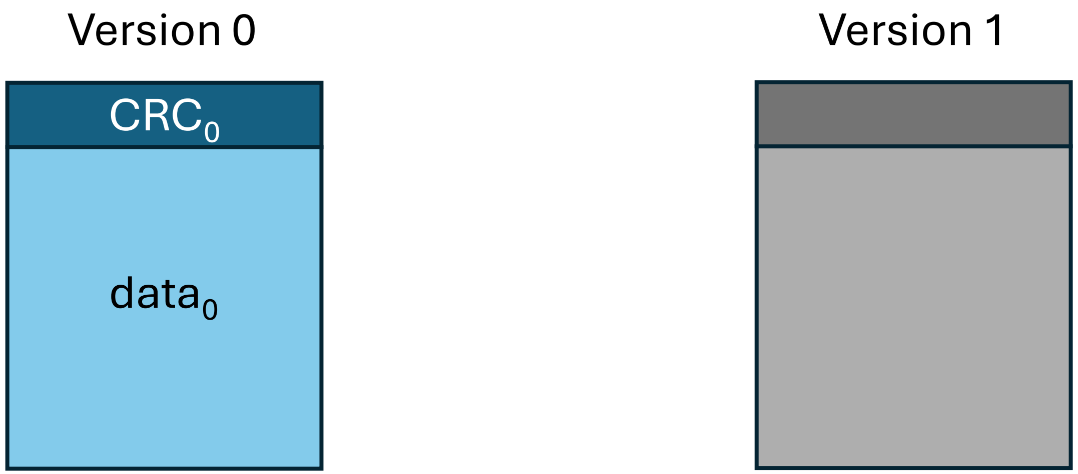
Introduced in NOVA-Fortis file system (Xu SOSP '17)



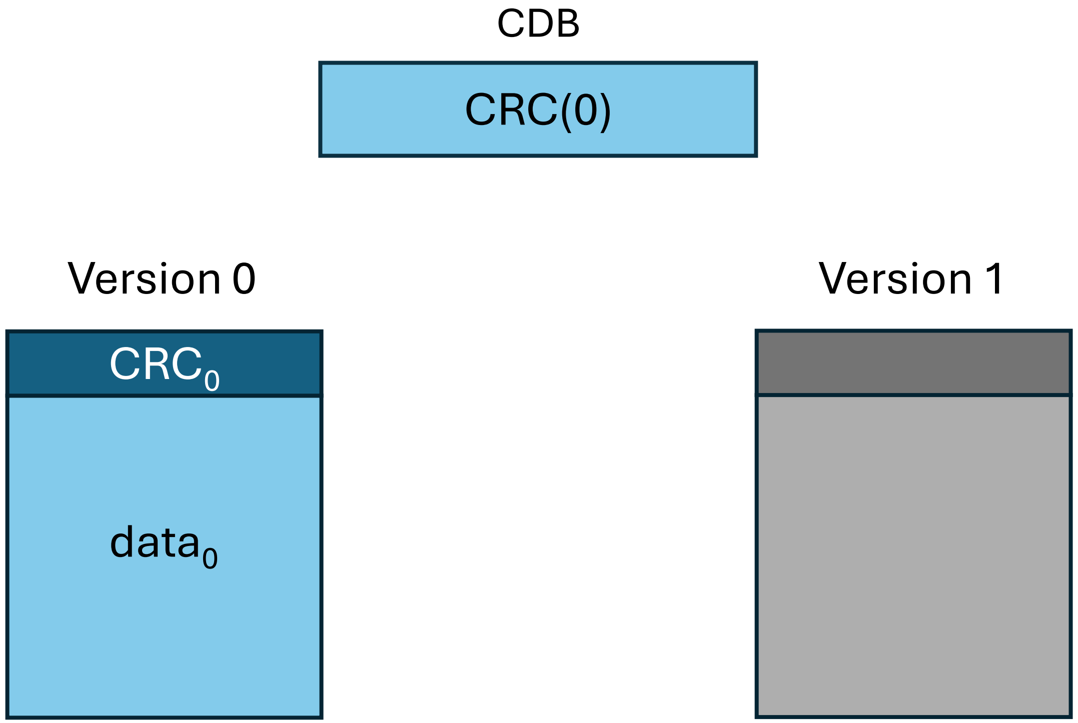
Our prior work found CRC atomicity bugs in NOVA-Fortis (LeBlanc EuroSys '23)

CRCs designed to detect random bit flips, not torn writes

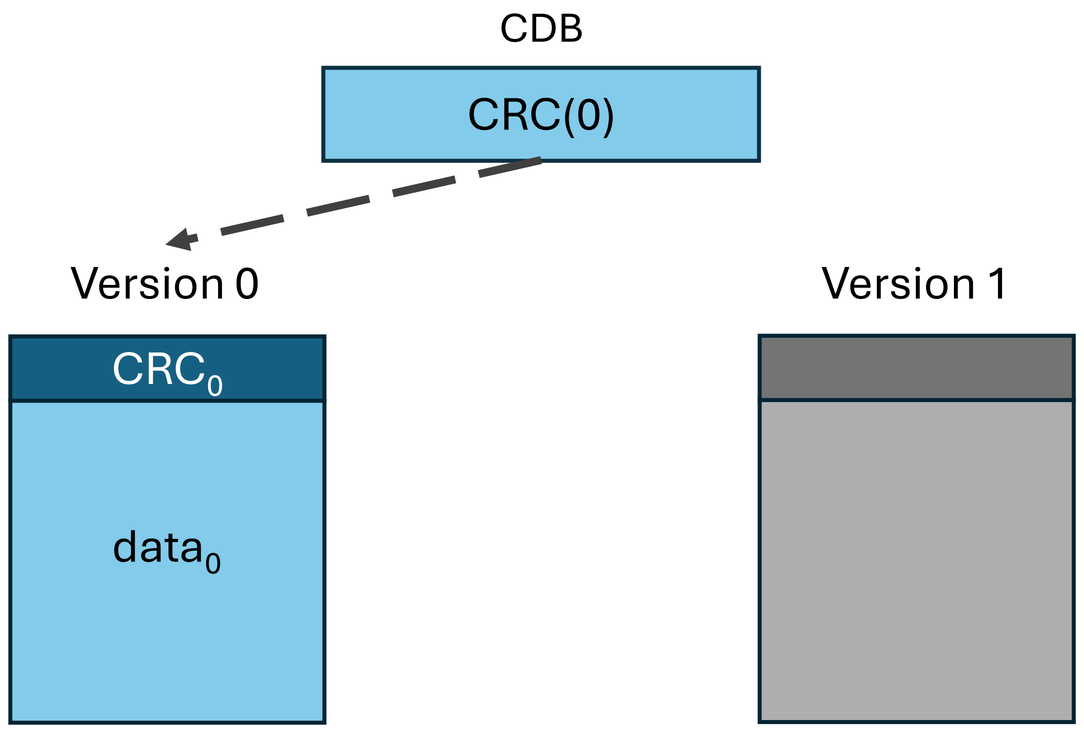
Corruption-detecting Boolean example



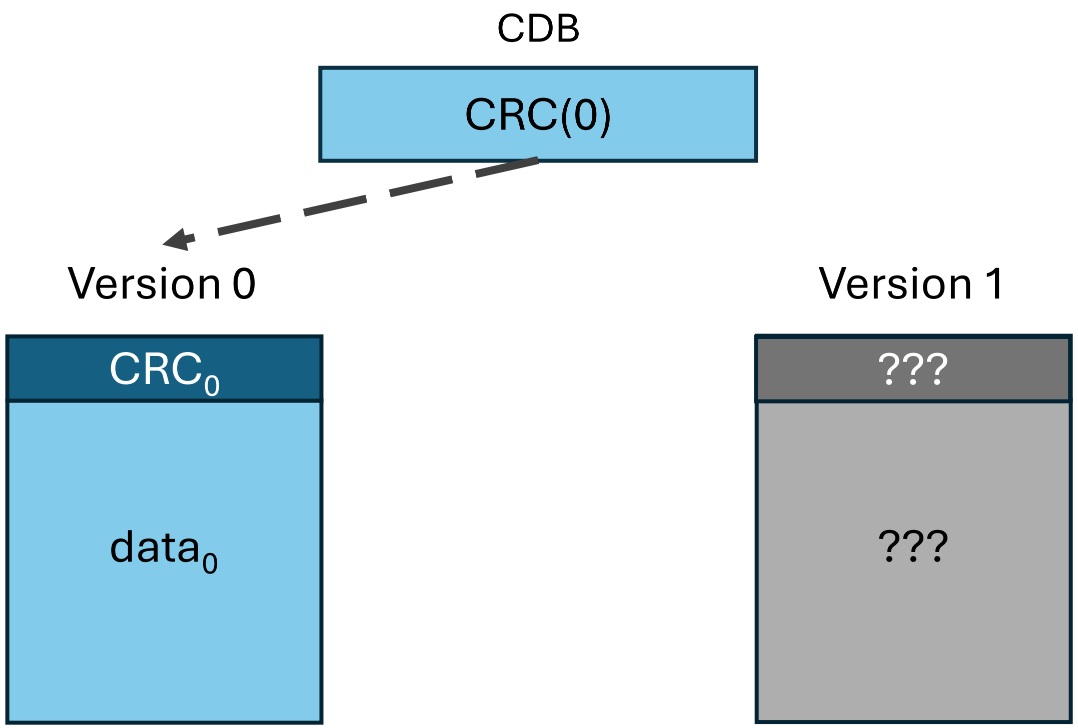
Corruption-detecting Boolean example



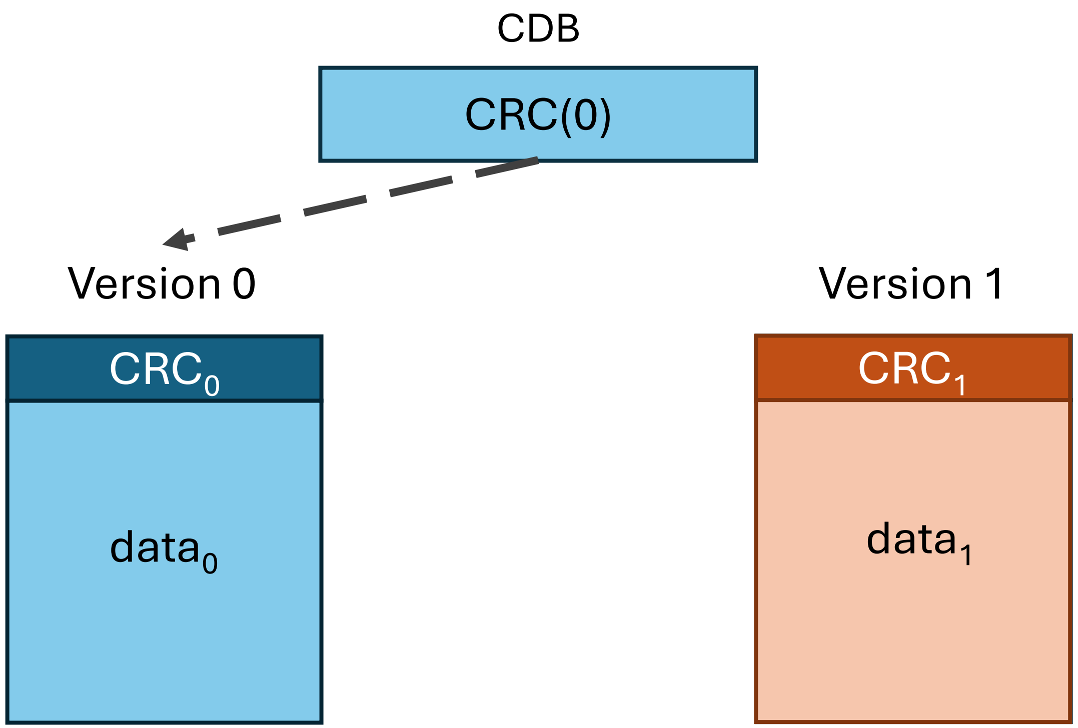
Corruption-detecting Boolean example



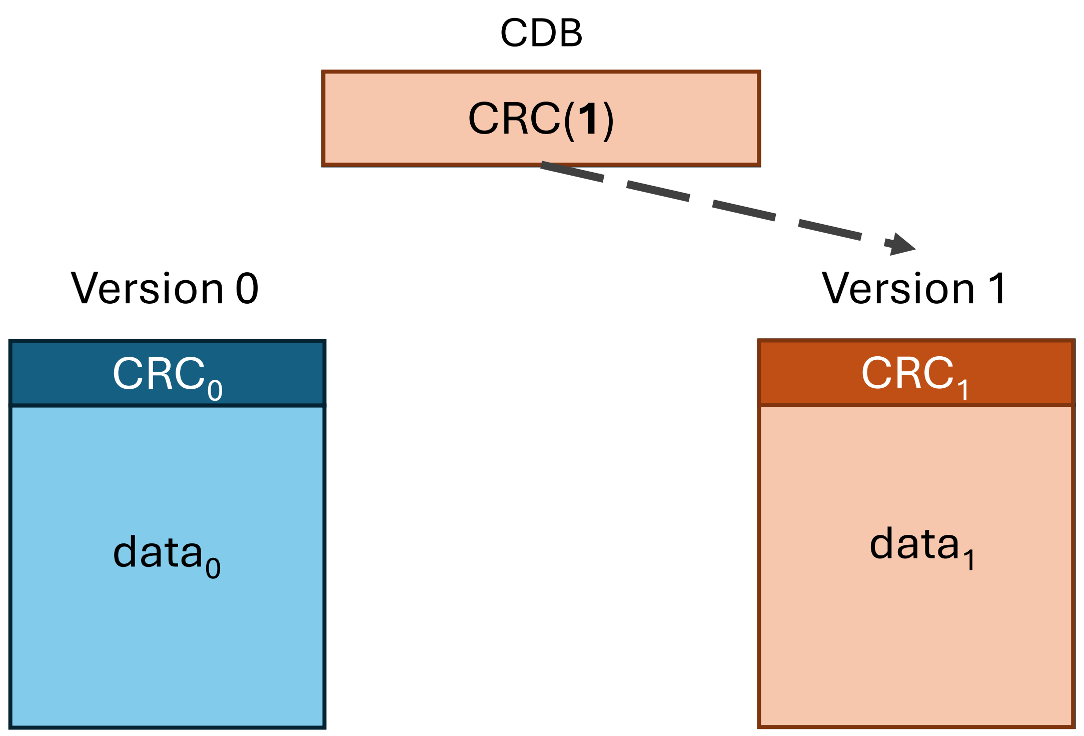
Corruption-detecting Boolean example



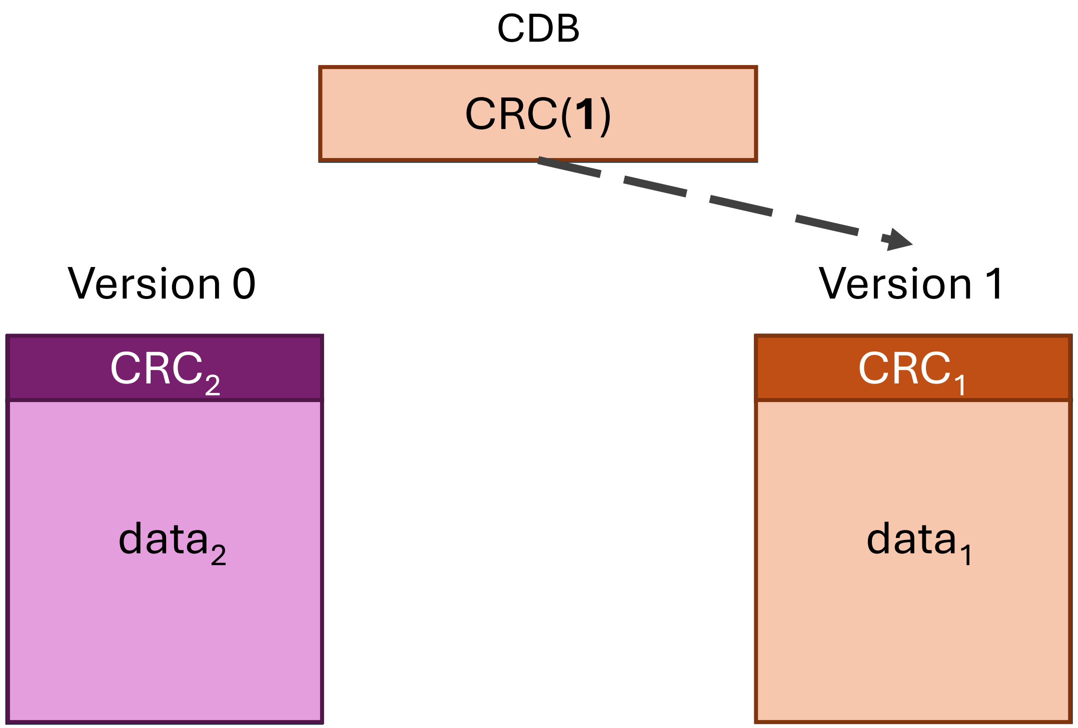
Corruption-detecting Boolean example



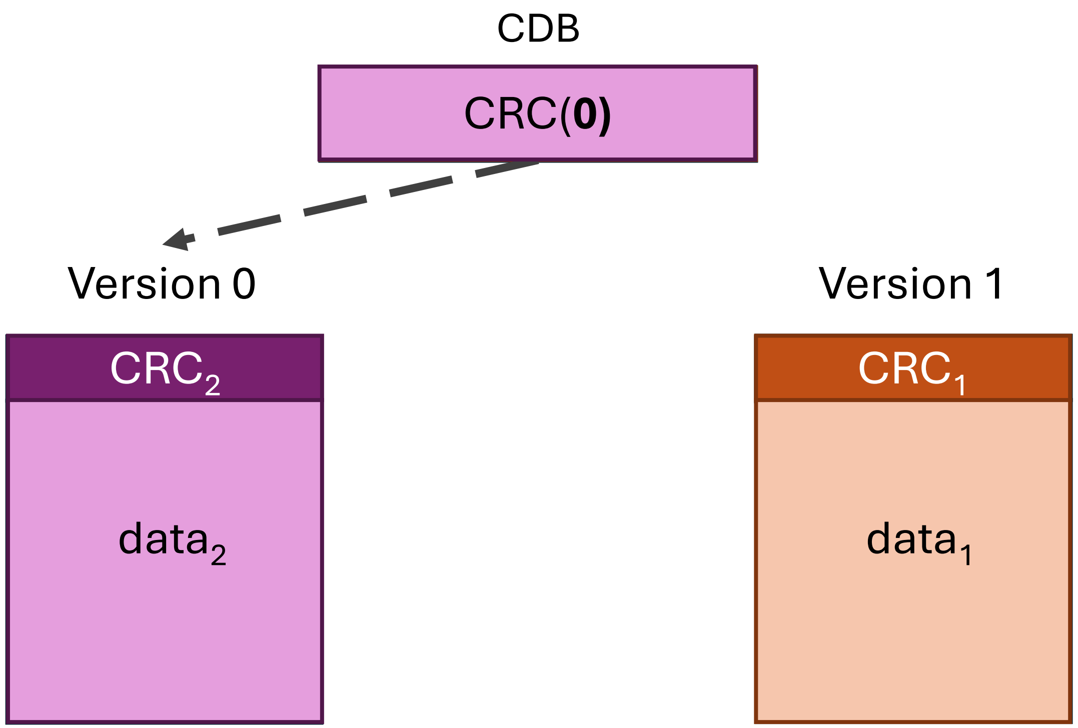
Corruption-detecting Boolean example



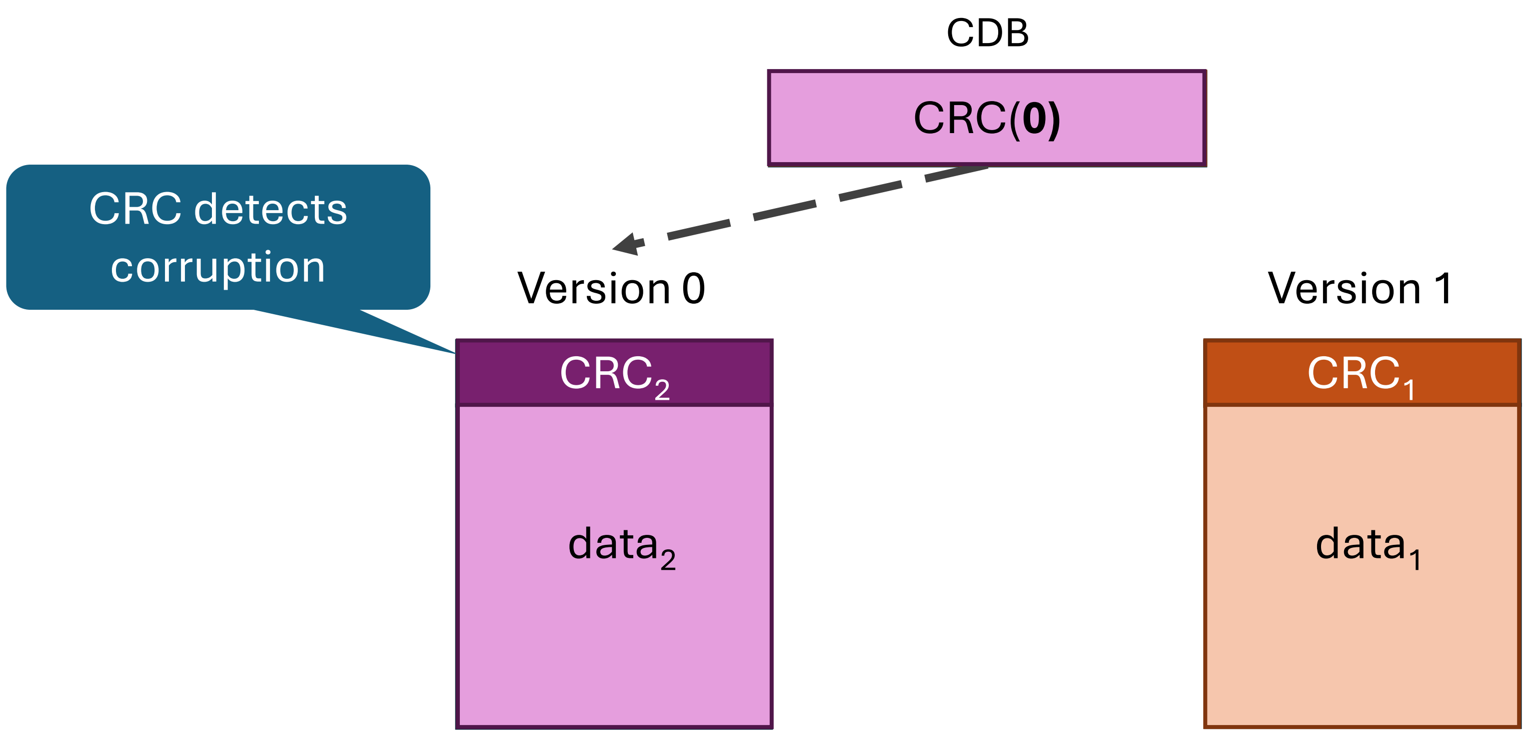
Corruption-detecting Boolean example



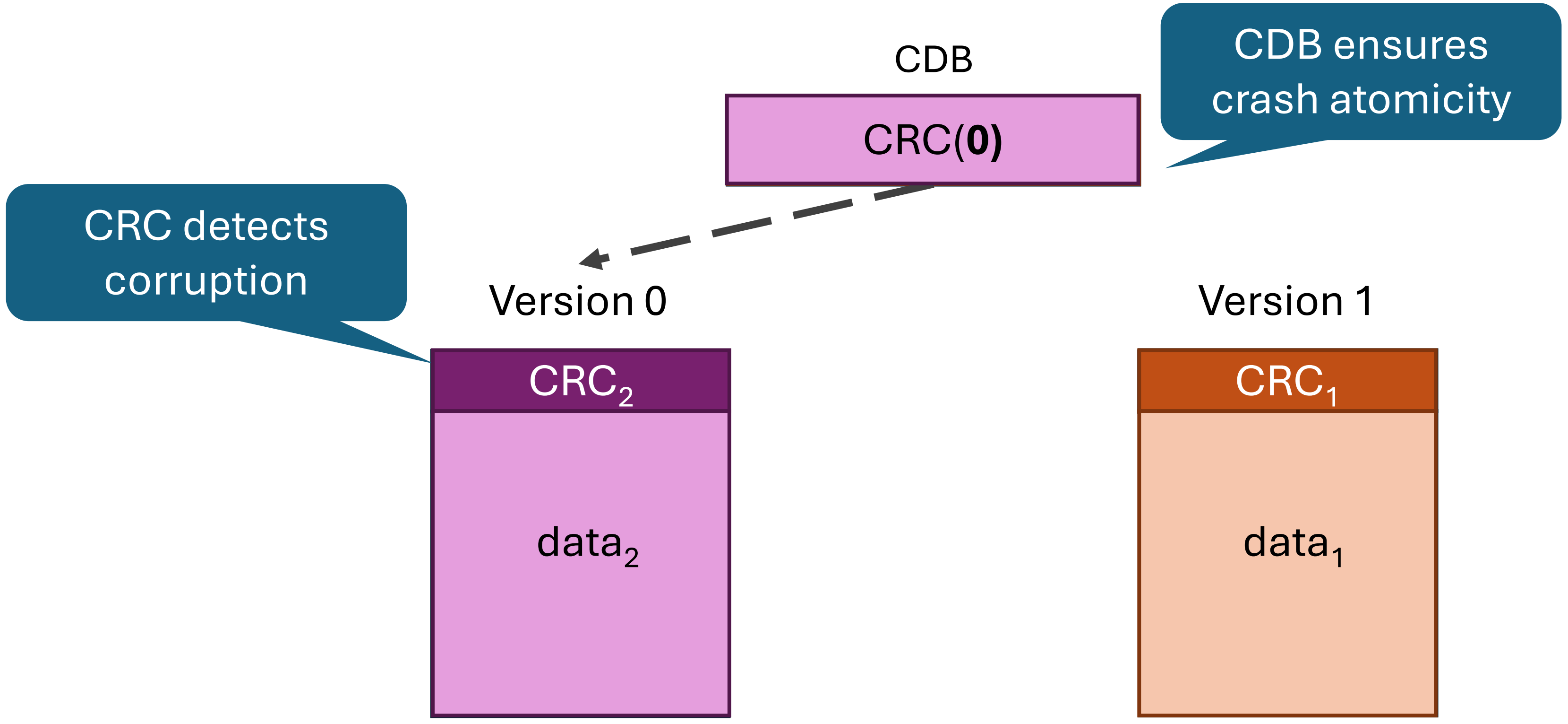
Corruption-detecting Boolean example



Corruption-detecting Boolean example



Corruption-detecting Boolean example



Using properties of CRC algorithms

CRC algorithms are engineered to *always* detect a certain number c of flipped bits! (Koopman 2024)

We can definitively prove the absence of up to c bits of corruption

Using properties of CRC algorithms

Depends on length of byte sequence; always ≥ 1

CRC algorithms are engineered to *always* detect a certain number c of flipped bits! (Koopman 2024)

We can definitively prove the absence of up to c bits of corruption

Using properties of CRC algorithms

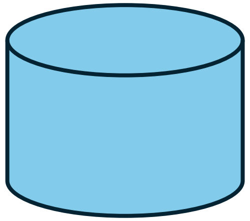
Depends on length of byte sequence; always ≥ 1

CRC algorithms are engineered to *always* detect a certain number c of flipped bits! (Koopman 2024)

We can definitively prove the absence of up to c bits of corruption

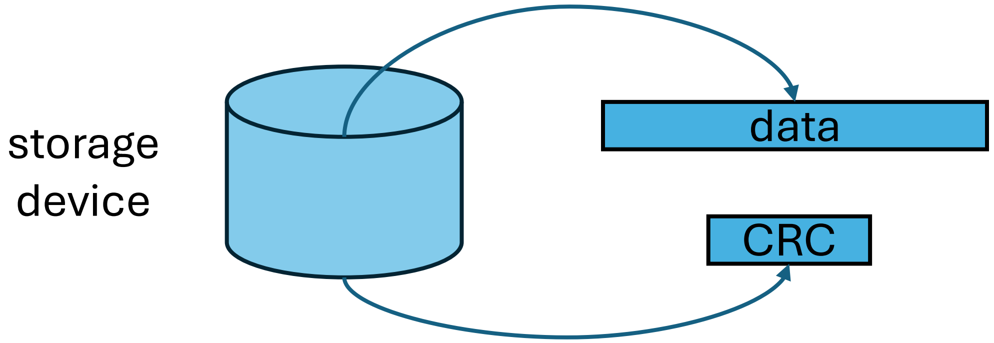
CRC check fails if and only if $[1, c]$ bits are corrupted

New corruption model

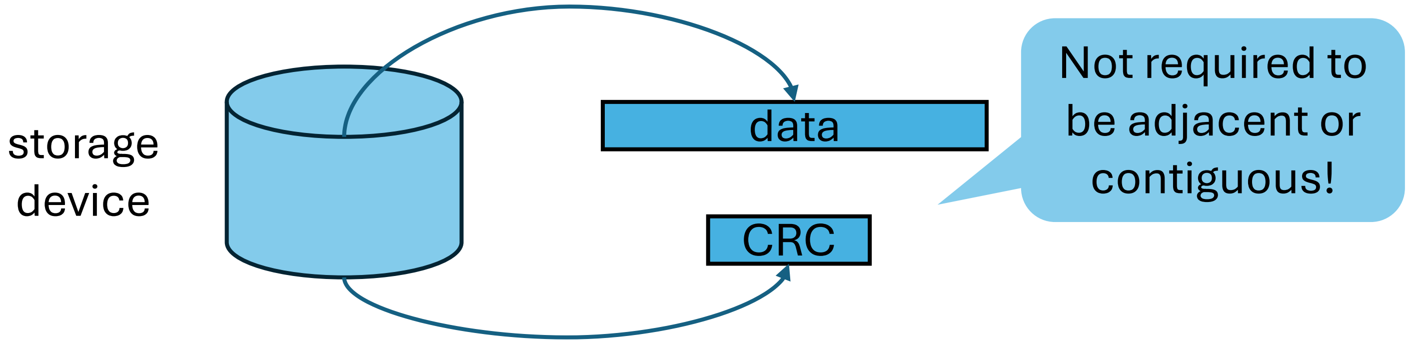


storage
device

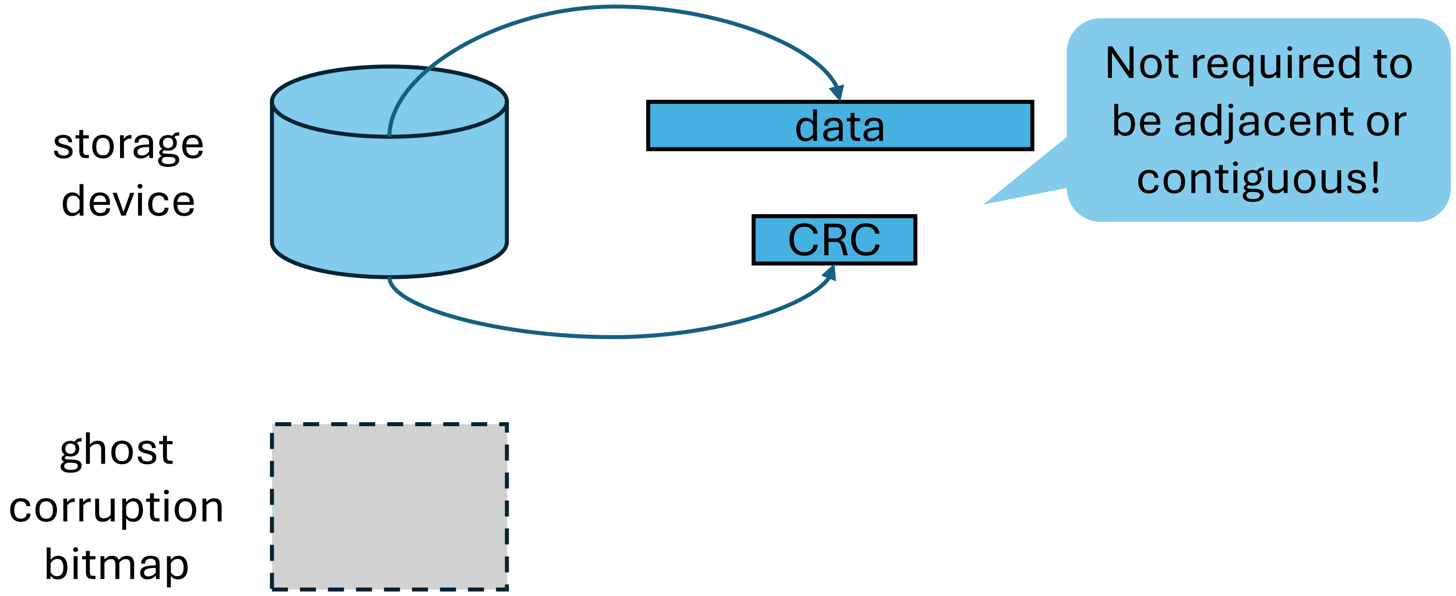
New corruption model



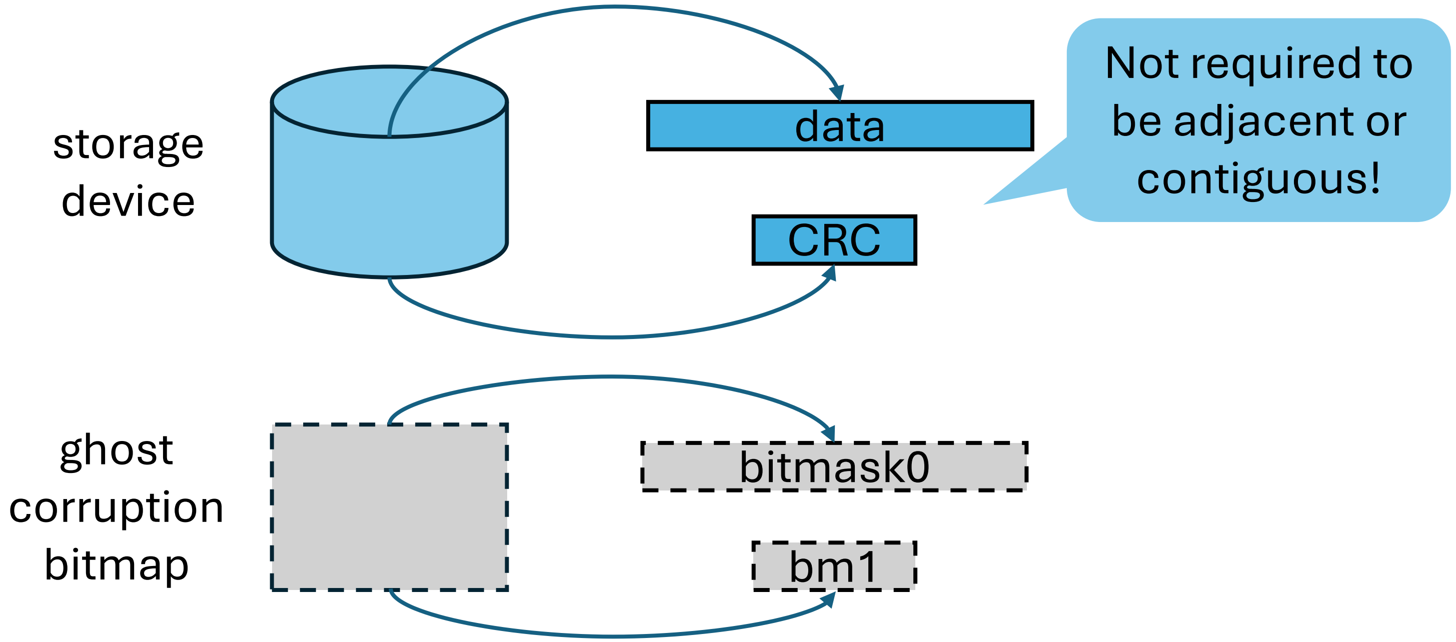
New corruption model



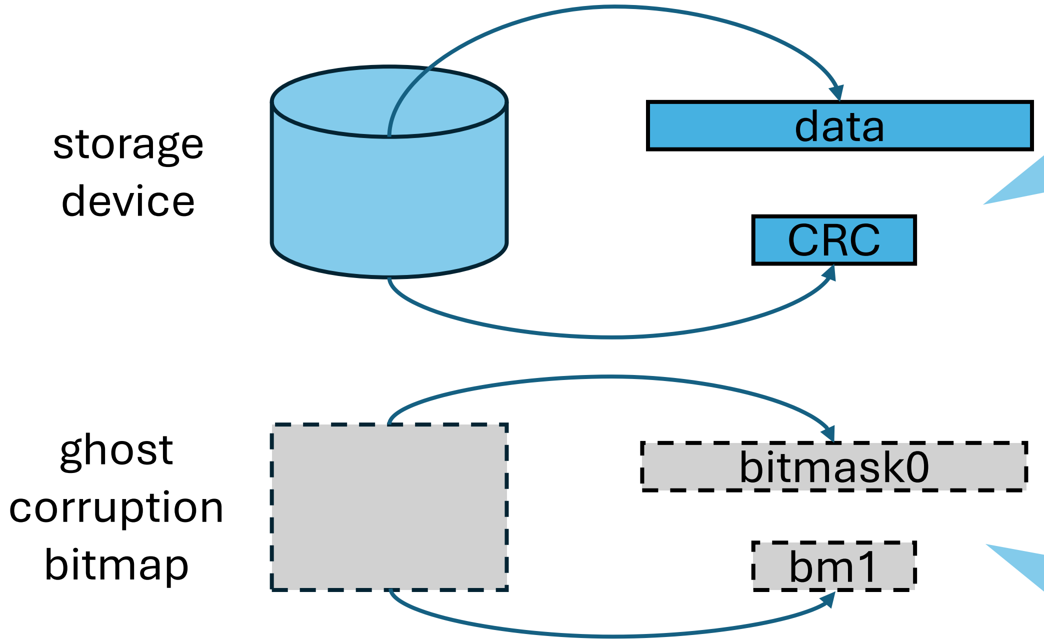
New corruption model



New corruption model



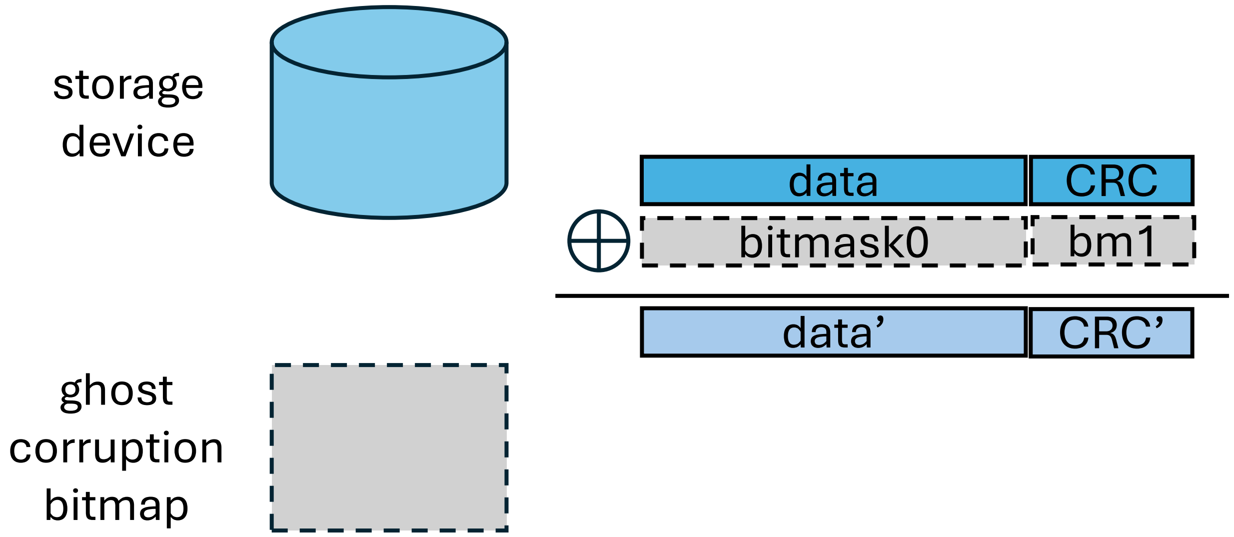
New corruption model



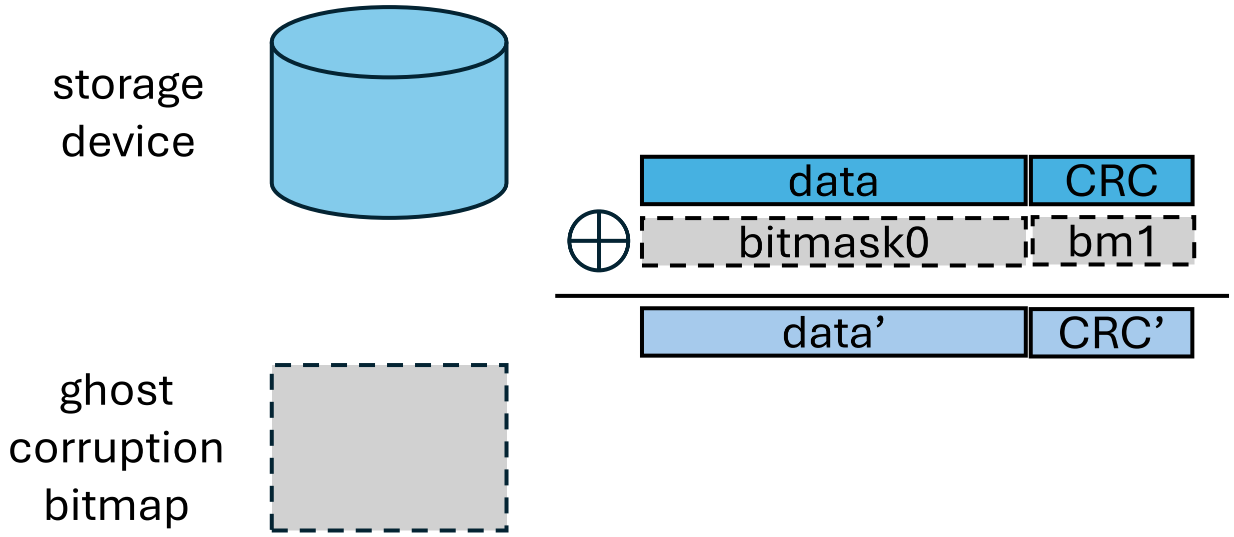
Not required to be adjacent or contiguous!

CRC algorithms are engineered to detect up to c bit errors

New corruption model

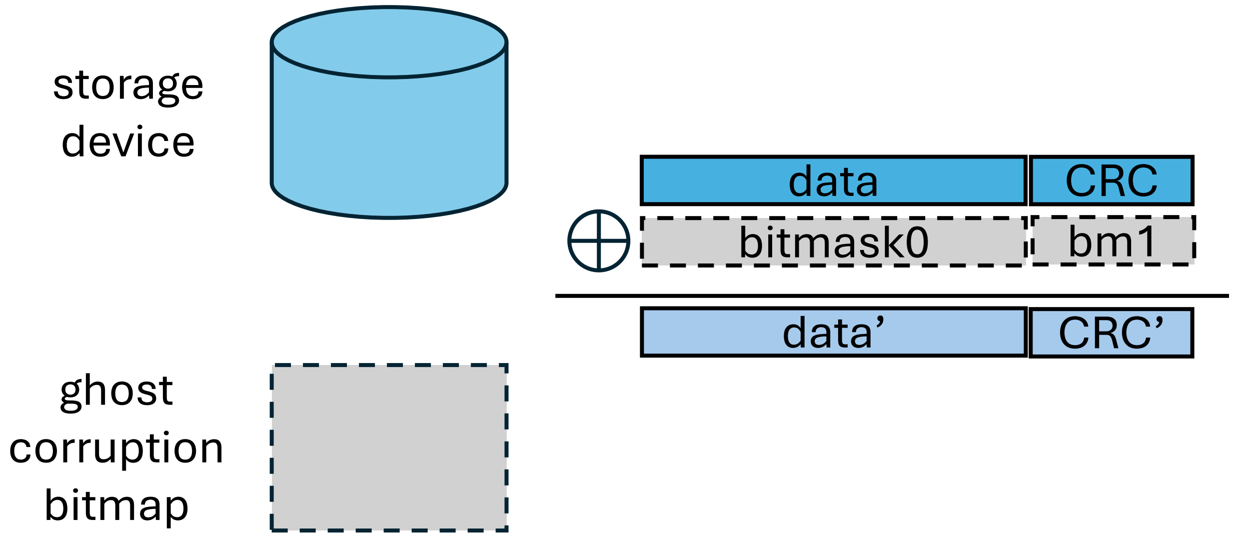


New corruption model



If bitmask contains $[1, c]$ bit flips, then CRC' does *not* match data'

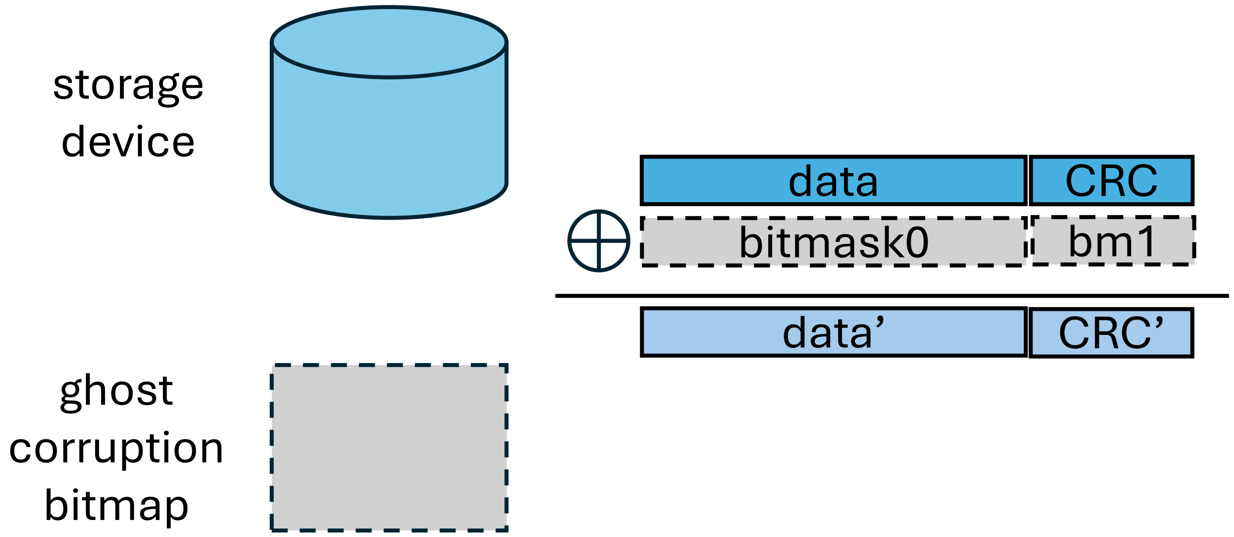
New corruption model



If bitmask contains $[1, c]$ bit flips, then CRC’ does *not* match data’

Assuming up to c bit flips, if CRC check passes, data’ is not corrupted

New corruption model



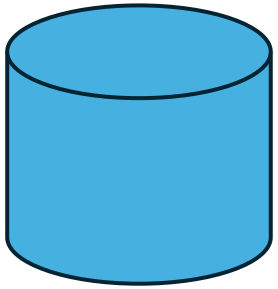
See paper for:

- Reasoning about corruption on byte-addressable storage
- New primitive for CRC management on PM

If bitmask contains $[1, c]$ bit flips, then CRC' does *not* match data'

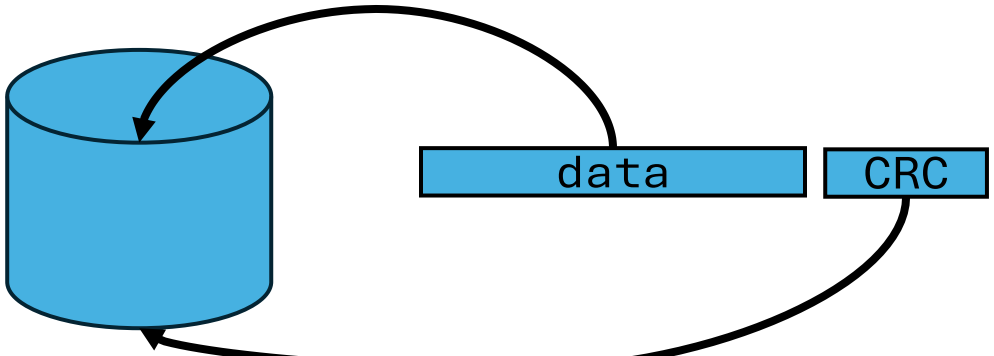
Assuming up to c bit flips, if CRC check passes, data' is not corrupted

New corruption model

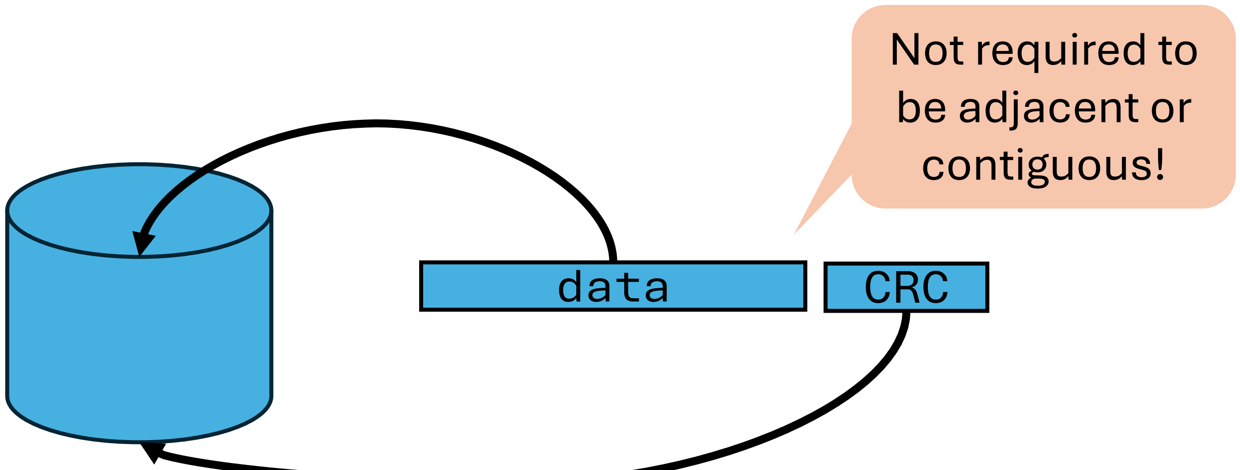


New corruption model

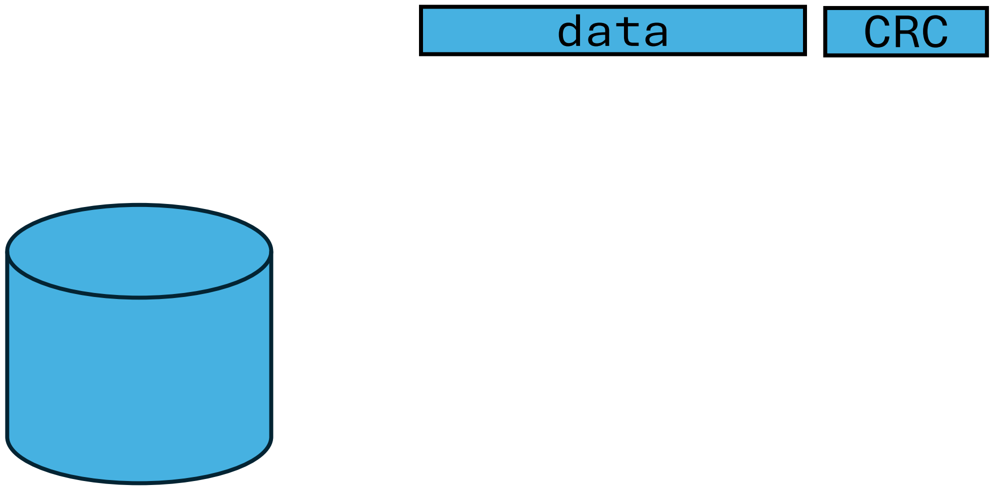
New corruption model



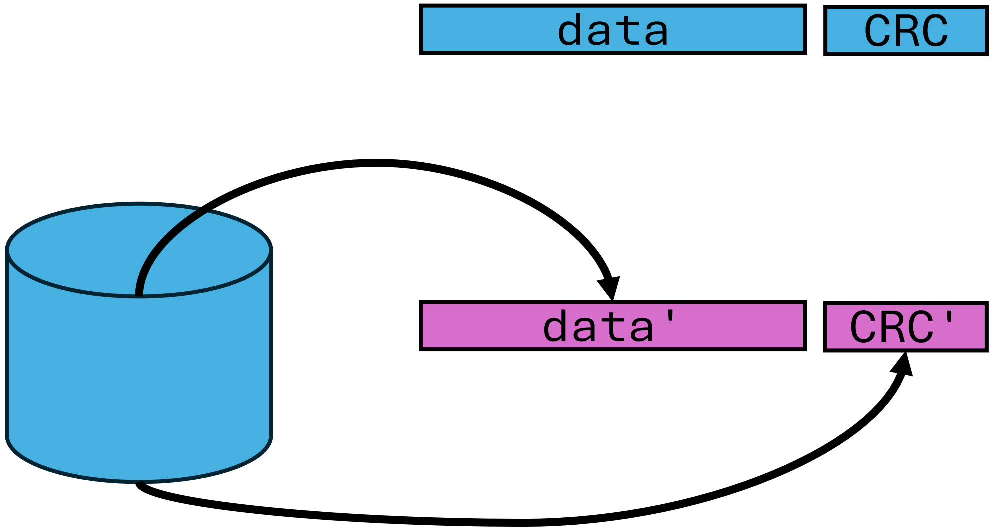
New corruption model



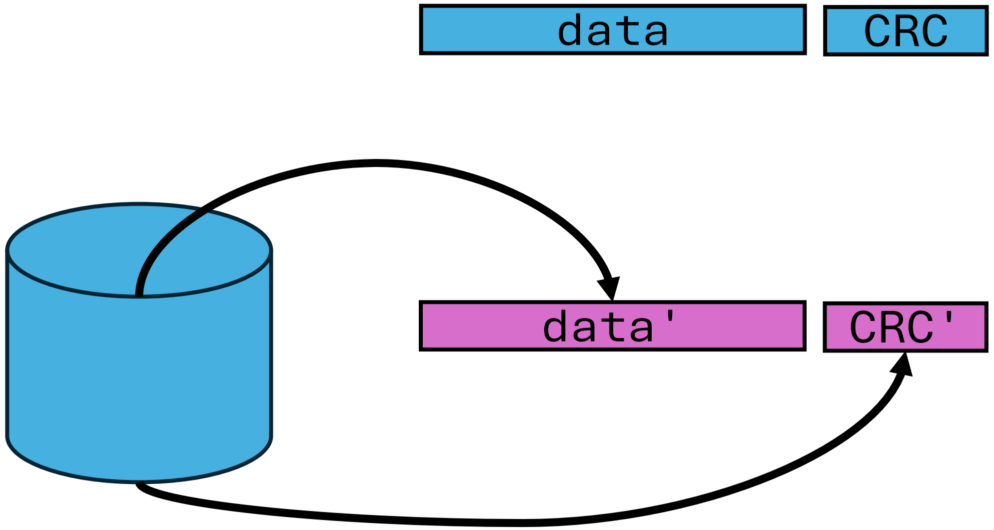
New corruption model



New corruption model

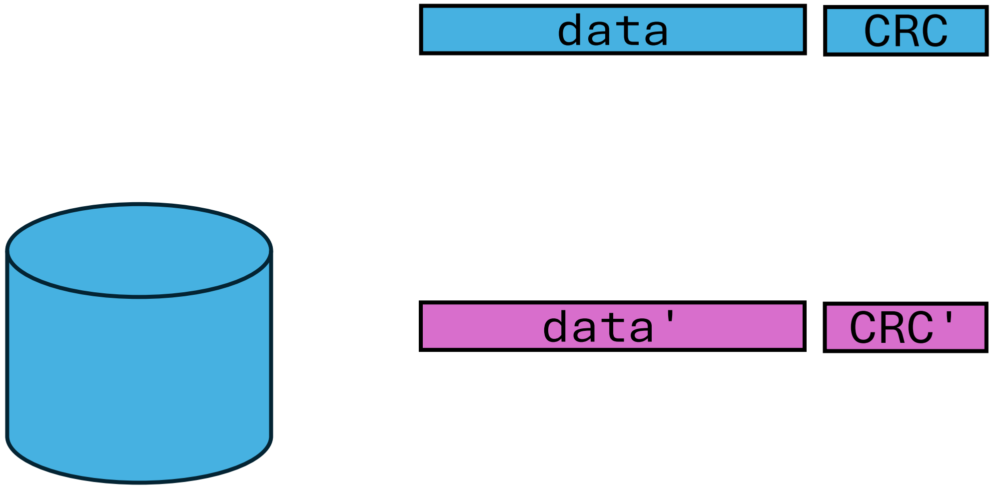


New corruption model



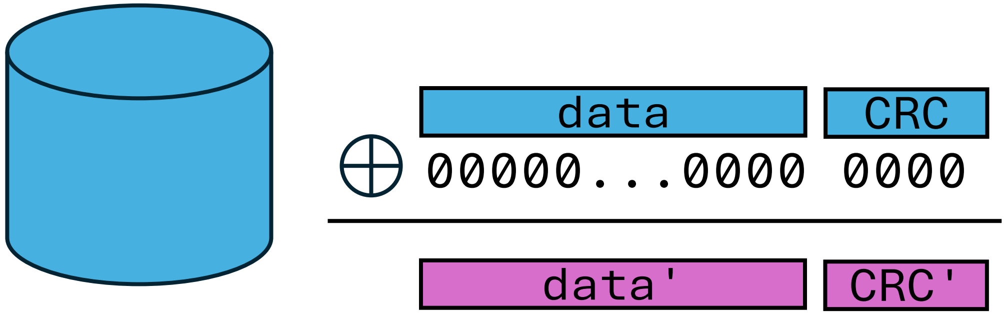
How are `data`/`data'` and `CRC`/`CRC'` related? How do we reason about this relationship?

New corruption model



How are `data`/`data'` and `CRC`/`CRC'` related? How do we reason about this relationship?

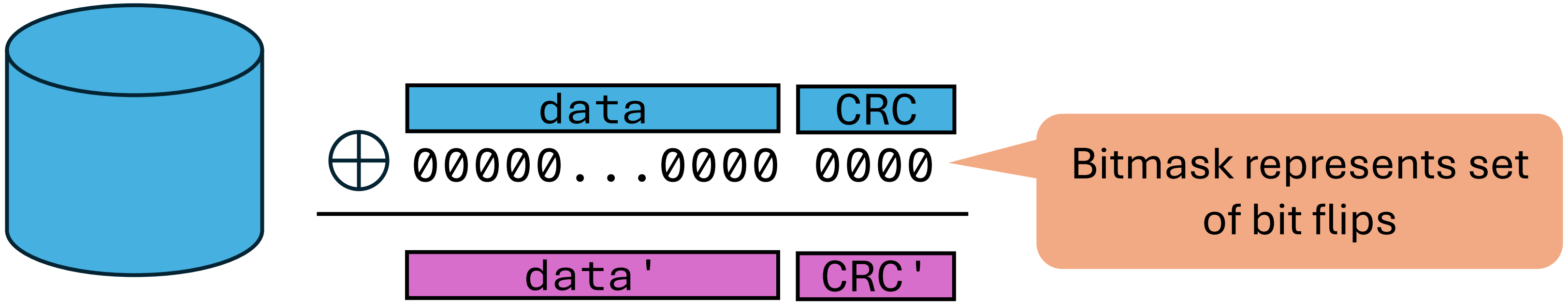
New corruption model



How are `data`/`data'` and CRC/`CRC'` related? How do we reason about this relationship?

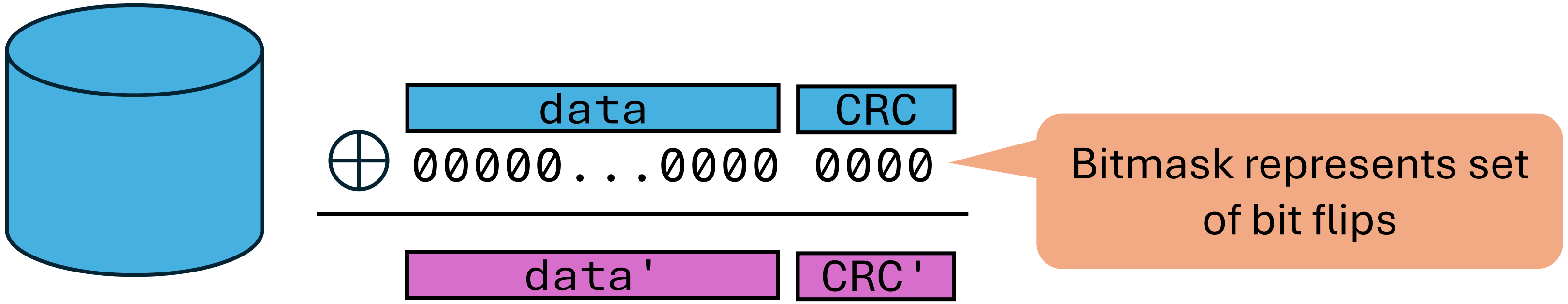
New corruption model

How are `data`/`data'` and `CRC`/`CRC'` related? How do we reason about this relationship?



New corruption model

How are `data`/`data'` and `CRC`/`CRC'` related? How do we reason about this relationship?

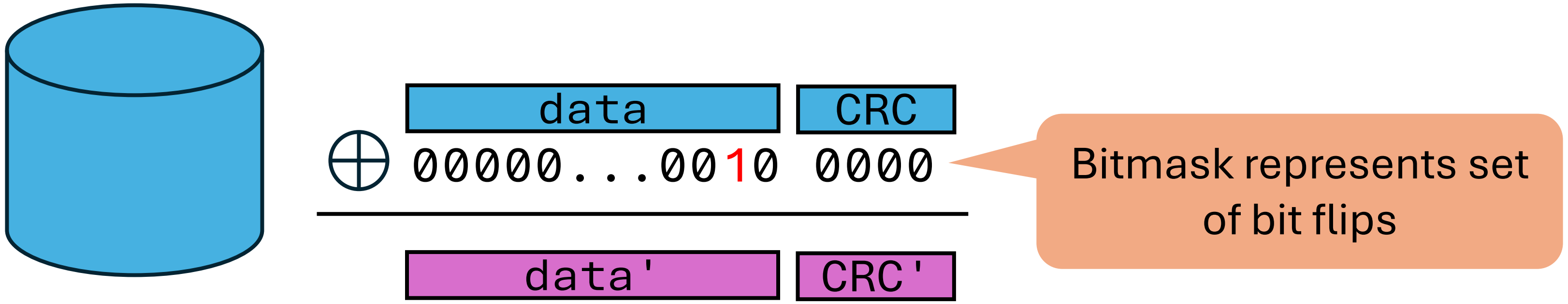


Guarantees:

- No bit flips ==> `CRC' == crc(data')`

New corruption model

How are `data`/`data'` and `CRC`/`CRC'` related? How do we reason about this relationship?

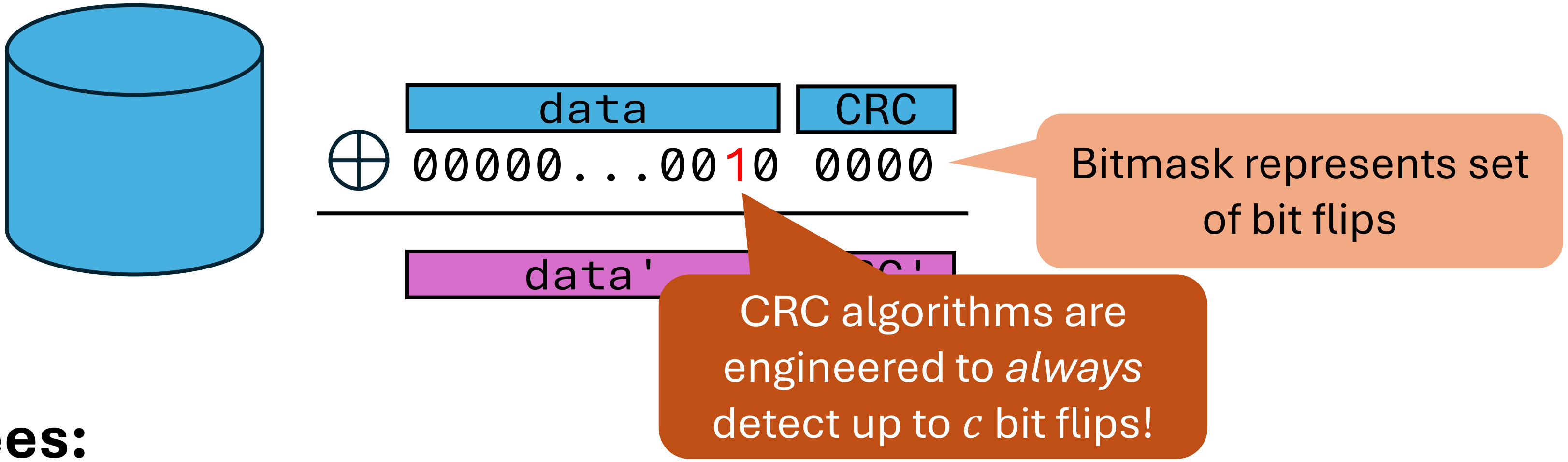


Guarantees:

- No bit flips ==> `CRC' == crc(data')`

New corruption model

How are `data`/`data'` and `CRC`/`CRC'` related? How do we reason about this relationship?

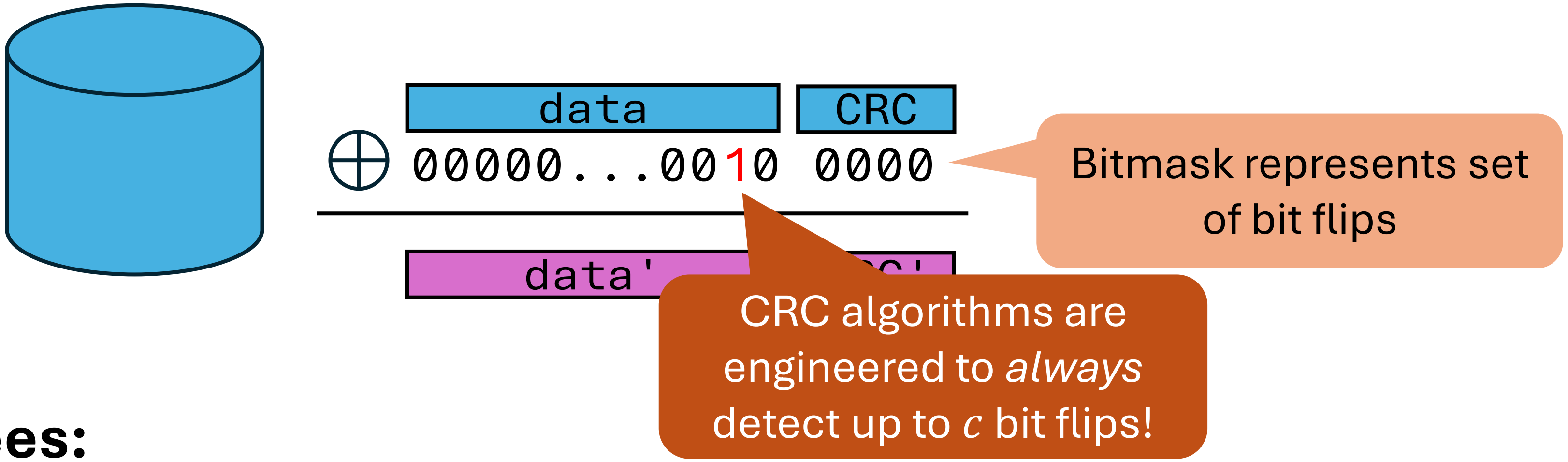


Guarantees:

- No bit flips ==> `CRC' == crc(data')`

New corruption model

How are `data`/`data'` and `CRC`/`CRC'` related? How do we reason about this relationship?

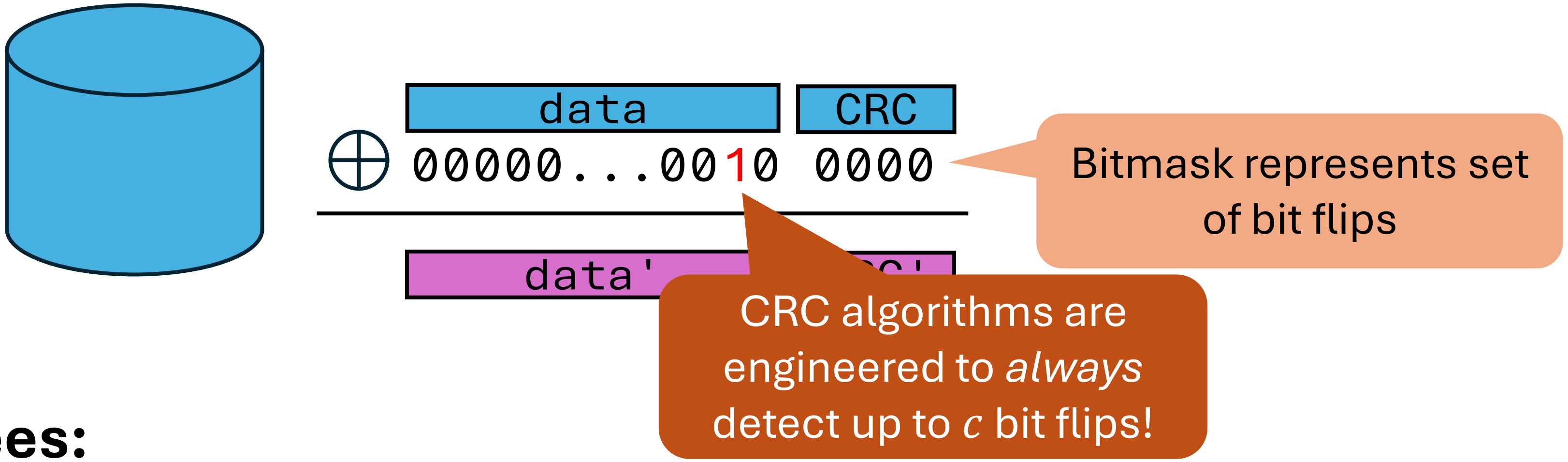


Guarantees:

- No bit flips ==> `CRC' == crc(data')`
- $1 \leq$ Population count of bitmask $\leq c$ ==> `CRC' != crc(data')`

New corruption model

How are `data`/`data'` and `CRC`/`CRC'` related? How do we reason about this relationship?



Guarantees:

- No bit flips ==> `CRC' == crc(data')`
- $1 \leq$ Population count of bitmask $\leq c$ ==> `CRC' != crc(data')`

Assuming $\leq c$ bit flips, CRC check proves whether data has been corrupted!