Verifying a Realistic Mutable Hash Table

Case Study

Samuel Chassot & Viktor Kunčak - 05.07.2024

Introduction

Mostly functional, immutable data structures verified

Mutable data structures

- Ubiquitous in practice
- Fundamental to applications
- But challenging to verify!

$\Rightarrow$ **Mutable data structures are ubiquitous and fundamental**
$\rightarrow$ **need verification**

Stainless: Automated Proof

Verification framework for Scala

```scala
def zip(xs: List[Int], ys: List[Boolean]): List[(Int, Boolean)] = {
  require(xs.size <= ys.size)
  (xs, ys) match
    case (Cons(x, xs0), Cons(y, ys0)) =>
           Cons((x, y), zip(xs0, ys0))

    case _ => Nil()
}.ensuring (res => res.map(p => p._1) == xs)
```

```
warning: Found counter-example:
warning:   xs: List[Int]         -> Cons[Int](0, Nil[Int]())
           ys: List[Boolean] -> Nil[Boolean]()
```

Stainless: Proof by Induction

```scala
/**
  * Proves that inserting a new pair does not change
    the presence of another key, nor its value.
  */
def lemma[B](l: List[(Long, B)], key: Long, v: B, oKey: Long): Unit = {
    require(invariant(l) && key != oKey)

    l match
          case Cons(hd, tl) if (hd._1 != oKey) => lemma(tl, key, v, oKey)
          case _ => ()

  }.ensuring(_ =>
        containsKey(insert(l, key, v), oKey) == containsKey(l, oKey)
        && lookup(insert(l, key, v), oKey) == lookup(l, oKey)
     )
```

Stainless: Proof by Induction

```scala
/**
  * Proves that inserting a new pair does not change
    the presence of another key, nor its value.
  */
def lemma[B](l: List[(Long, B)], key: Long, v: B, oKey: Long): Unit = {
    require(invariant(l) && key != oKey)

    l match
      case Cons(hd, tl) if (hd._1 != oKey) => lemma(tl, key, v, oKey)
      case _ =>
        assert(containsKey(l, oKey) == containsKey(insert(l, key, v), oKey))
        assert(lookup(insert(l, key, v), oKey) == lookup(l, oKey))

  }.ensuring(_ =>
    containsKey(insert(l, key, v), oKey) == containsKey(l, oKey)
    && lookup(insert(l, key, v), oKey) == lookup(l, oKey)
  )
```

LongMap Open Addressing

LongMap Interface

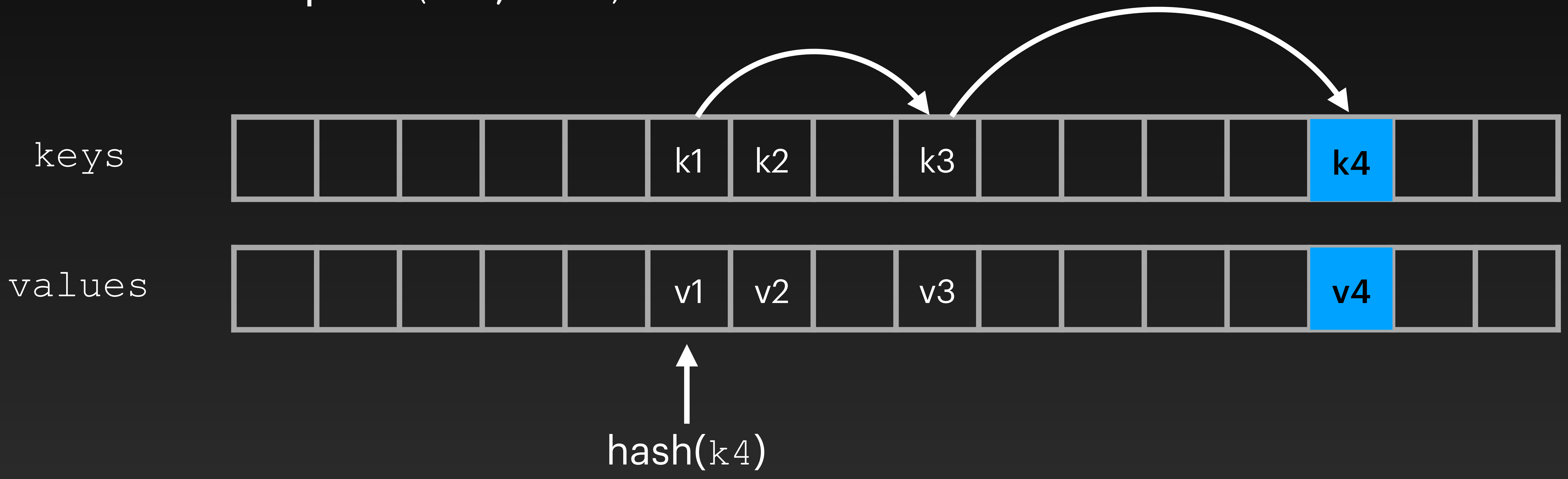
Hash Table, 64-bit Integer Keys

```
trait LongMap[V]:
  def contains(key: Long): Boolean
  def apply(key: Long): V                // Lookup
  def update(key: Long, v: V): Boolean
  def remove(key: Long): Boolean
  def repack(): Boolean
```

LongMap Hash Table

64-bit keys, open addressing, non-linear probing

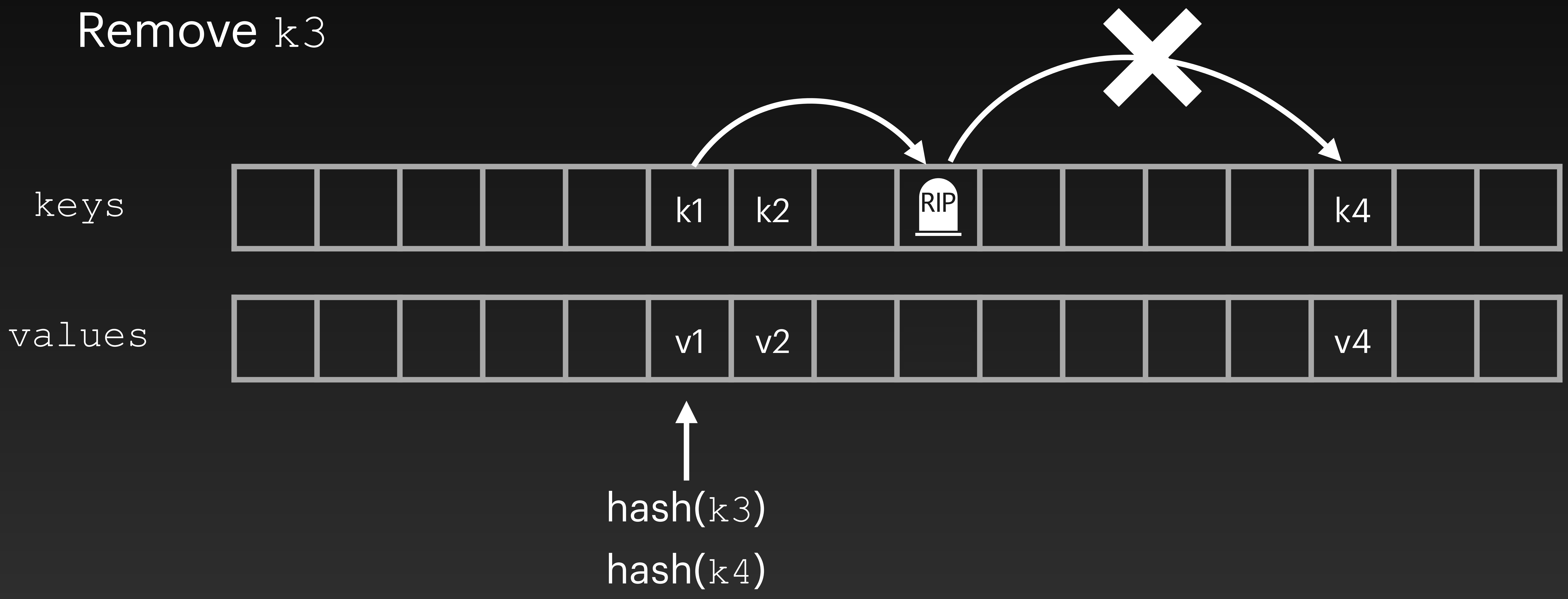
Insert the pair `(k4, v4)`



LongMap Hash Table

64-bit keys, open addressing, non-linear probing

Remove `k3`



Implementation changes for verification

Adapting for verification

Summary

Refactor while loops to tail recursion

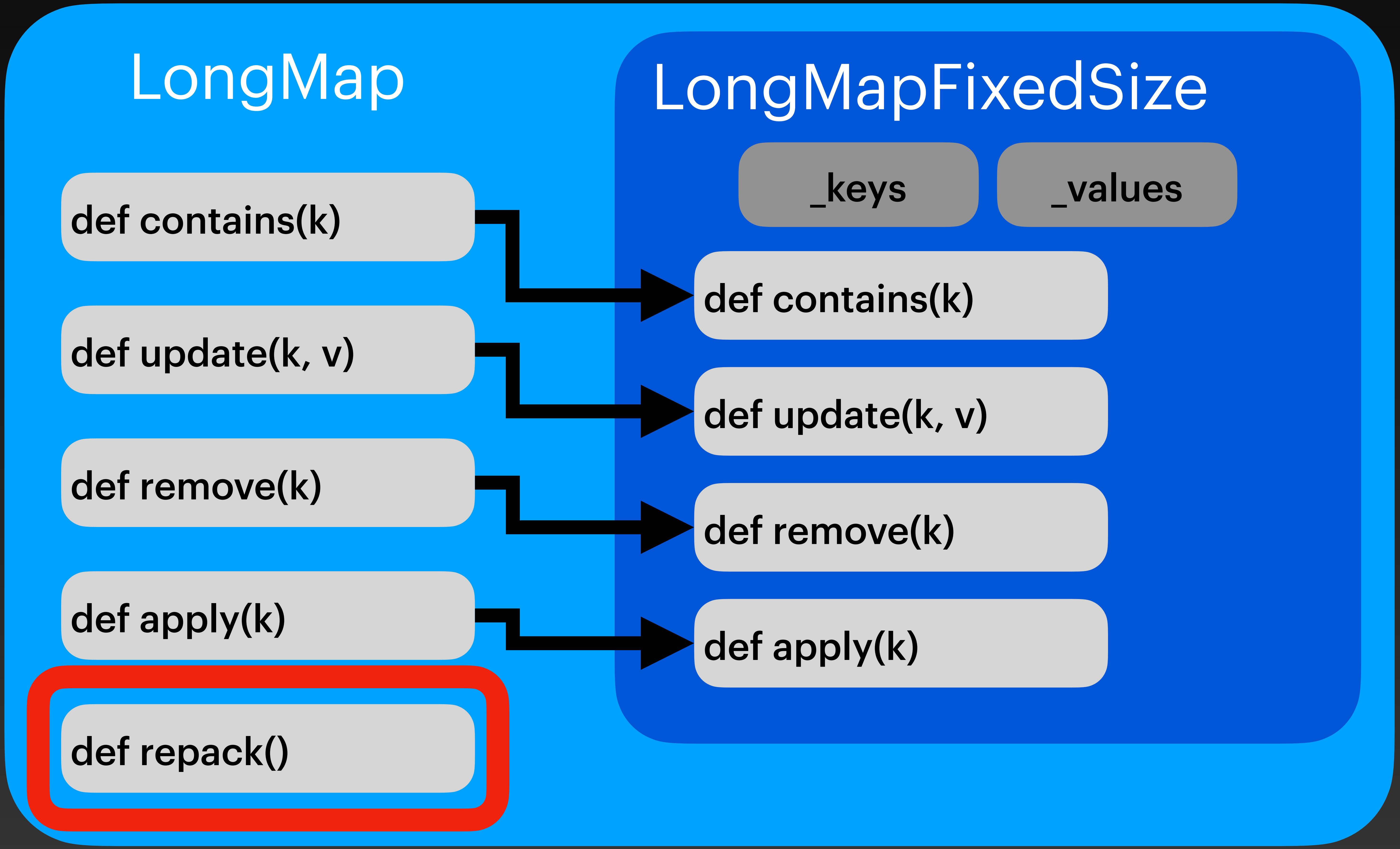
Add loop counter checks to prove termination

Typing and initialisation of arrays → new level of indirection

Refactor applying the decorator design pattern

Adapting for verification

Decorator Pattern



⇒ **Modular proof without compromising performance**

Repack

Algorithm pseudo code

```
// Resize arrays and rebalance keys (pseudocode)
def repack() =
    val size = this.computeArraySize()
    val newMap = new LongMapFixedSize(size)
    for k, v <- this do
        newMap.update(k, v)
    this.underlying = newMap
```

Aliasing!

- Stainless disallows this kind of aliasing!
- We introduce a new structure: `Cell`

Cell & Swap Operation

Aliasing in repack

```
class Cell[T](v: T):
  def swap(other: Cell[T])
  def v(): T


// Resize arrays and rebalance keys (pseudocode)
def repack() =
  val size = this.computeArraySize()
  val newMap = Cell(new LongMapFixedSize(size))
  for k, v <- this do
    newMap.v().update(k, v)
  this.underlying.swap(newMap)
```

⇒ **Greater expressiveness without introducing aliasing**

Verification effort

Specification

ListLongMap Interface

```scala
trait ListLongMap[B](toList: List[(Long, B)]) {

  def contains(key: Long): Boolean

  def get(key: Long): Option[B]

  def apply(key: Long): B

  def +(keyValue: (Long, B)): ListLongMap[B]

  def -(key: Long): ListLongMap[B]
}
```

⇒ Executable specification → better proof and readability

Specification

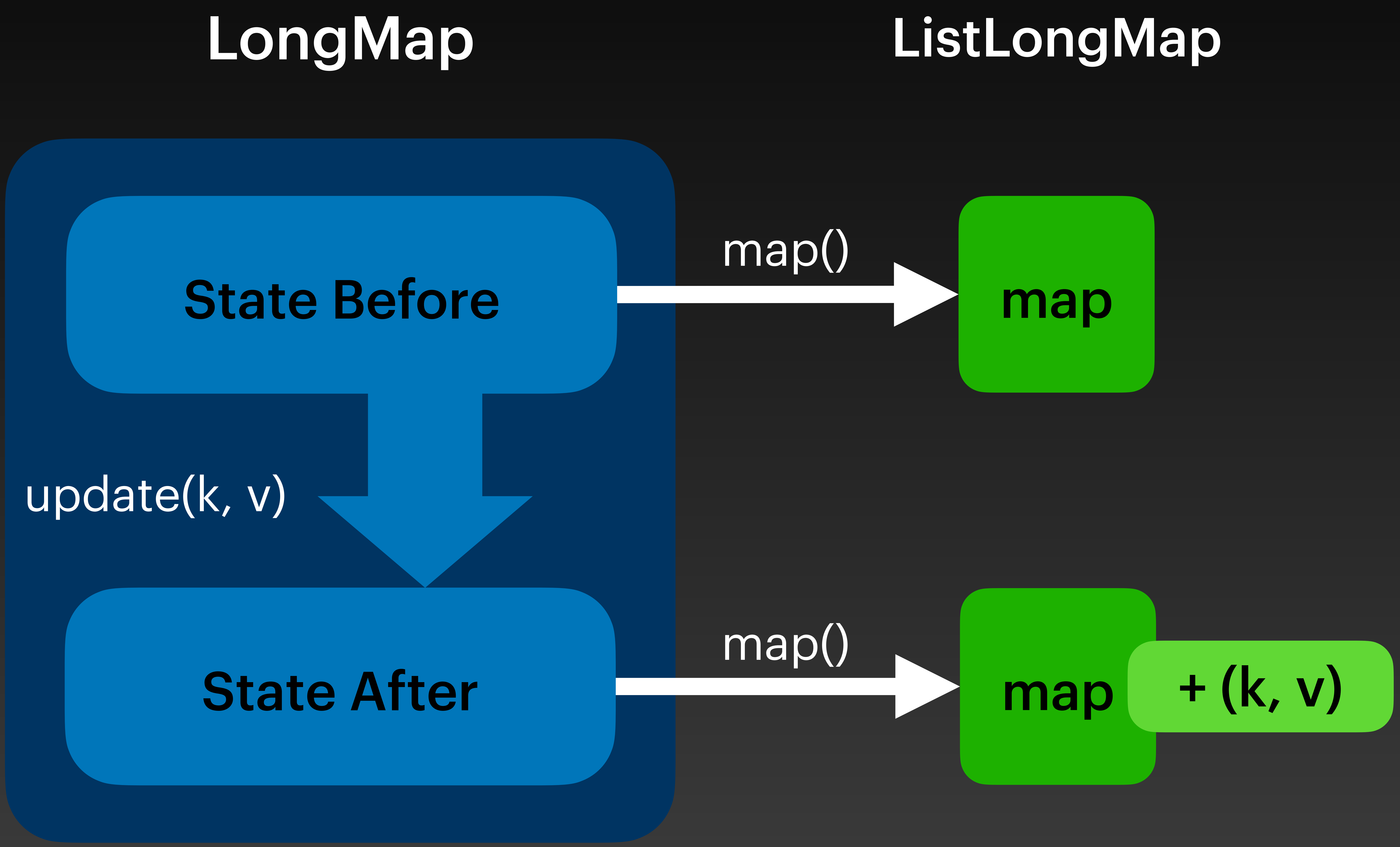
ListLongMap

```
def addStillContains[B](
  lm: ListLongMap[B],
  a: Long,
  b: B,
  a0: Long
): Unit = {
  require(lm.contains(a0))
  // ...
} ensuring(_ =>
 (lm + (a, b)).contains(a0)
)
```

```
def addApplyDifferent[B](
    lm: ListLongMap[B],
    a: Long,
    b: B,
    a0: Long
): Unit = {
  require(lm.contains(a0) && a0 != a)
  // ...
} ensuring(_ =>
 (lm + (a -> b))(a0) == lm(a0)
)
```

Verification using abstraction function

Proof Structure



Verification

Proof Structure

```
// add or update an existing binding
def update(key: Long, v: V): Boolean = {
  require(valid)
  val repacked = if (imbalanced()) {
    repack()
  } else {
    true
  }
  if (repacked) {
    underlying.v.update(key, v)
 } else {
    false
 }
} ensuring (res =>
 valid &&
 (if (res) map.contains(key) &&
 (map == old(this).map + (key, v)) else map == old(this).map))
```

Bug in Deployed Implementation

Bug in the Original Implementation

New size computation

```
// Compute the new size for the array based on map’s state
def computeNewMask(mask: Int, _size: Int, _vacant: Int) = {
  var m = mask
    if (2 * (_size + _vacant) >= mask && !(5 * _vacant > mask)) {
   m = ((m << 1) + 1) & IndexMask
  }
   while (m > 8 && 8 * _size < m) {
     m = m >>> 1
   }
   m
}
```

`8*_size` **overflows** → `m` is too small to accommodate all pairs

De Boer, De Gouw, Klamroth, Jung, Ulbrich, Weigl: Formal Specification and Verification of JDK’s Identity Hash Map Implementation. Formal Aspects of Computing 2023

Performance analysis

Statistics

Lines of Code

Class	Program LOC	Proof + Specification LOC	Total LOC
ListLongMap	156	678	834
MutableLongMap	409	7'358	7'767

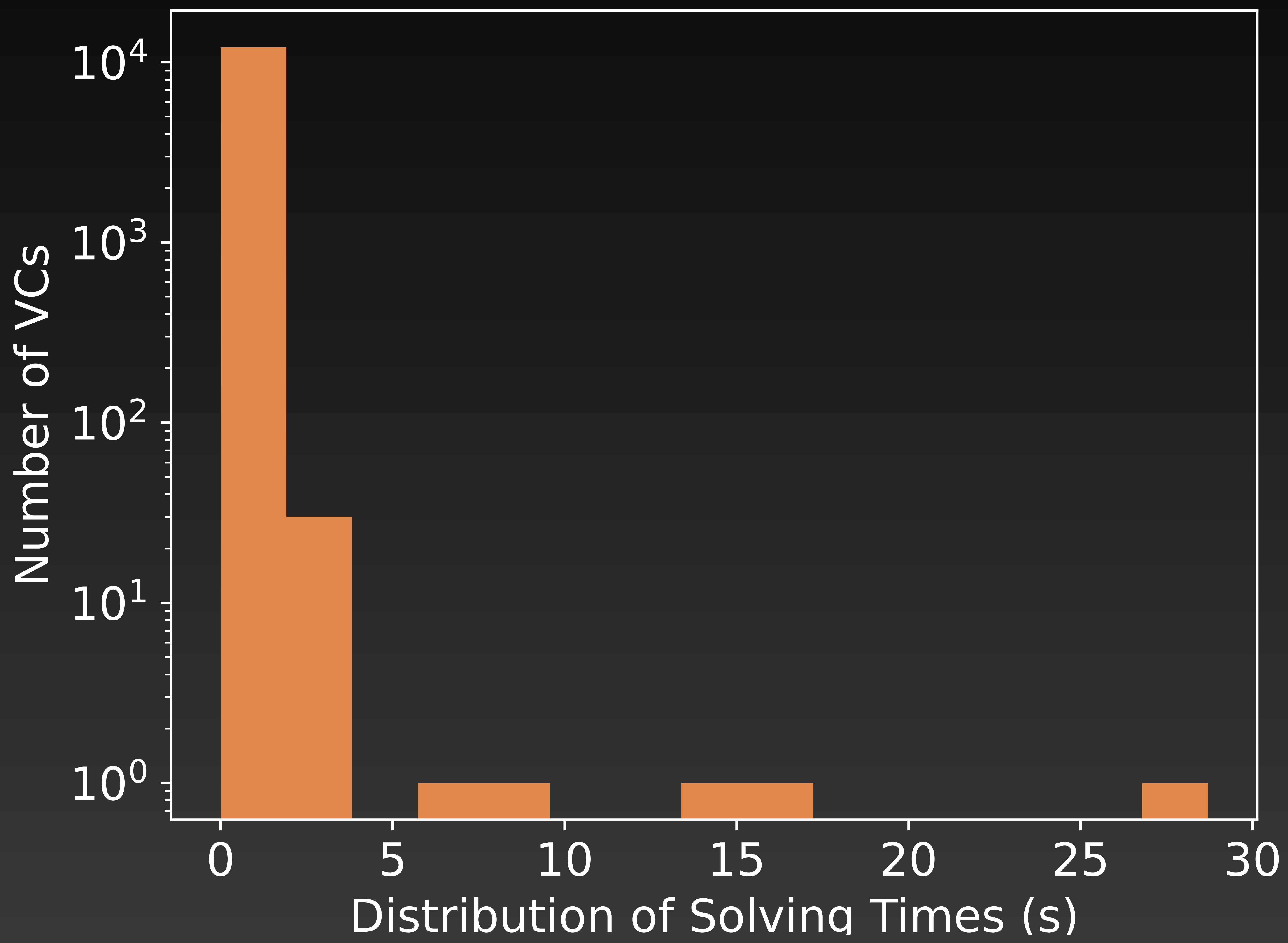
Verification Performance

Verification Conditions Solving Time

12’122 VCs

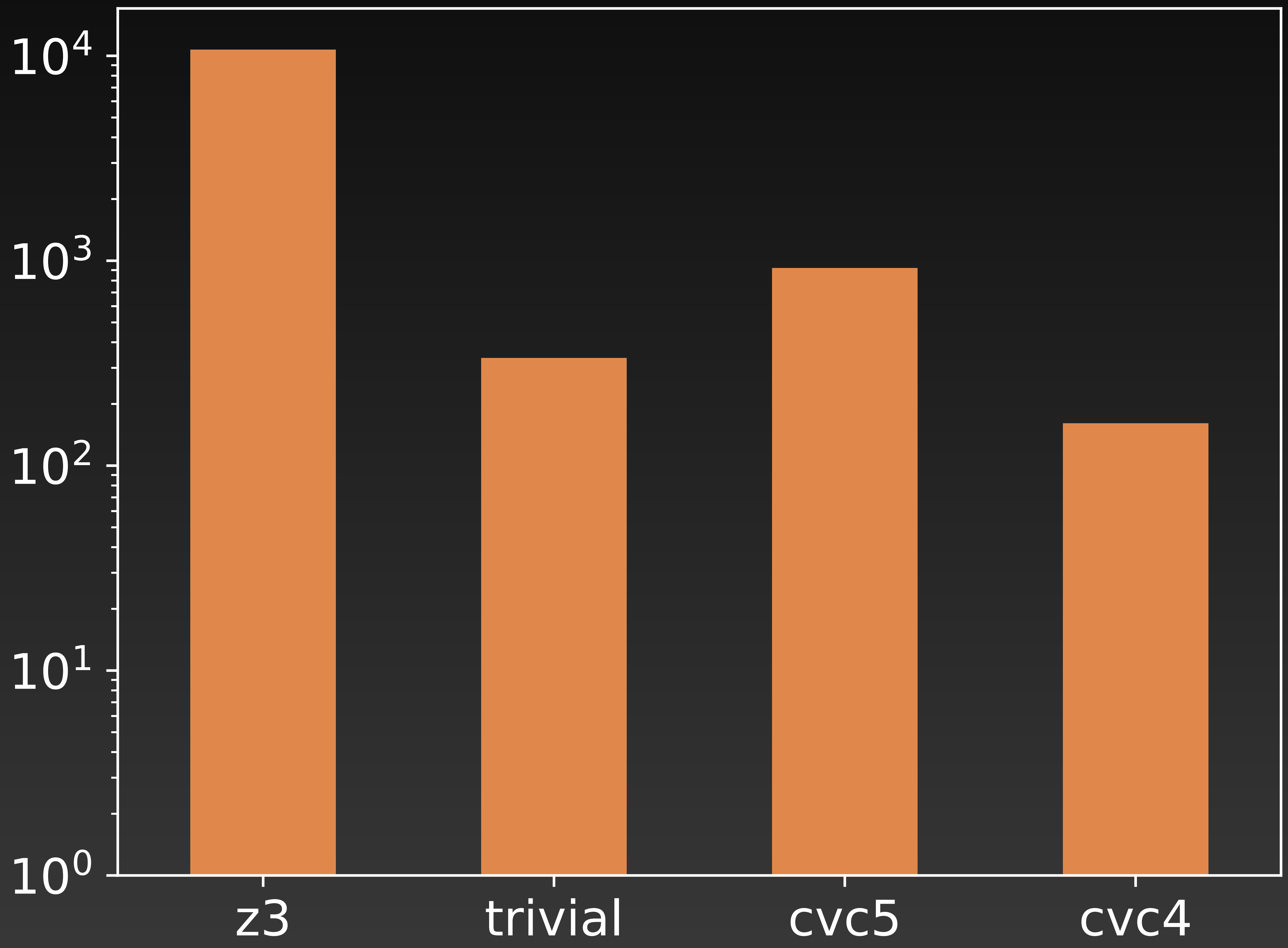
Mean: 0.16 second

Median: 0.1 second



Verification Performance

VCs per Solvers Distribution



Performance Evaluation

Protocol

Original LongMap, **Verified** LongMap, Scala **HashMap** (arbitrary keys), and **Opti** (without the indirection in `_values`)

Scenarios (see paper)

1. Lookups in pre-populated map
2. **Population of the map, followed by lookups**
3. Population, deletion of 1/2 keys, population, followed by lookups

For 2^{15} and 2^{22} randomly ordered pairs

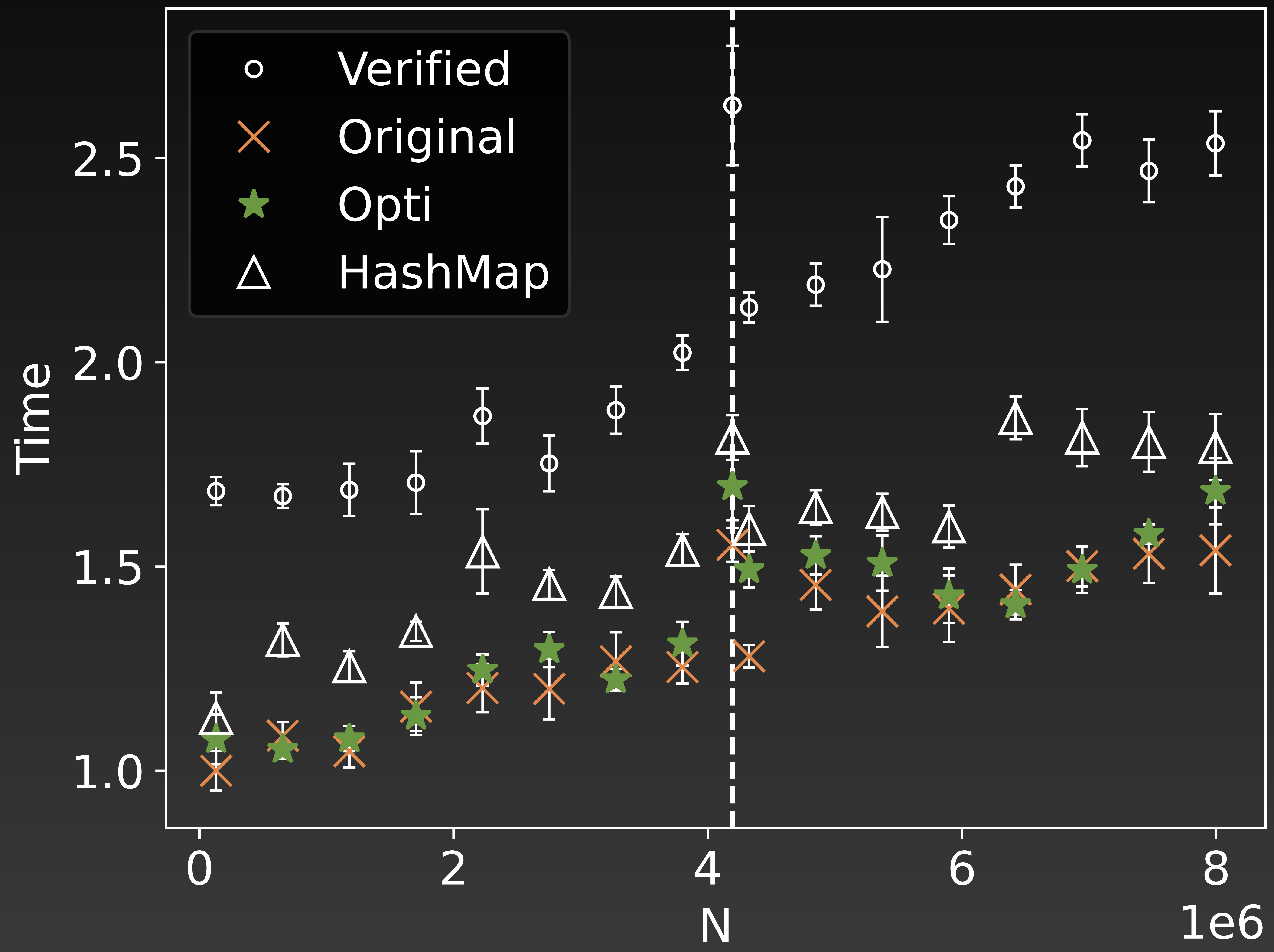
Performance Evaluation

Population + Lookups: 2^{22} Pairs, 2^4 Initial Capacity

Resizing

To populate Original: ~ 1380 ms

To populate Verified: ~ 2300 ms



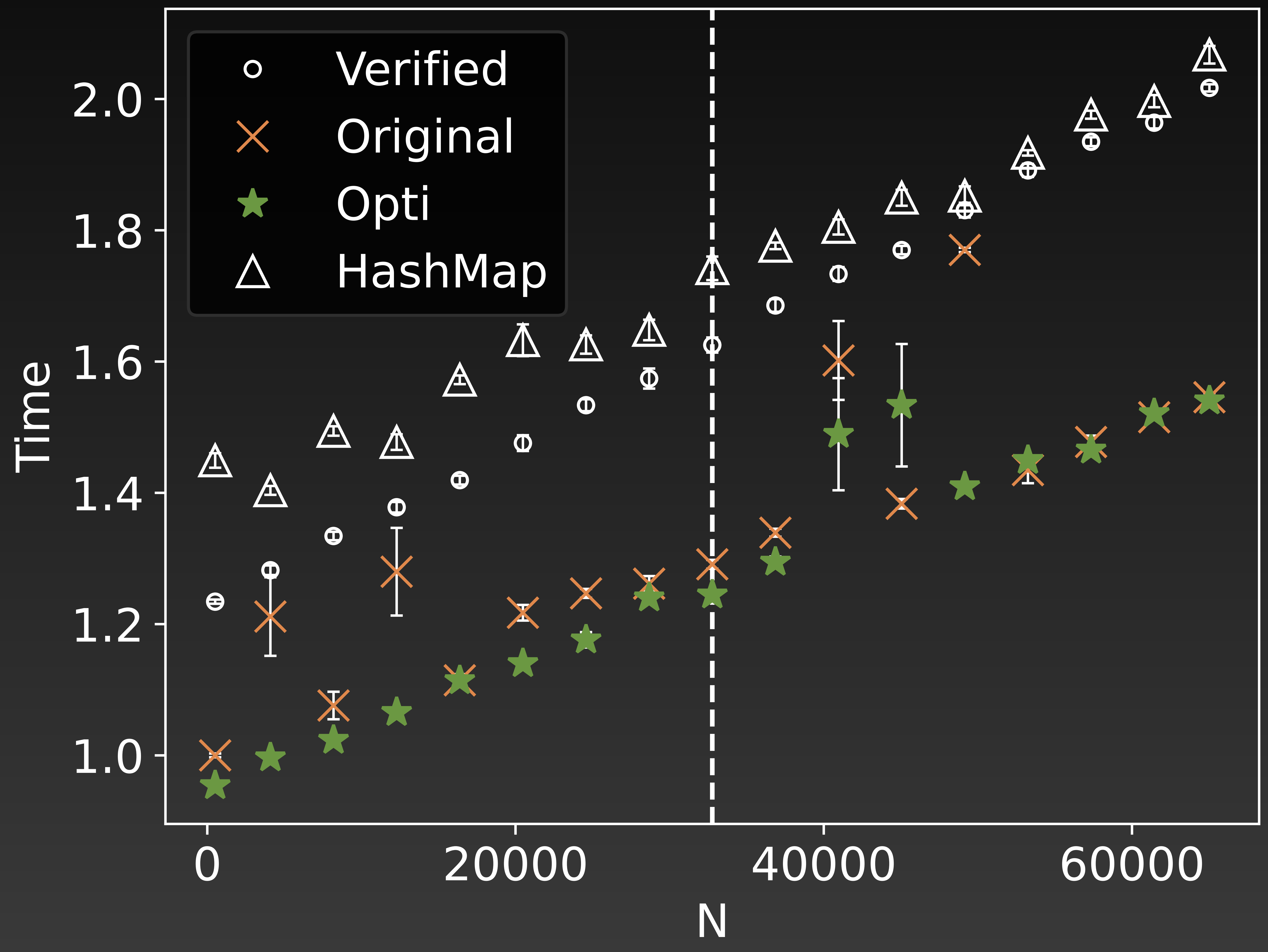
Performance Evaluation

Population + Lookups: 2^{15} pairs, 2^{17} Initial Capacity

NO resizing

To populate Original: ~1500 μs

To populate Verified: ~ 1900 μs



Performance Evaluation

Consequences of adapting

Indirection in `_values`

→ responsible for most overhead (cf **Opti**)

Initialisation: writes values arrays (no nulls)

→ slower than original but infrequent calls

Counter checks (termination check)

→ very little impact (cf **Opti**)

Conclusion

Contributions

Verified `LongMap` from Scala standard library

- realistic highly performant mutable Hash Table

Performance within 1.5x of original

- close to `HashMap` of the Scala library

Introduced a swap operation in Stainless

- better expressiveness without aliasing (e.g., decorator)

Found a bug in the original implementation

Code: github.com/epfl-lara/bolts/tree/main/data-structures/maps/mutablemaps

Contact: samuel.chassot@epfl.ch

Backup slides

Related works

Case studies

1. De Boer, De Gouw, Klamroth, Jung, Ulbrich, Weigl: *Formal Specification and Verification of JDK's Identity Hash Map Implementation*. Formal Aspects of Computing 2023

2. Hance, Lattuada, Hawblitzel, ,Howell, Johnson, Parno: *Storage Systems are Distributed Systems (So Verify Them That Way!)*. OSDI 2020

3. Polikarpova, Tschannen, Furia: *A fully verified container library*. Formal Aspects of Computing 2018

4. Jahob Hashtables Codebase, https://github.com/epfl-lara/jahob/tree/master/examples/containers/hashtable

Implementation

Probing function

```
def nextIndex(ee: Int, x: Int, mask: Int): Int =
  (ee + 2 * (x + 1) * x - 3) & mask
```

- `_keys` and `_values` $N = 2^n$ for $3 \leq n \leq 30$
- $mask = N - 1$

Adapting for verification

- While loops → tail recursive functions
  - For more flexible specification
- MSBs passed information to ADTs
  - used when returning an index in the array
  - For better SMT performance

Adapting for verification

- Counter to prove termination
  - Used in probing loops
  - Could not prove that the probing function would terminate

Adapting for verification

_values array

- Indirection in the `_values` array
  - Original: `Array[AnyRef]` with casts
  - Not possible with Stainless
  - Verified: `Array[ValueCell[V]]`
  - `ValueCellFull[V](v: V)` or `EmptyCell[V]()`

Adapting for verification

`_values` array

- In original implementation: casts + `null` initial values

```
_values: Array[AnyRef] = new Array[AnyRef](N)
def set(i: Int, v: V) = _values(i) = v.asInstanceOf[AnyRef]
def get(i: Int): V = _values(i).asInstanceOf[V]
```

- Our version

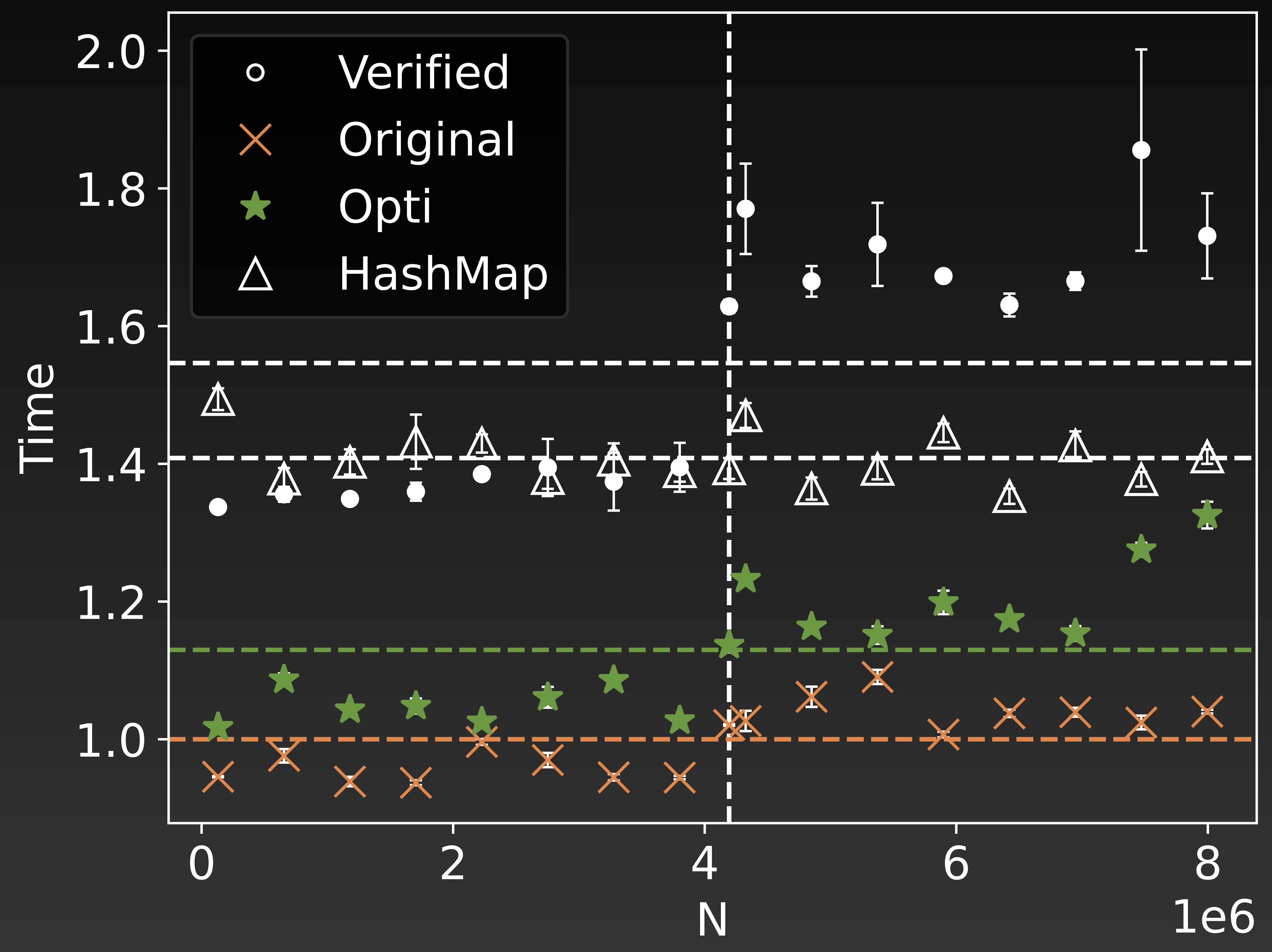
```
trait ValueCell[V]
case class ValueCellFull[V](v: V) extends ValueCell[V]
case class EmptyCell[V]() extends ValueCell[V]

_values: Array[ValueCell[V]]  = Array.fill(N)(EmptyCell[V]())
def set(i: Int, v: V) = _values(i) = ValueCellFull(v)
def get(i: Int): V = _values(i).getOrDefault
```

⇒ `Nulls` and casts replaced by a new level of indirection

Performance Evaluation

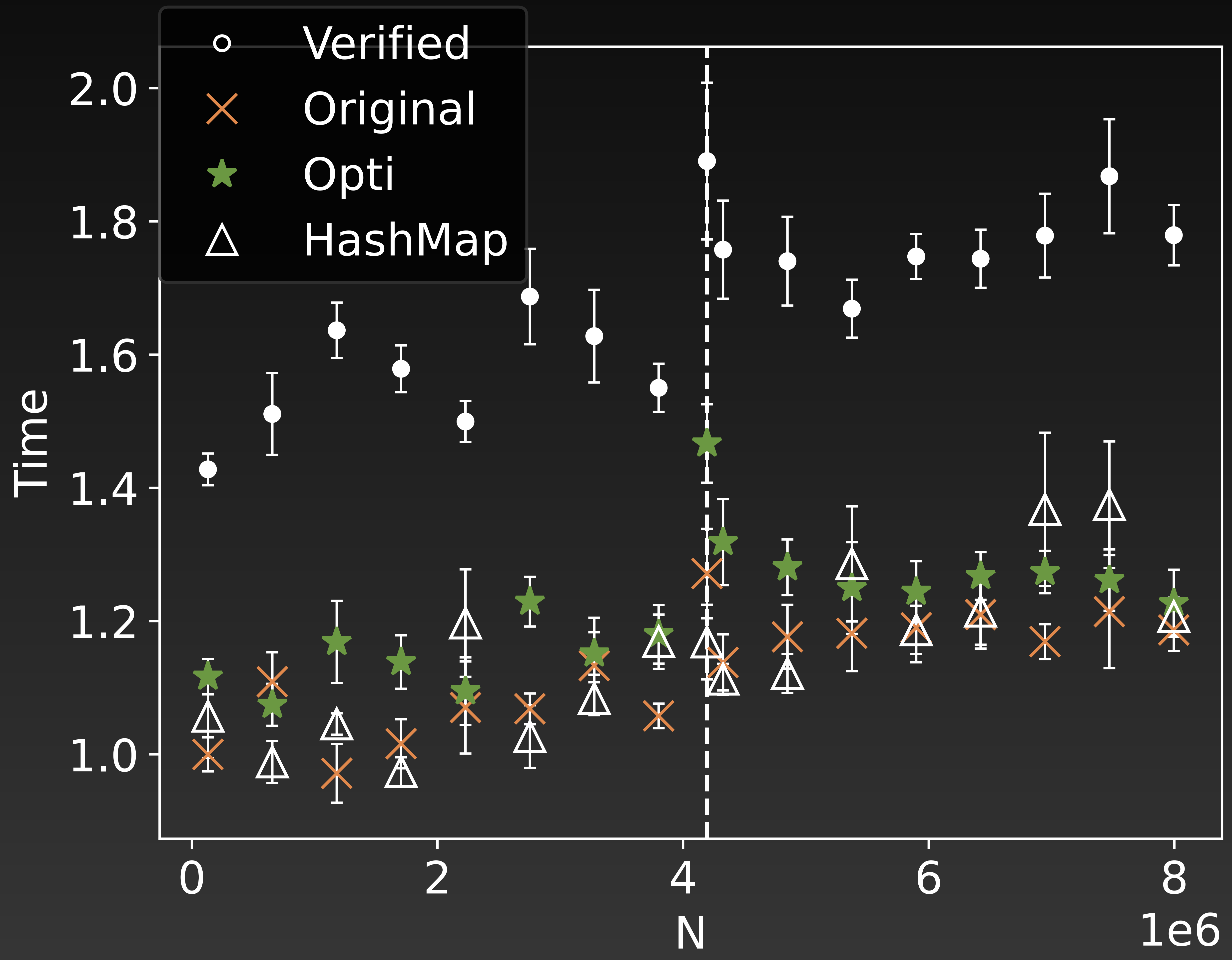
Scenario 1: lookup in pre-populated



2^{22} pairs, (normalised per operation)

Performance Evaluation

Scenario 3: population with remove + lookups



2^{22} pairs, remove 2^{21}, initial capacity 16