Stainless Verifier and Composition in Verification

Samuel Chassot EPFL - 18.12.2024

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**

Automated Proof: Zip

Why3

```
module ZipDemo
  (* Omitted imports *)

  let rec zip (xs: list int) (ys: list bool) : list (int, bool)
    requires { length xs <= length ys }
    ensures { map  (fun p -> let (x, _) = p in x) result = xs }
    variant { xs }
    = match xs, ys with
      | Cons x xs0, Cons y ys0 -> Cons (x, y) (zip xs0 ys0)
      | _, _ -> Nil
      end
end
```

```
❯ why3 prove -P cvc5 zip.mlw
File zip.mlw:
Goal zip'vc.
Prover result is: Unknown (unknown + incomplete) (0.03s, 12488 steps).
```

```
❯ why3 prove -P cvc5 zip.mlw
File zip.mlw:
Goal zip'vc.
Prover result is: Valid (0.02s, 11175 steps).
```

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)
```

```
stainless summary

zip.scala:5:5:     zip  non-negative measure                                       valid  U:smt-cvc5  0.0
zip.scala:9:11:    zip  postcondition                                              valid  U:smt-cvc5  0.1
zip.scala:9:24:    zip  measure decreases                                          valid  U:smt-cvc5  0.0
zip.scala:9:24:    zip  precond. (call zip((scrut._1.t): @DropVCs , (scrut._...))  valid  U:smt-cvc5  0.1
zip.scala:11:15:   zip  postcondition                                              valid  U:smt-cvc5  0.0
----------------------------------------------------------------------------------------------------------
total: 5    valid: 5    (0 from cache, 0 trivial) invalid: 0    unknown: 0    time:    0.29
```

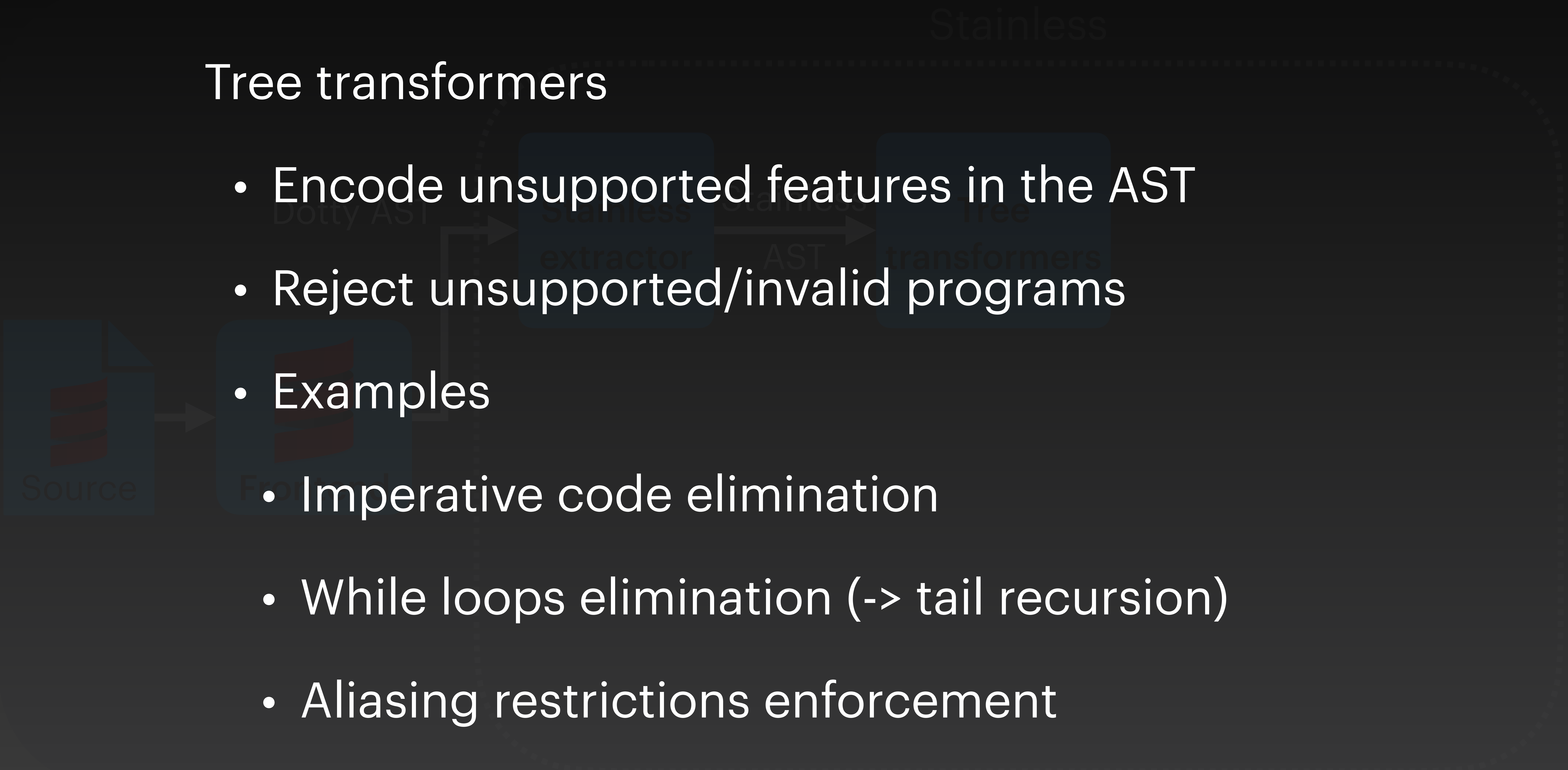
```
warning: Found counter-example:
warning:   xs: List[Int]          -> Cons[Int](0, Nil[Int]())
           ys: List[Boolean] -> Nil[Boolean]()
```

Stainless: Proof by Induction

```
/**
  * 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 Nil => ()
          case Cons(hd, tl) if (hd._1 == oKey) => ()
          case Cons(hd, tl) if (hd._1 != oKey) => lemma(tl, key, v, oKey)

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

Stainless

Verification framework for Scala



Tree transformers

- Encode unsupported features in the AST
- Reject unsupported/invalid programs
- Examples
  - Imperative code elimination
  - While loops elimination (-> tail recursion)
  - Aliasing restrictions enforcement

Stainless

Verification framework for Scala

Source

Frontend

Dotty AST

Stainless

Stainless extractor

Stainless AST

Tree transformers

Stainless AST

Verification conditions generator based on SystemFR type checking

Inox AST

Inox

Inox unrolling engine

SMT Solvers

Z3

cvc5

Princess

[Jad Hamza, Nicolas Voirol, and Viktor Kunčak. System FR: Formalized foundations for the Stainless verifier. Proc. ACM Program. Lang, (OOPSLA), November 2019.]

[Philippe Suter, et al. Satisfiability modulo recursive programs. In Static Analysis Symposium (SAS), 2011.]
[Nicolas Voirol, et al. Counter-example complete verification for higher-order functions. In Scala Symposium, 2015.]

LongMap case study

Chassot, S., Kunčak, V. (2024). Verifying a Realistic Mutable Hash Table. In: Benzmüller, C., Heule, M.J., Schmidt, R.A. (eds) Automated Reasoning. IJCAR 2024. Lecture Notes in Computer Science(), vol 14739. Springer

LongMap Interface

Hash Table, 64-bit Integer Keys

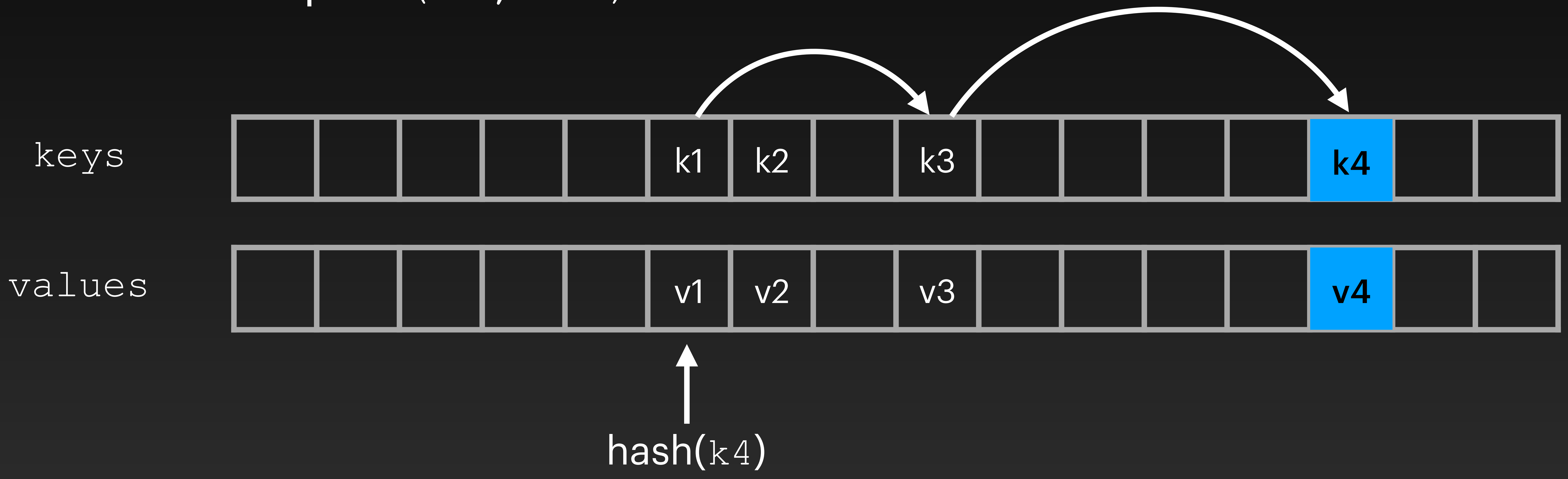
```
trait LongMap[V]:
  def contains(key: Long): Boolean
  def apply(key: Long): V             // Lookup
  def update(key: Long, v: V): Boolean
  def remove(key: Long): Boolean
  def repack(): Boolean
```

LongMap open addressing

LongMap Hash Table

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

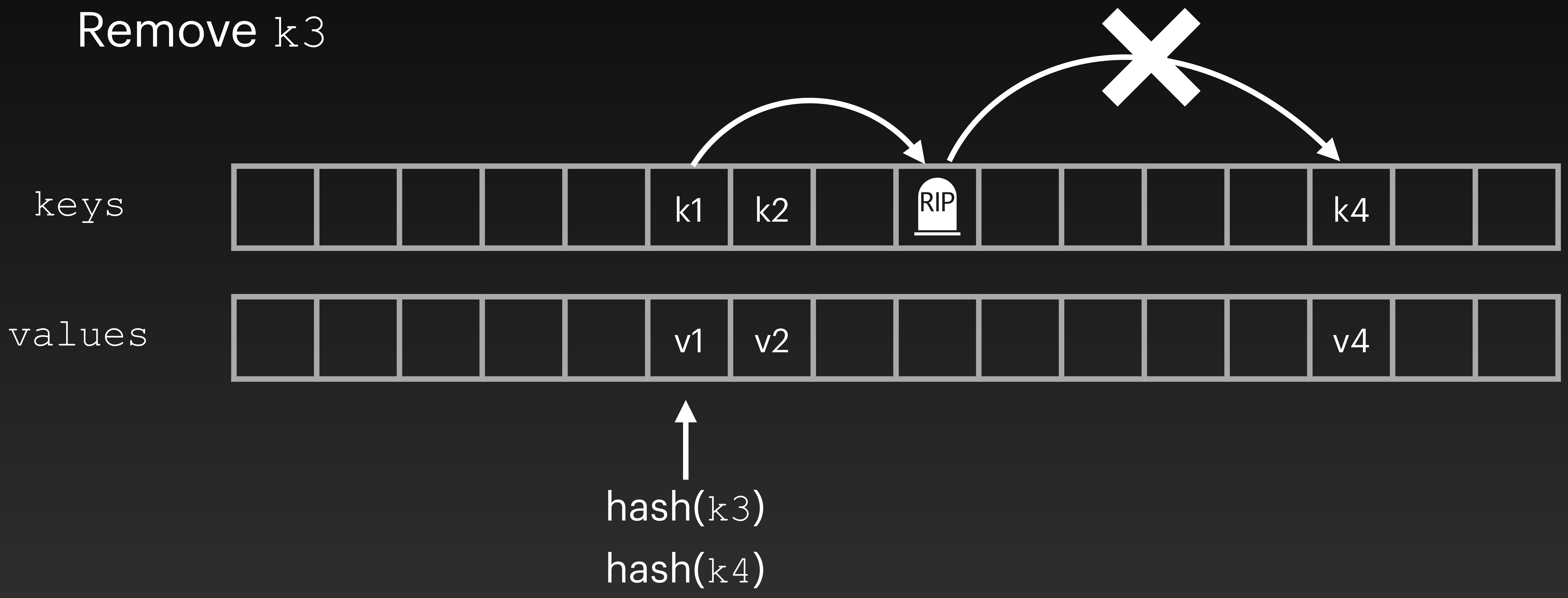
Insert the pair `(k4, v4)`



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

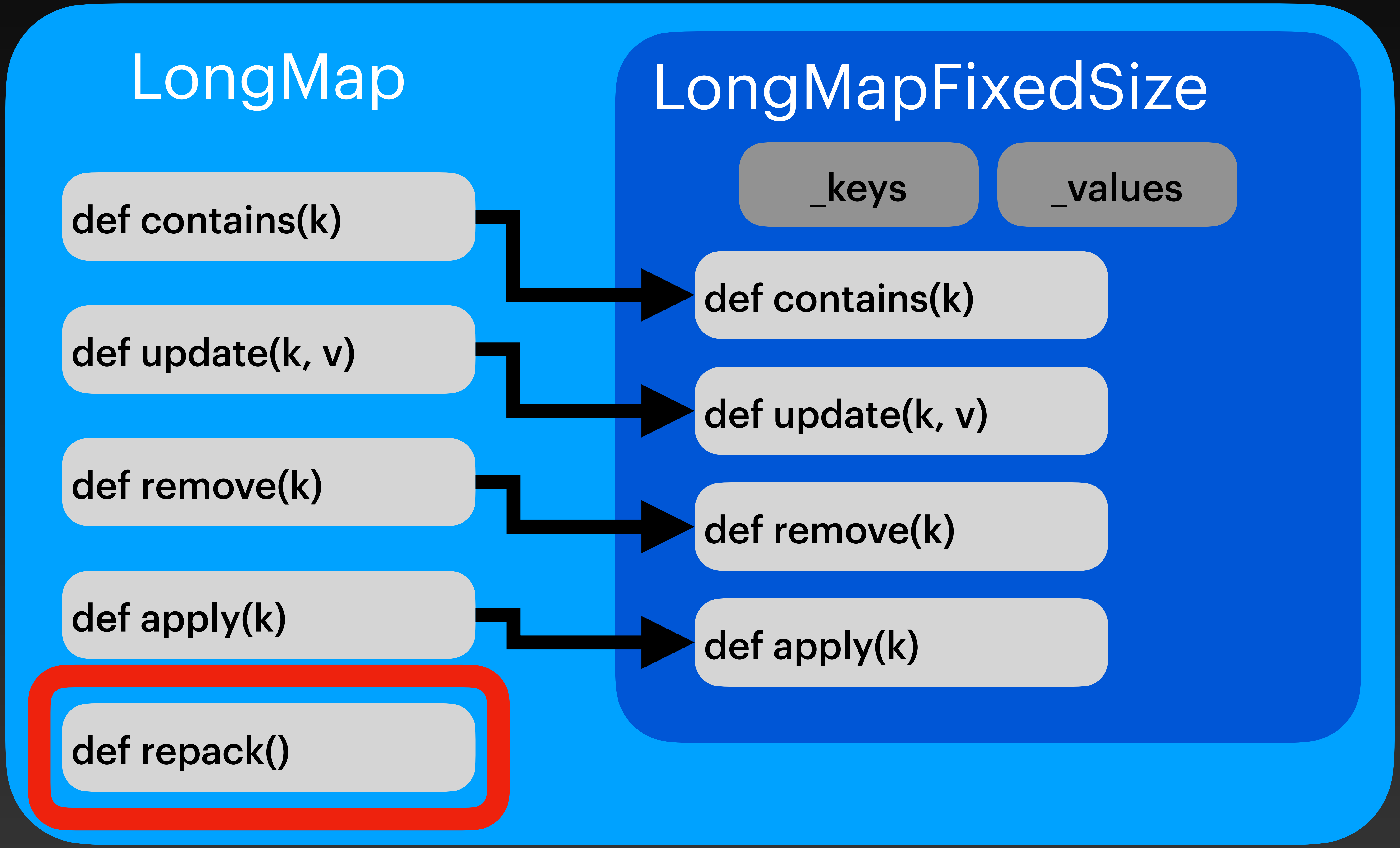
Add loop counter checks to prove termination

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[@mutable 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

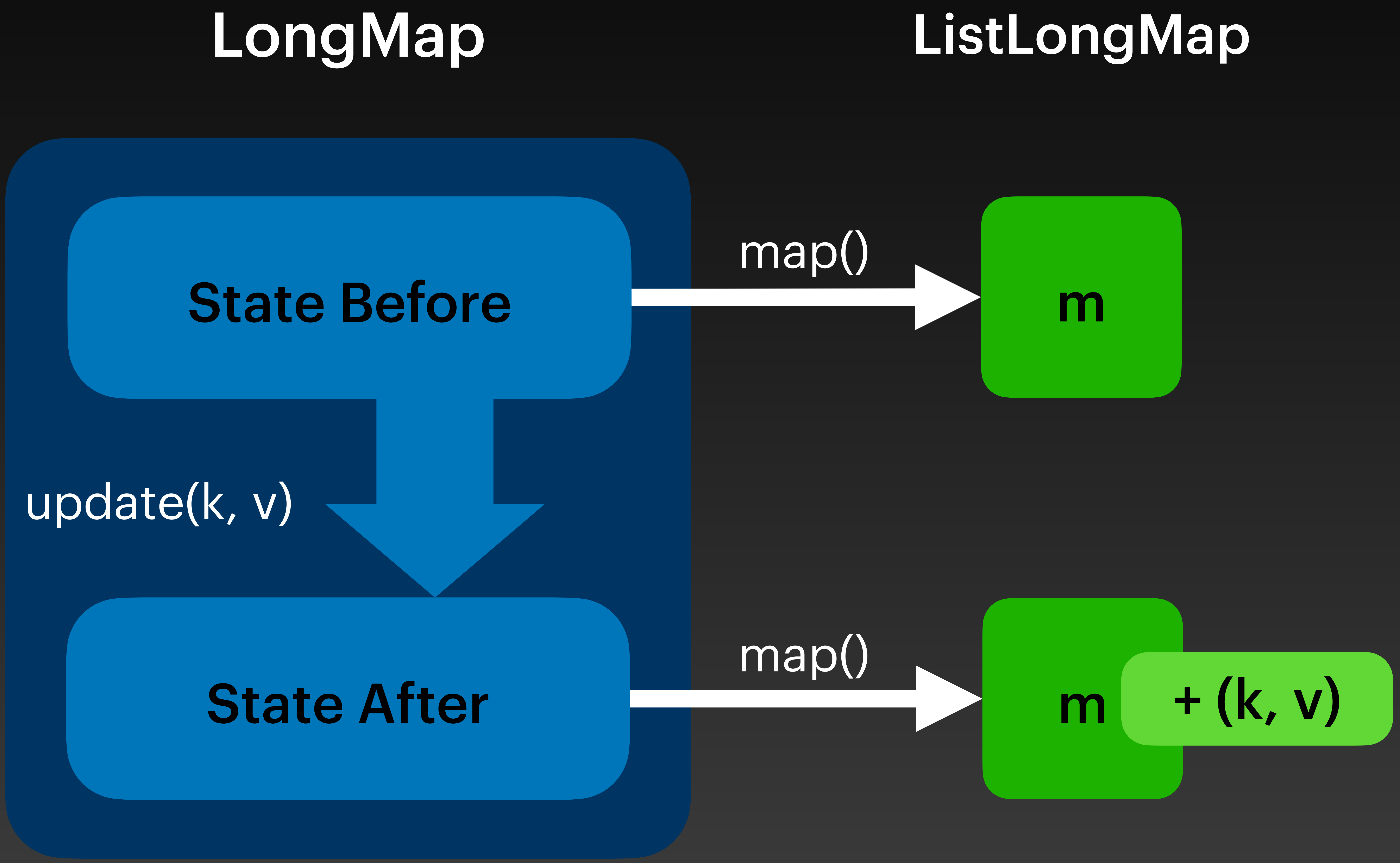
ListLongMap

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

```scala
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

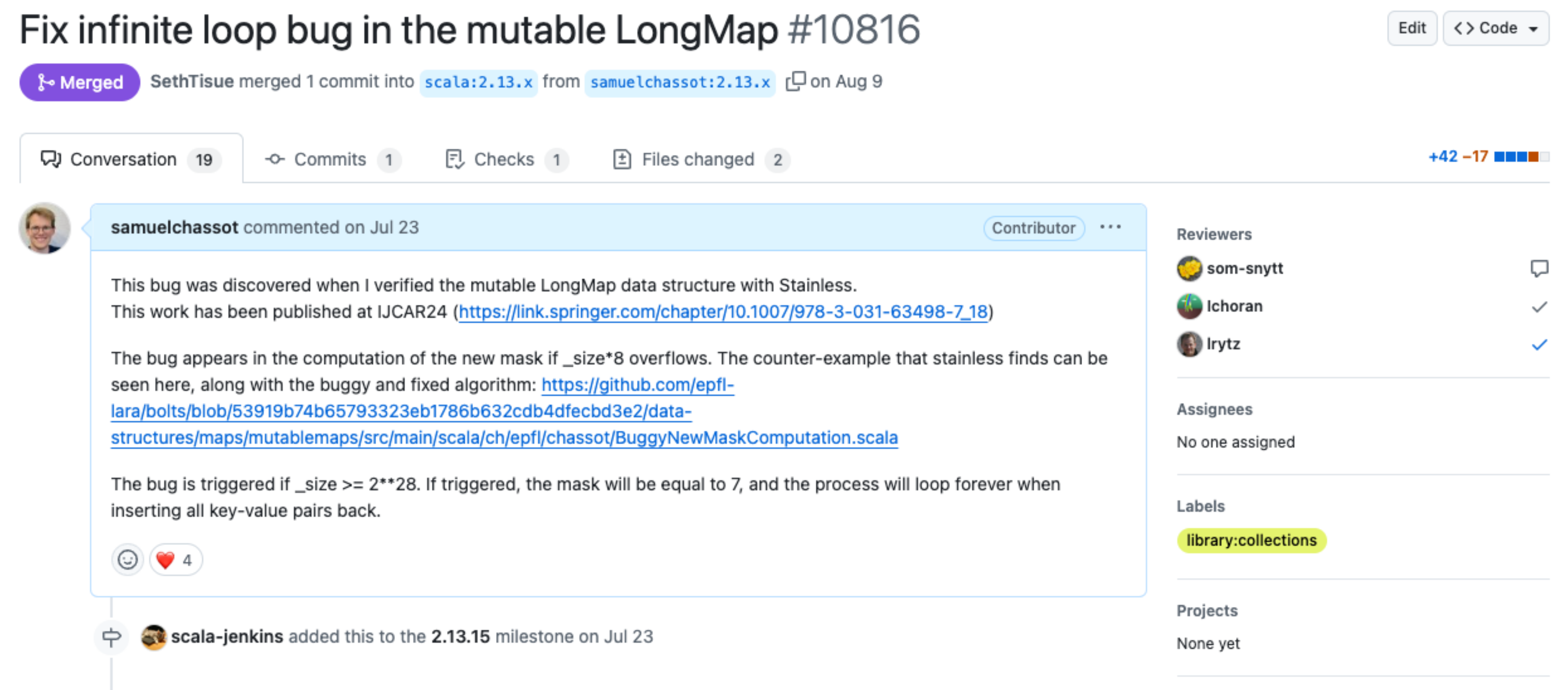
```
// Compute the new size for the array based on map’s state
def computeNewMask(mask: Int, _size: Int, _vacant: Int) = {
  var m = mask
  if (2 * (_size + _vacant) >= mask && !(5 * _vacant > mask)) {
    m = ((m << 1) + 1) & IndexMask
  }
  while (m > 8 && 8 * _size < m) {
    m = m >>> 1
  }
  m
}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

`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

Bug in the Original Implementation

New size computation



Fix infinite loop bug in the mutable LongMap #10816

SethTisue merged 1 commit into `scala:2.13.x` from `samuelchassot:2.13.x` on Aug 9

samuelchassot commented on Jul 23 Contributor

This bug was discovered when I verified the mutable LongMap data structure with Stainless.
This work has been published at IJCAR24 (https://link.springer.com/chapter/10.1007/978-3-031-63498-7_18)

The bug appears in the computation of the new mask if _size*8 overflows. The counter-example that stainless finds can be seen here, along with the buggy and fixed algorithm: https://github.com/epfl-lara/bolts/blob/53919b74b65793323eb1786b632cdb4dfecbd3e2/data-structures/maps/mutablemaps/src/main/scala/ch/epfl/chassot/BuggyNewMaskComputation.scala

The bug is triggered if _size >= 2**28. If triggered, the mask will be equal to 7, and the process will loop forever when inserting all key-value pairs back.

scala-jenkins added this to the **2.13.15** milestone on Jul 23

Reviewers
- som-snytt
- Ichoran ✓
- lrytz ✓

Assignees
No one assigned

Labels
library:collections

Projects
None yet

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 && _size < (m >>> 3)) {
  decreases(m)
    m = m >>> 1
  }
  m
}.ensuring (res => validMask(res) && _size <= res + 1)
```

Formally verified fix

Performance analysis

Statistics

Lines of Code

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

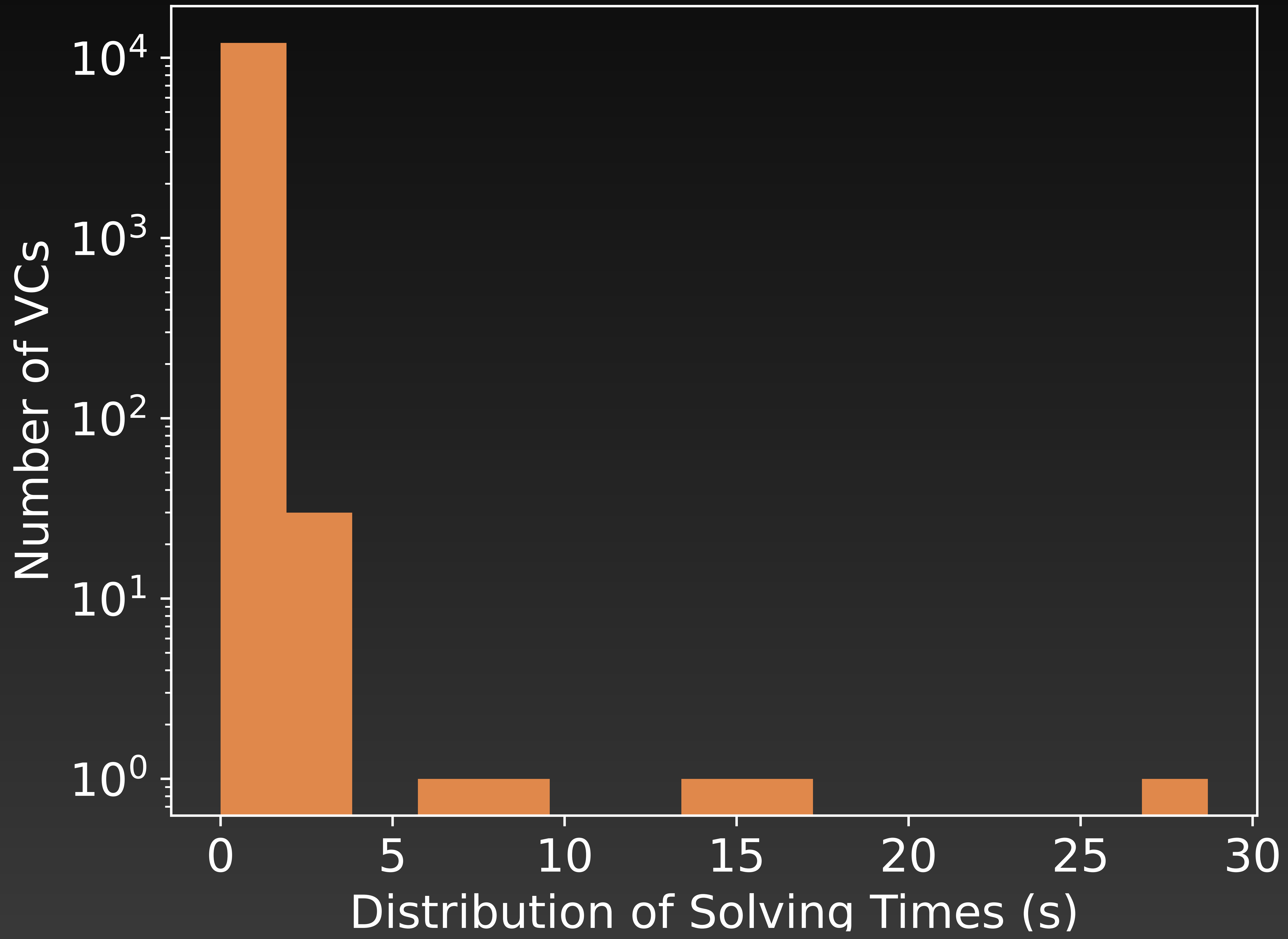
Verification Performance

Verification Conditions Solving Time

12’122 VCs

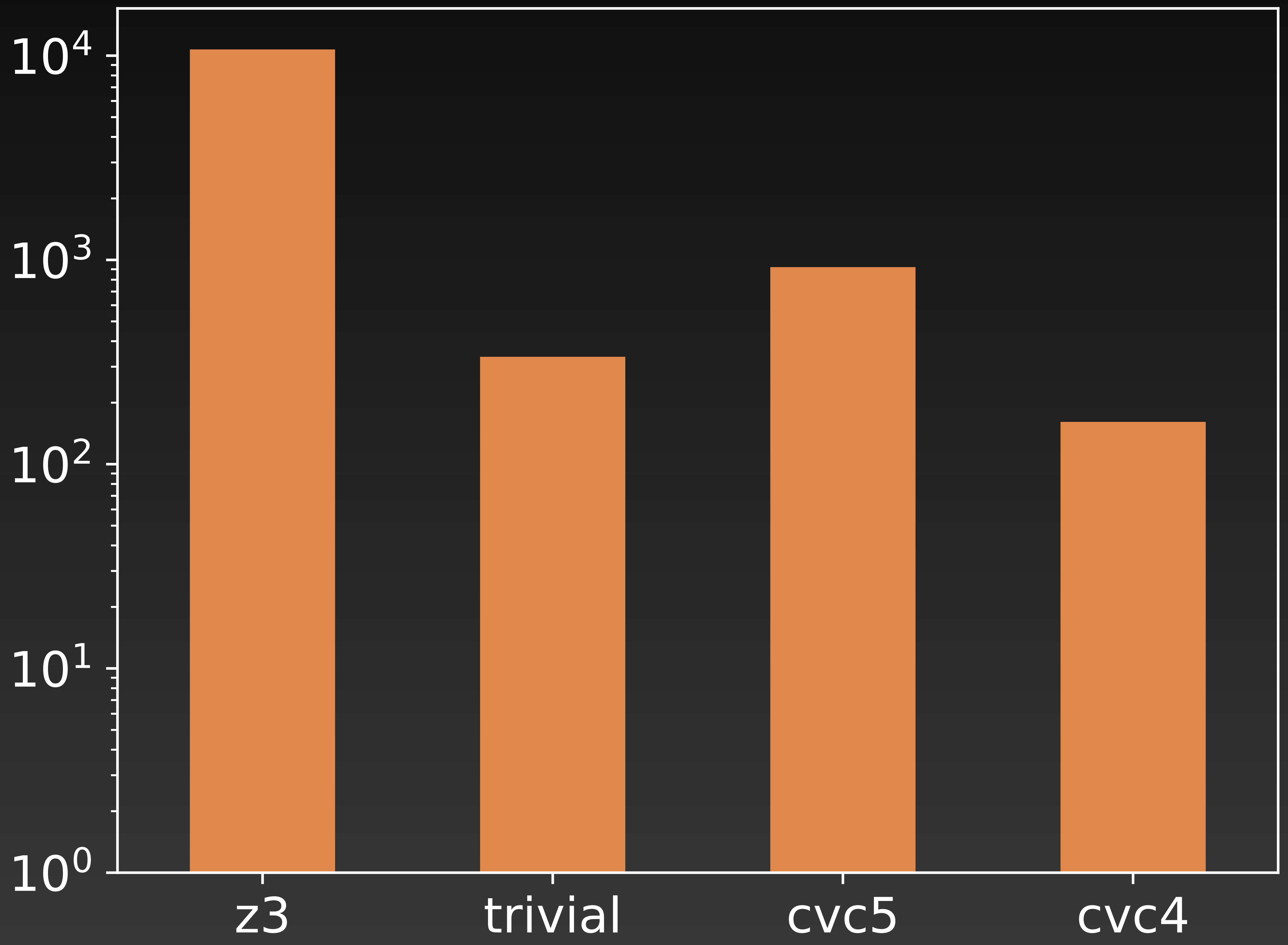
Mean: 0.16 second

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

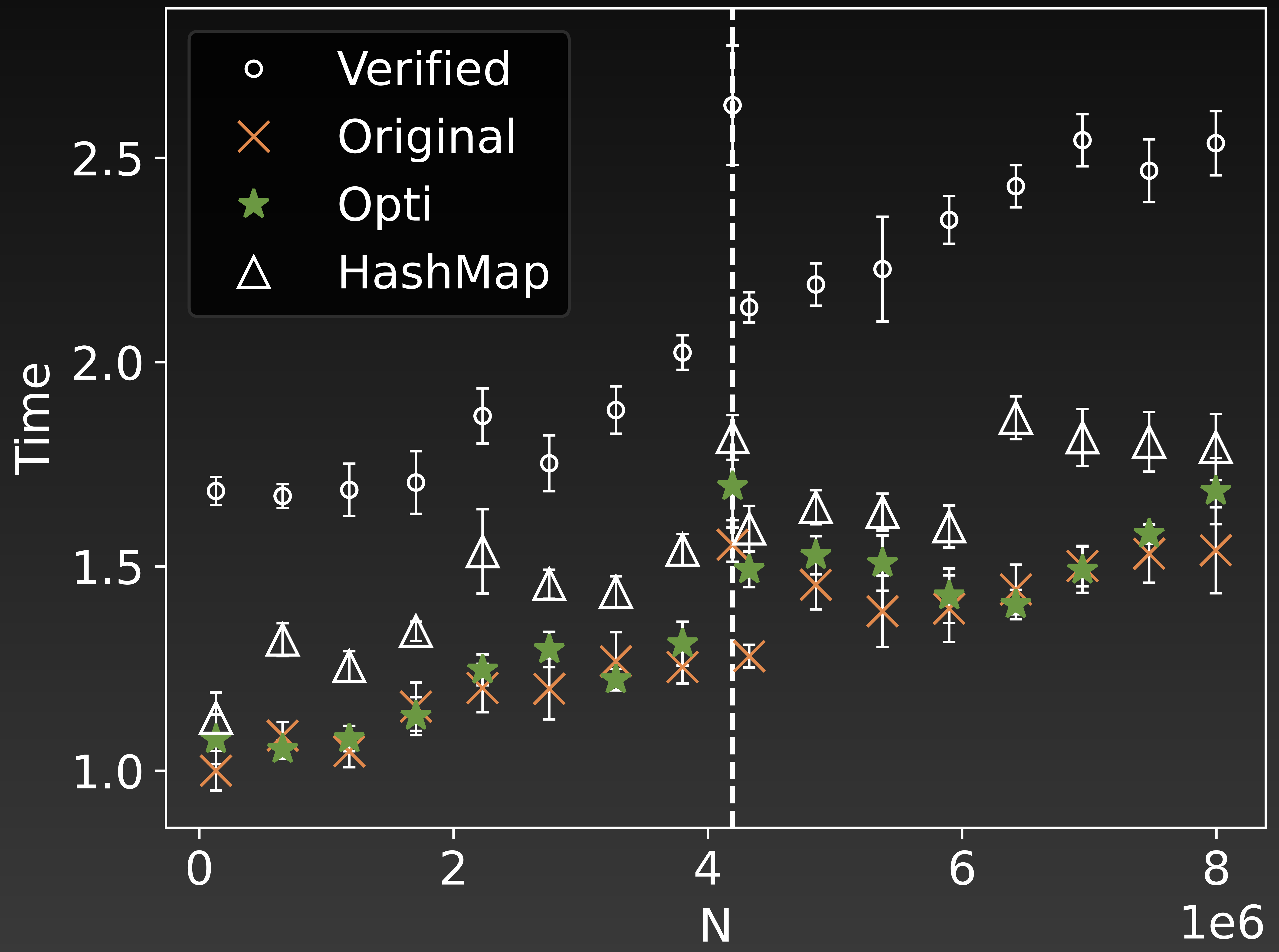
Performance Evaluation

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

Resizing

To populate Original: ~ 1380 ms

To populate Verified: ~ 2300 ms



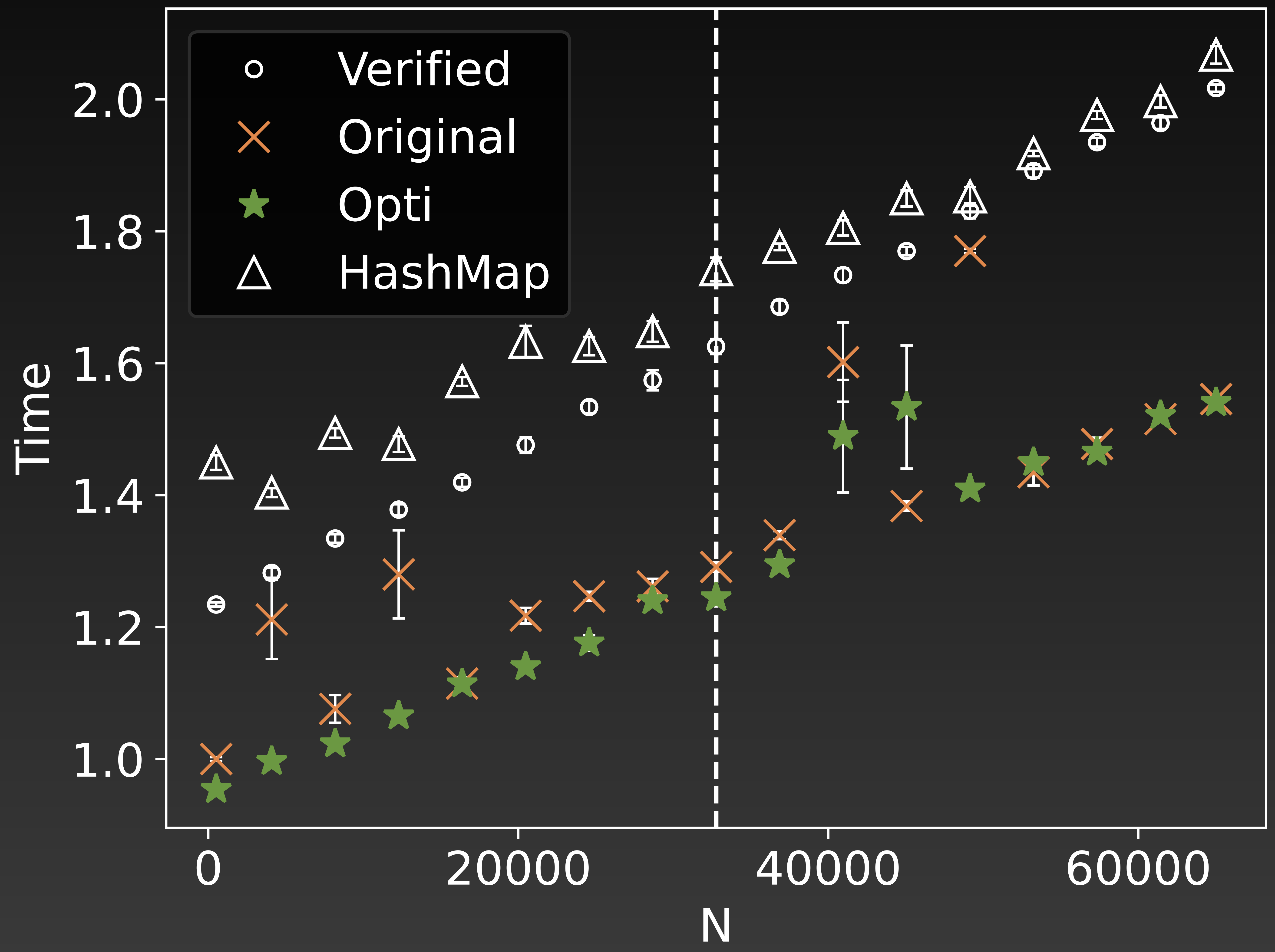
Performance Evaluation

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**)

Hash Table with generically typed keys

HashMap Interface

Hash Table, Generically typed keys

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

```
trait Hashable[K] {
  @pure
  def hash(k: K): Long
}
```

Specification

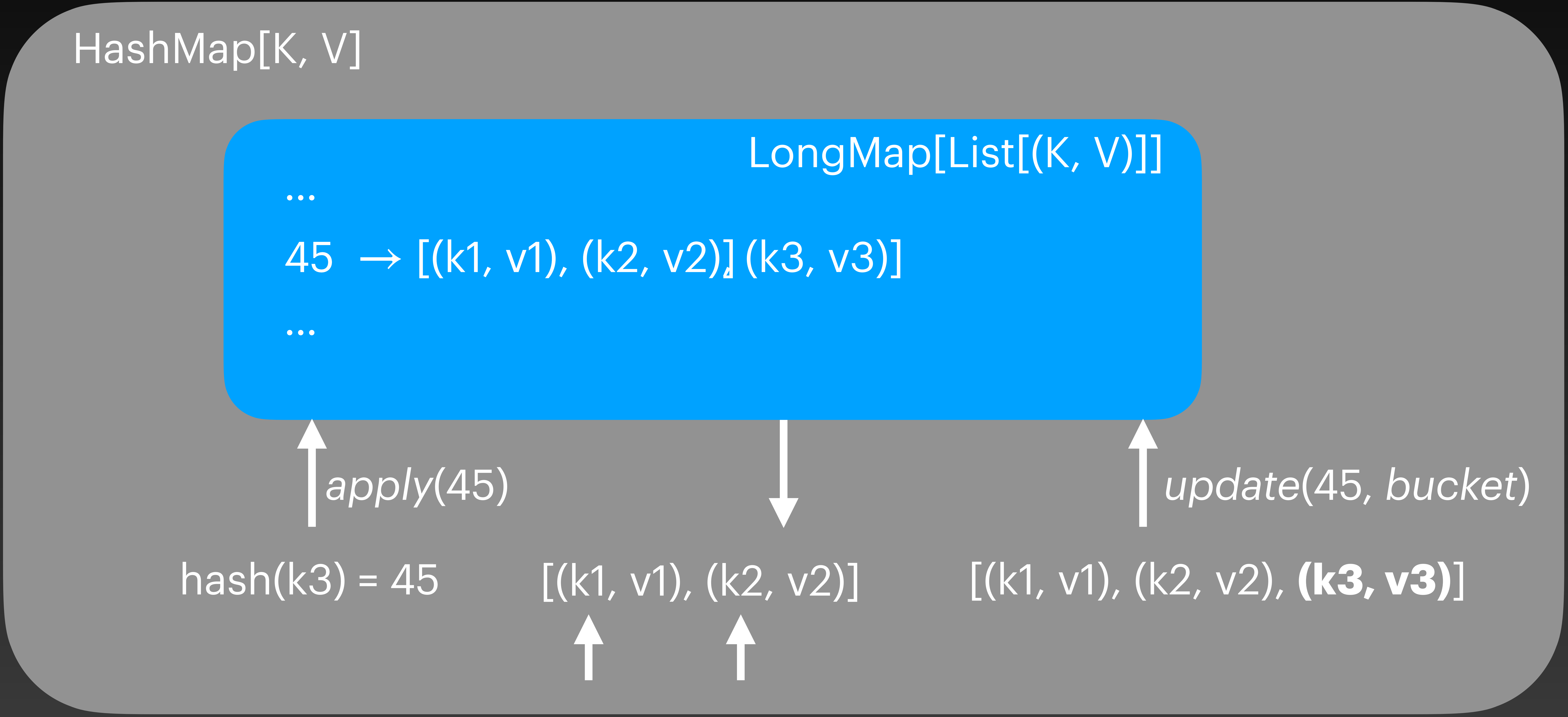
ListMap Interface

```scala
trait ListMap[K, B](toList: List[(K, B)]) {

  def contains(key: K): Boolean

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

  def apply(key: K): B

  def +(keyValue: (K, B)): ListMap[B]

  def -(key: K): ListMap[B]
}
```

⇒ Difference with ListLongMap → NOT ordered anymore

Implementation

Insert (k3: K, v3: V)



HashMap[K, V]

LongMap[List[(K, V)]]

…

45 → [(k1, v1), (k2, v2)], (k3, v3)]

…

apply(45)

update(45, *bucket*)

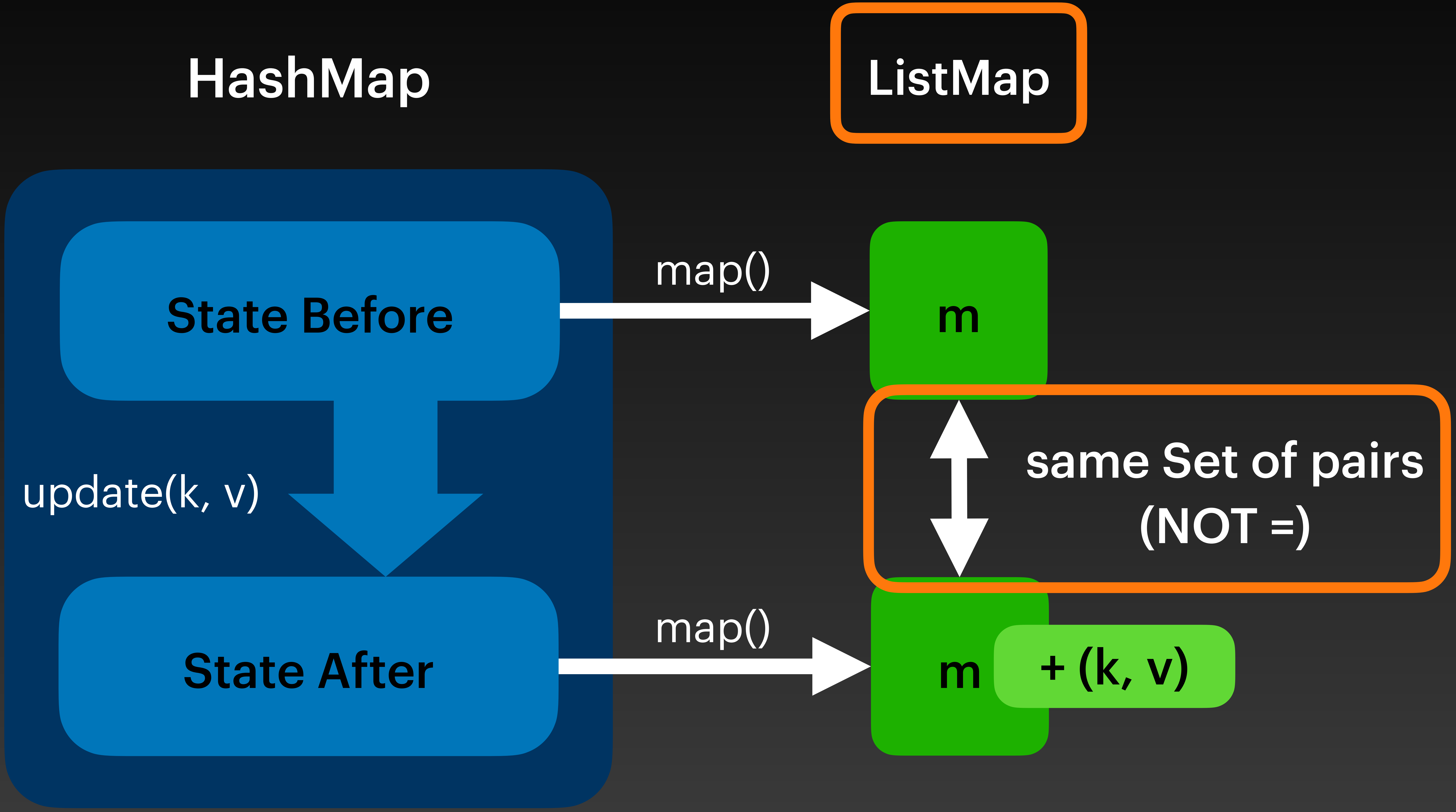
hash(k3) = 45

[(k1, v1), (k2, v2)]

[(k1, v1), (k2, v2), **(k3, v3)**]

Verification using abstraction function

Proof Structure



HashMap

ListMap

State Before

update(k, v)

State After

map()

m

same Set of pairs
(NOT =)

map()

m + (k, v)

Code size

Class	Program LOC	Proof + Specification LOC	Total LOC
MutableLongMap	409	7'358	7'767
MutableHashMap	95	1'230	1'325

Hash Set

HashSet Interface

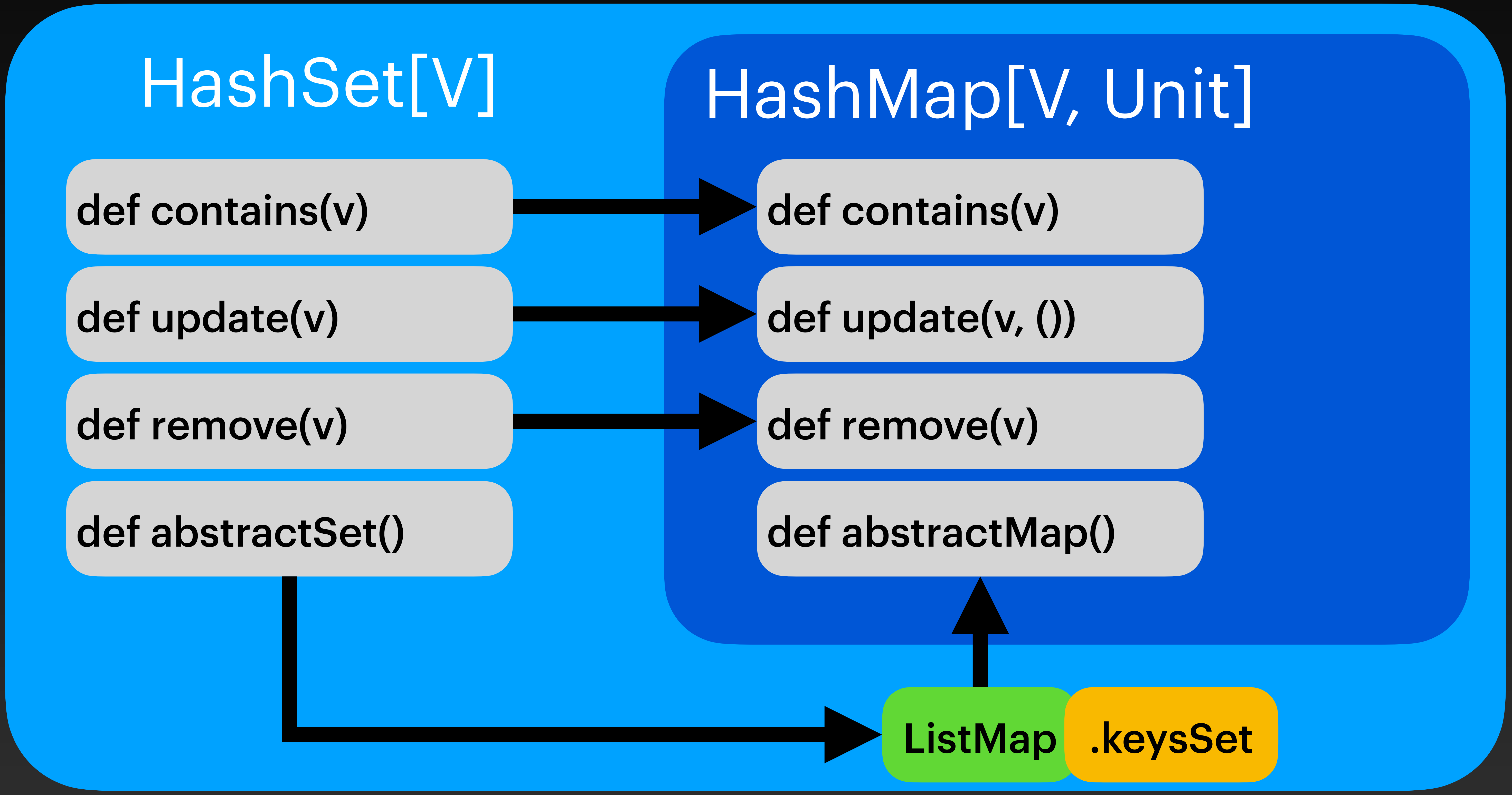
Generically typed Set

```
trait HashSet[V]:
  def contains(v: V): Boolean
  def update(v: V): Boolean
  def remove(v: V): Boolean
```

```
trait Hashable[V] {
  @pure
  def hash(v: V): Long
}
```

Implementation & Verification

HashSet



HashSet[V]

def contains(v)

def update(v)

def remove(v)

def abstractSet()

HashMap[V, Unit]

def contains(v)

def update(v, ())

def remove(v)

def abstractMap()

ListMap .keysSet

Verification

- Add *keysSet* operation on *ListMap* and lemmas

Code size

HashSet

Class	Program LOC	Proof + Specification LOC	Total LOC
MutableLongMap	409	7'358	7'767
MutableHashMap	95	1'230	1'325
MutableHashSet	24	114	138

Application: caching

Caching

General pattern

$$f : A \to B \qquad\qquad cache : A \to option[B]$$

$$valid(cache) = \forall a : A \,.\; cache(a) = some(b) \implies b = f(a)$$

$\Rightarrow$ **General reusable pattern**

Caching

Tailored lemmas

```
def lemmaUpdatePreservesForallPairs[K, V](
    hm: HashMap[K, V],
    k: K,
    v: V,
    p: ((K, V)) => Boolean
  ): Unit = {
    require(hm.valid)
    require(hm.map.forall(p))
    require(p((k, v)))
    // ...
  }.ensuring(_ => {
    hm.update(k, v)
    hm.map.forall(p)
  })
```

```
def lemmaRemovePreservesForallPairs[K, V](
    hm: HashMap[K, V],
    k: K,
    p: ((K, V)) => Boolean
  ): Unit = {
    require(hm.valid)
    require(hm.map.forall(p))
    // ...
  }.ensuring(_ => {
    hm.remove(k)
    hm.map.forall(p)
  })
```

```
def lemmaForallPairsThenForLookup[K, V](
  hm: HashMap[K, V],
  k: K,
  p: ((K, V)) => Boolean
): Unit = {
  require(hm.valid)
  require(hm.map.forall(p))
  require(hm.contains(k))
  // ...
}.ensuring(_ => p((k, hm.apply(k))))
```

Ongoing work

Regex & lexical analysis

Regular expressions matching engine formally verified

- based on Brzozowski derivatives
- Applied caching pattern
- Optimised using Zipper[*]

Lexer

- Verified with respect to maximum munch principle
- Verified invertibility under some conditions
- (Future) Caching for better performance

[*] Romain Edelmann - Efficient Parsing with Derivatives and Zippers (thesis)

Other work

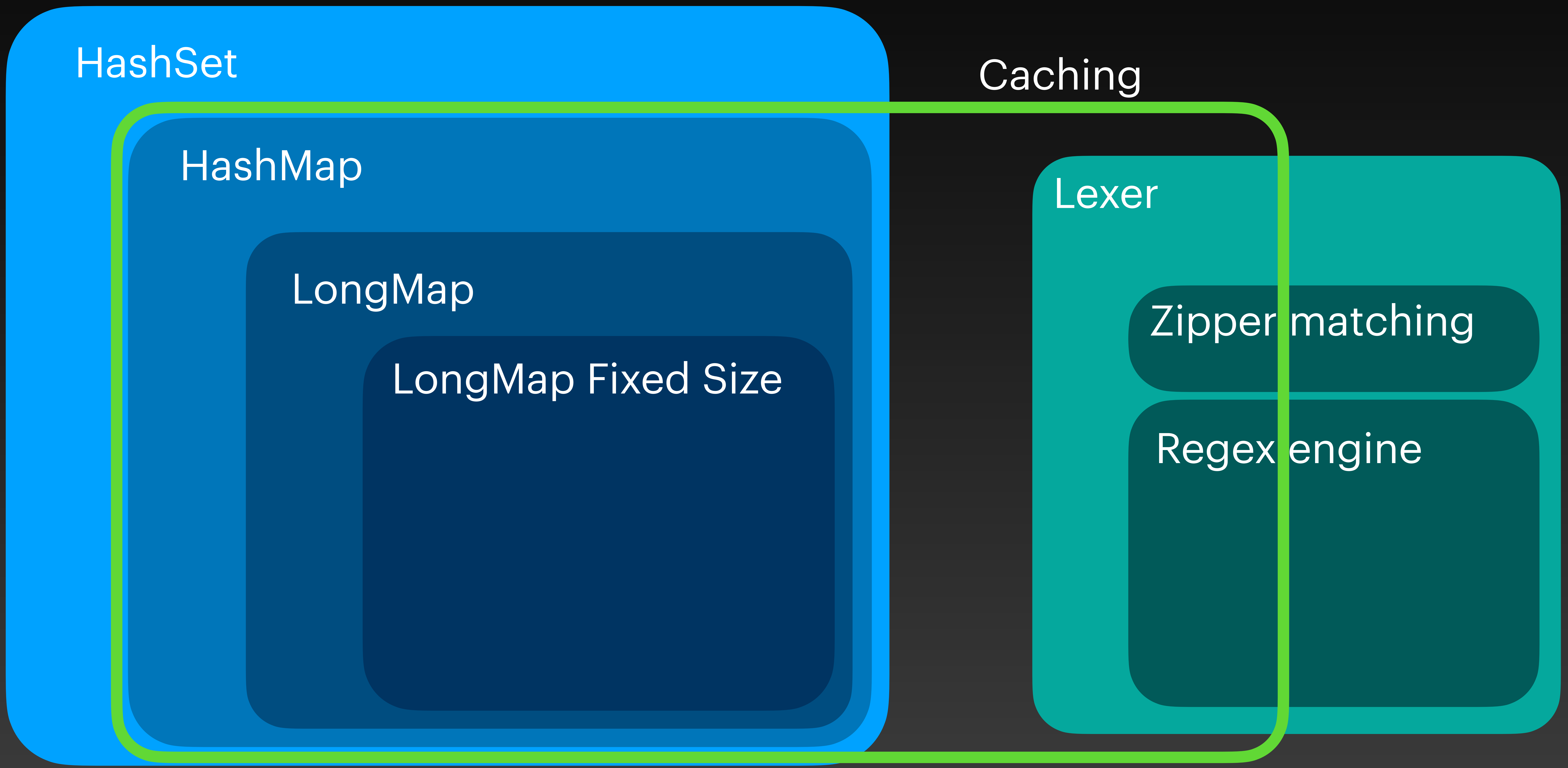
ASN.1 compiler verification

Verification of a compiler for ASN.1 serialisation format

- Project in collaboration with the ESA
- Verification of a bit stream data structure
  - No runtime errors
  - Invertibility
- Generating serialiser and deserialiser code
  - Absence of runtime errors -> no annotations required
  - Invertibility -> generating annotations
- Published at VMCAI 2025

Conclusion

Composition and decorator pattern



HashSet

HashMap

LongMap

LongMap Fixed Size

Caching

Lexer

Zipper matching

Regex engine

References

Chassot, S., Kunčak, V. (2024). Verifying a Realistic Mutable Hash Table. In: Benzmüller, C., Heule, M.J., Schmidt, R.A. (eds) Automated Reasoning. IJCAR 2024. Lecture Notes in Computer Science(), vol 14739. Springer

Mario Bucev, Samuel Chassot, Simon Felix, Filip Schramka, & Viktor Kunčak. (2024). Formally Verifiable Generated ASN.1/ACN Encoders and Decoders: A Case Study.

Backup slides

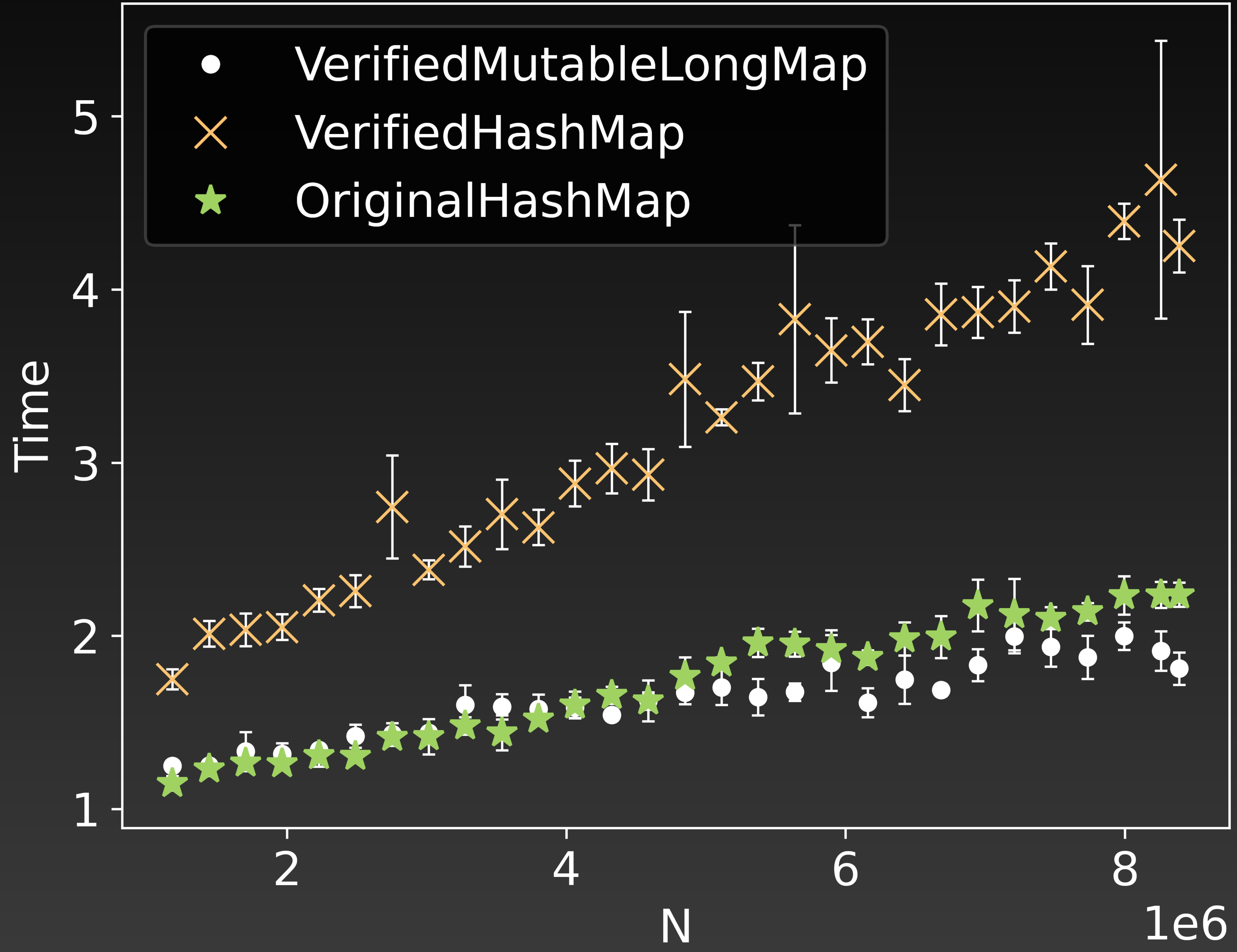
Performance Evaluation

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

Resizing

To populate Original: ~ 1700 ms

To populate Verified: ~ 2500 ms



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]()`

Conclusion

Verified `LongMap` from Scala standard library

Built on top of it

- Hash Table with generically typed keys
- Hash Set with generically typed values
- Enriched with lemmas tailored for caching

Composition and decorator pattern → verification efficiency

⇒ **Offering performant verified software**

Code: github.com/epfl-lara/bolts/tree/main/data-structures/maps/mutablemaps Contact: samuel.chassot@epfl.ch

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]
```

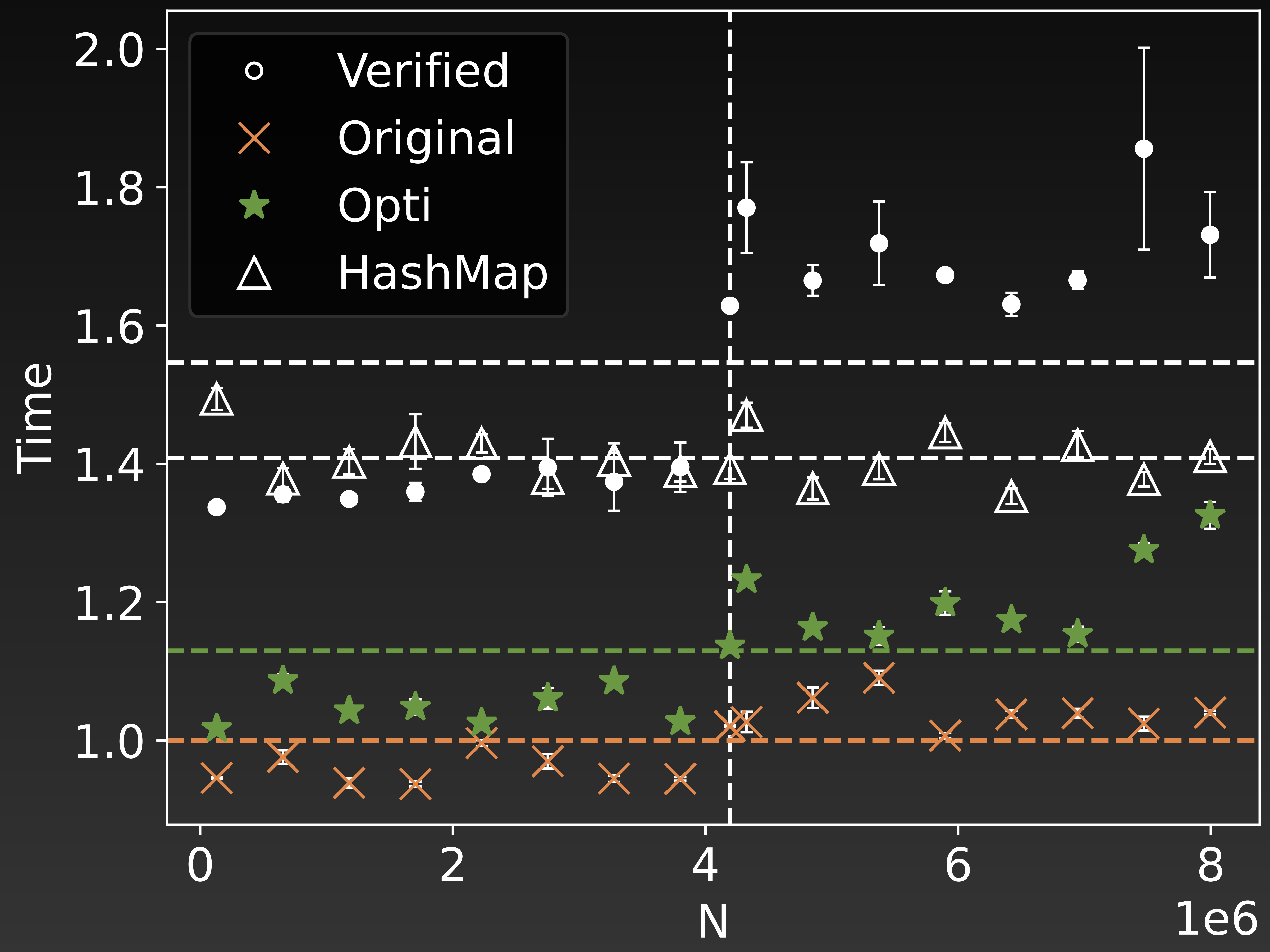
- Our version

```
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

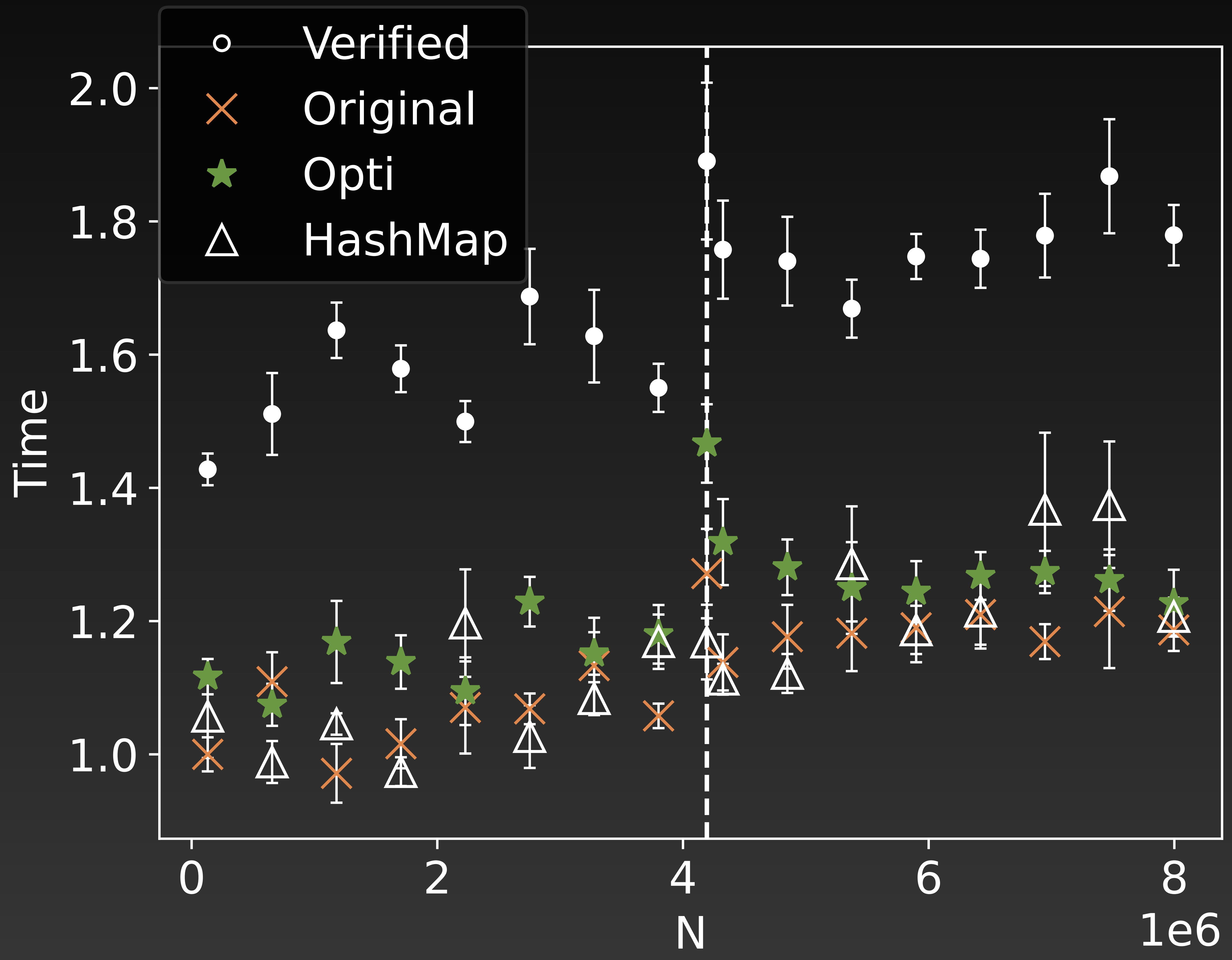
Scenario 1: lookup in pre-populated



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

Performance Evaluation

Scenario 3: population with remove + lookups



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