A JVM for Soft-Error-Prone Embedded Systems

Isabella Stilkerich, Michael Strotz, Christoph Erhardt, Martin Hoffmann, Daniel Lohmann, Fabian Scheler, Wolfgang Schröder-Preikschat

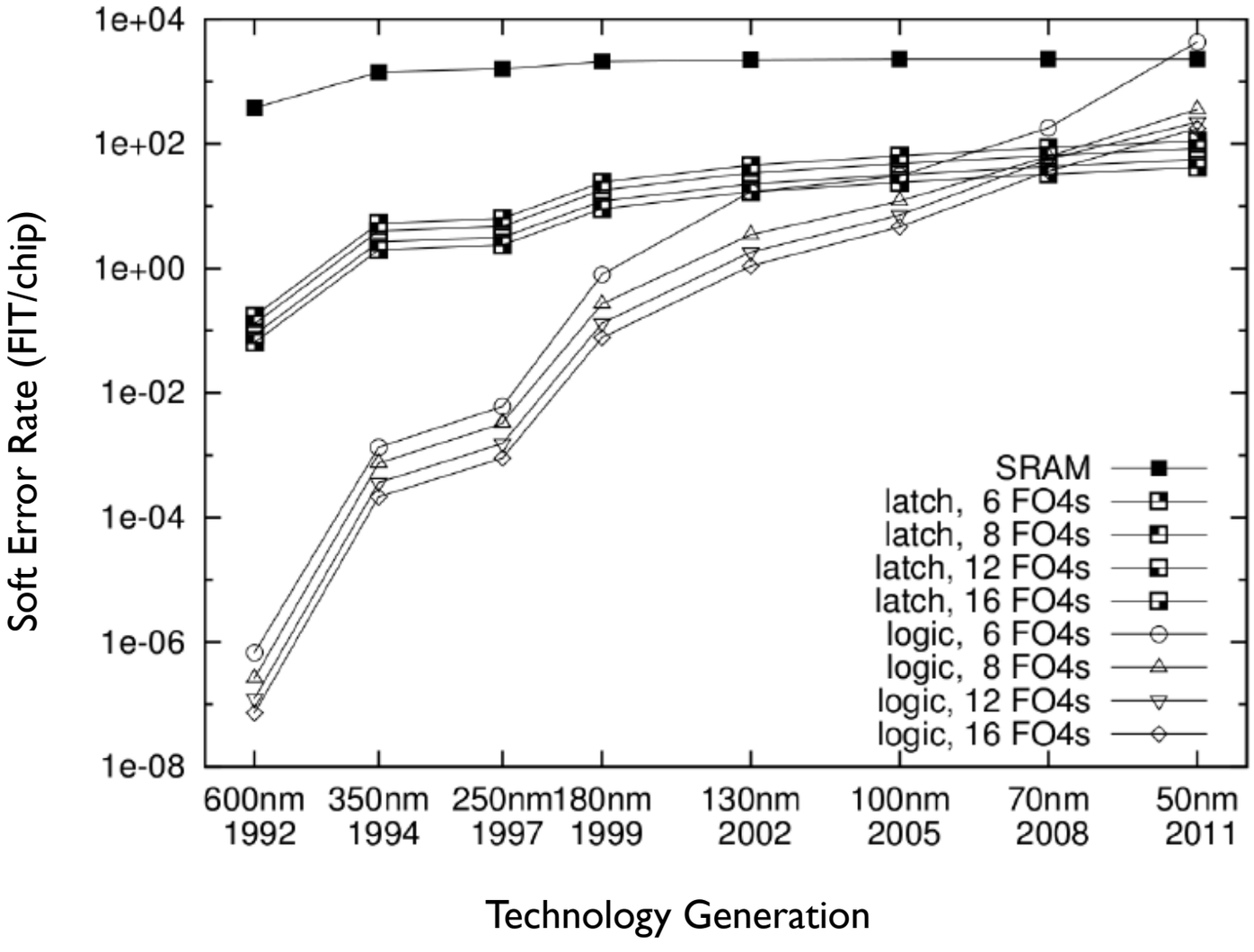
Department of Computer Science 4
Distributed Systems and Operating Systems

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Motivation

Motivation

- Transient hardware faults become more likely
  - Soft error rate in logic has increased by 9 orders of magnitude
  - Soft error rate in SRAM is constantly high
  - Soft errors cannot be ignored anymore

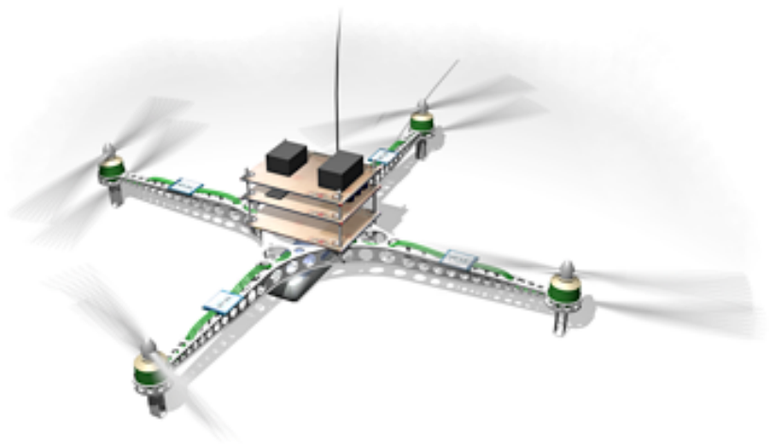


Motivation

- Transient hardware faults become more likely
  - Soft error rate in logic has increased by 9 orders of magnitude
  - Soft error rate in SRAM is constantly high
  - Soft errors cannot be ignored anymore
- Hardware-based fault tolerance (FT) techniques
  - Expensive: size, weight and power

Motivation

- Transient hardware faults become more likely
  - Soft error rate in logic has increased by 9 orders of magnitude
  - Soft error rate in SRAM is constantly high
  - Soft errors cannot be ignored anymore
- Hardware-based fault tolerance (FT) techniques
  - Expensive: size, weight and power
- Software-based fault tolerance (FT) techniques



Motivation (2)

- Most techniques concentrate on application protection
- Previous work: employ a type-safe language (Java) for SW-based FT
  - Use the type system for automatic ...
    - Isolate (SW-based memory protection)
    - Replication of critical modules
    - State protection and recovering
- Primary focus on protecting the application
  - System software is considered trusted compute base
  - Bit flips can corrupt system software, SW-based memory protection

Type Safety and Software Isolation

- Type safety ensures memory safety in absence of transient errors
  - Only explicit references can be used
  - No access outside an object's boundary
  - Type determines in which way memory is used
- Our software isolation approach is based on
  - Type safety
  - Logical separation of global data
- **But bit flips can break type safety and software isolation!**
- **How can we harden the JVM itself?**

Agenda

What can be done to preserve type safety and software isolation in the face of soft errors?

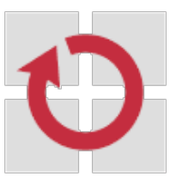
Can the characteristics of Java be useful to achieve this goal?

Case study with the embedded KESO Java Virtual Machine

1. Hardening the KESO JVM
2. Tailoring the safety measures
3. Evaluation

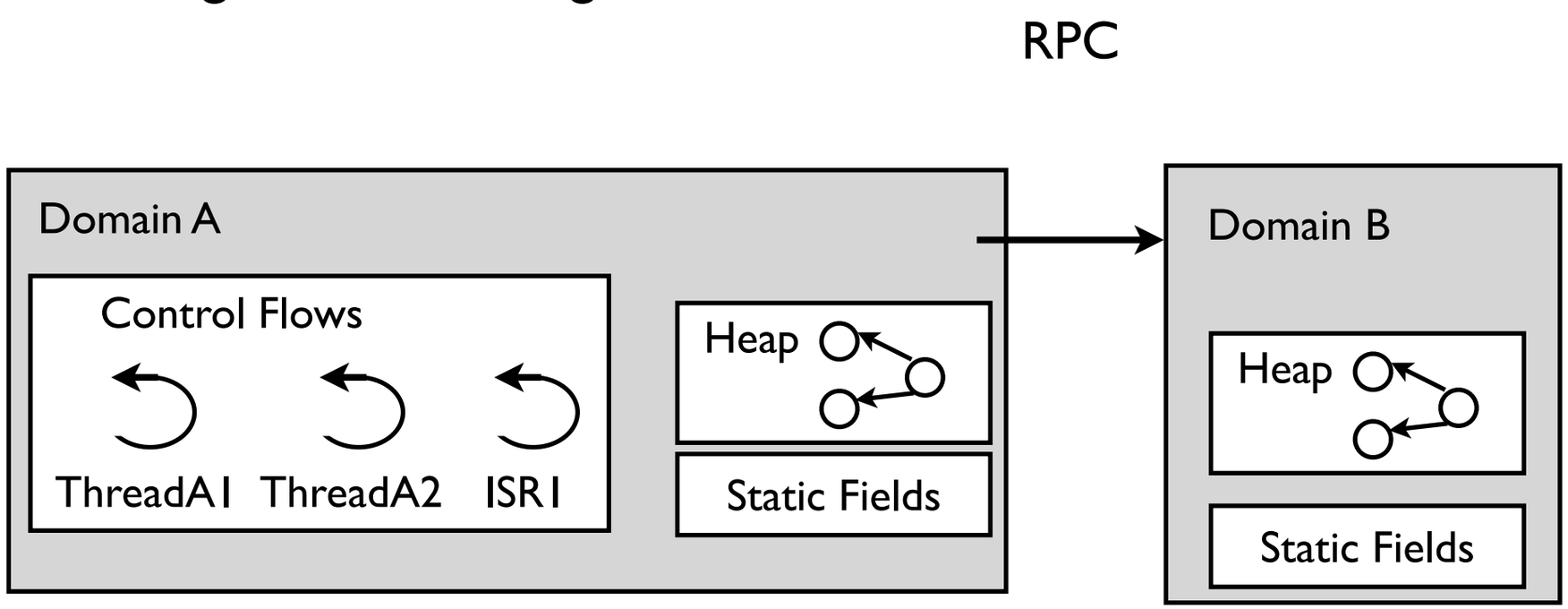
Fault Model

- Soft errors visible at the programming interface of the processor
- Focus of this work: **protect the JVM**
  - Protection of application data achieved on higher abstraction level
- Trusted memory base: read-only memory (ROM)
  - More robust against soft errors
  - Program code and constant data are considered to be safe
  - G. Cellere et al.: Neutron-induced soft errors in advanced flash memories (IEDM 2008)

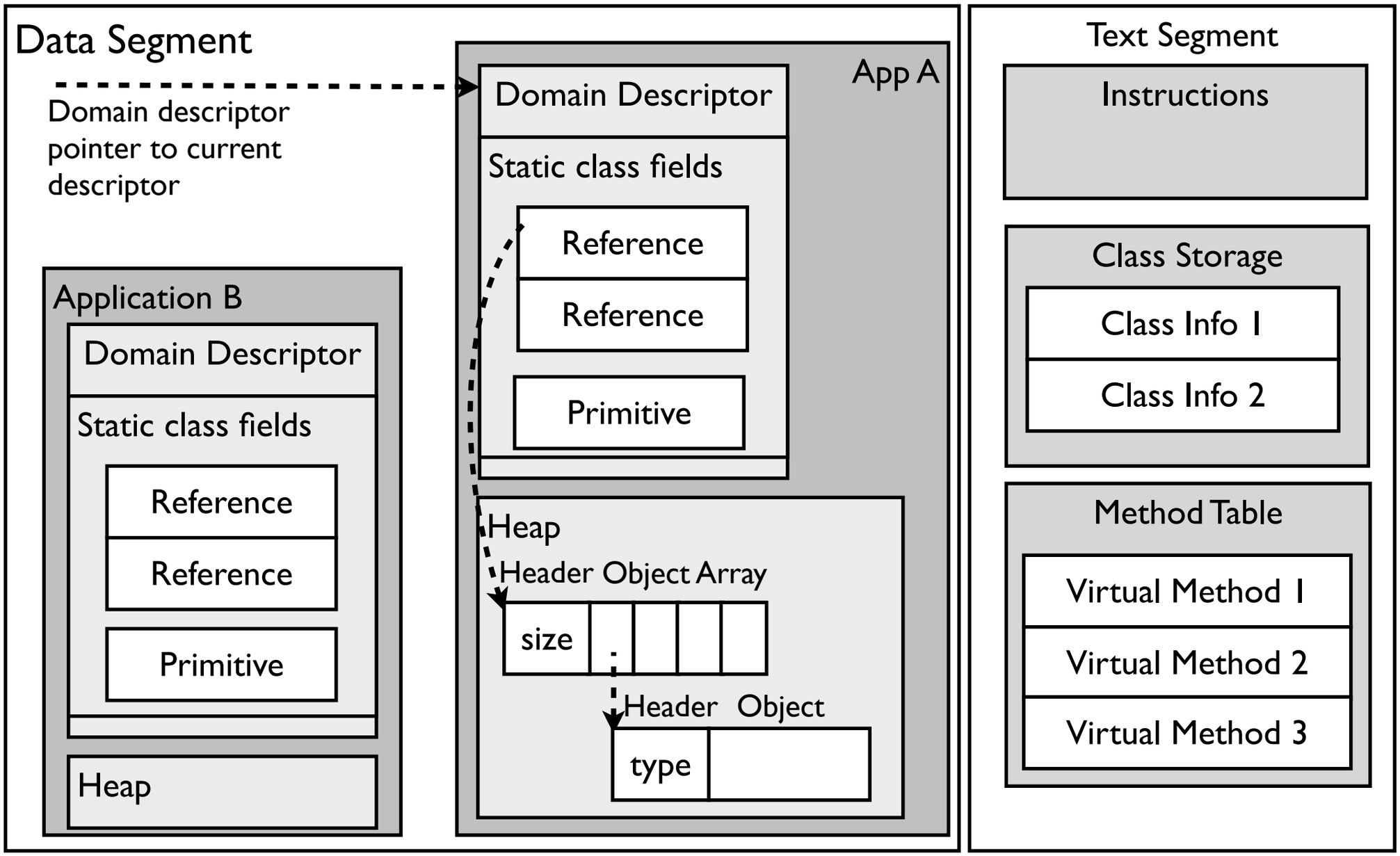


The KESO JVM

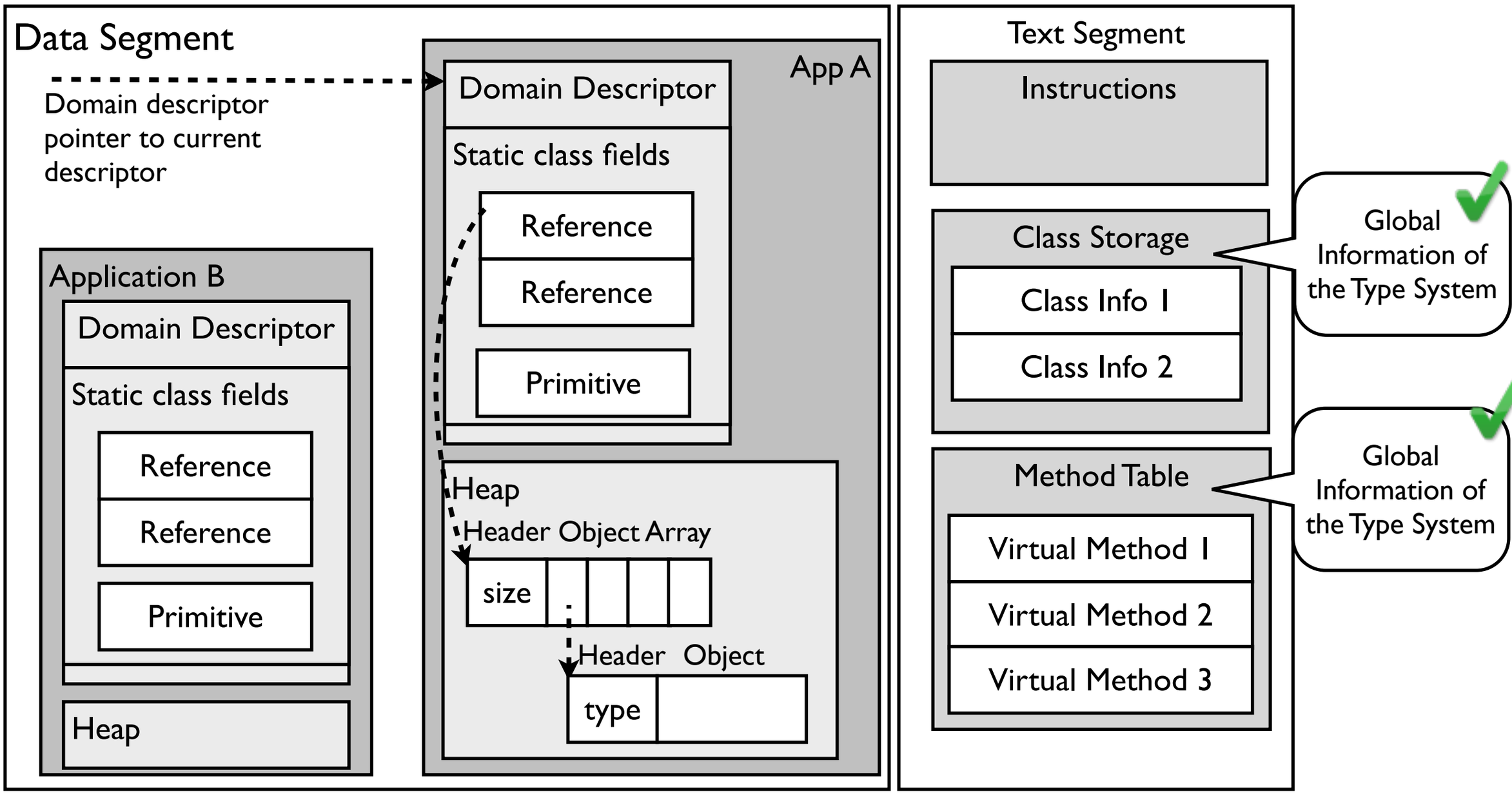
- Java-to-C ahead-of-time compiler
- VM tailoring, static configuration



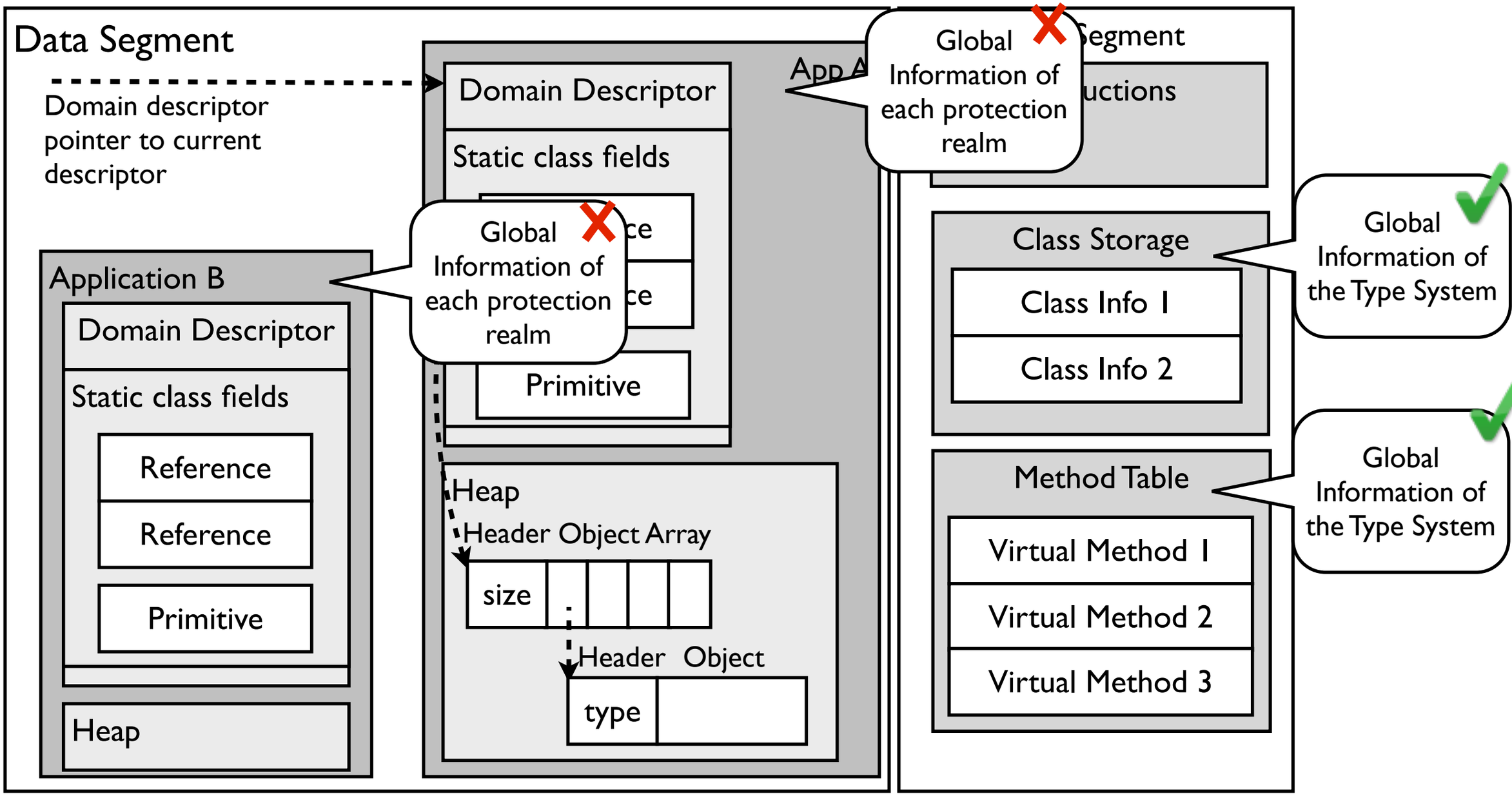
Runtime Information of KESO



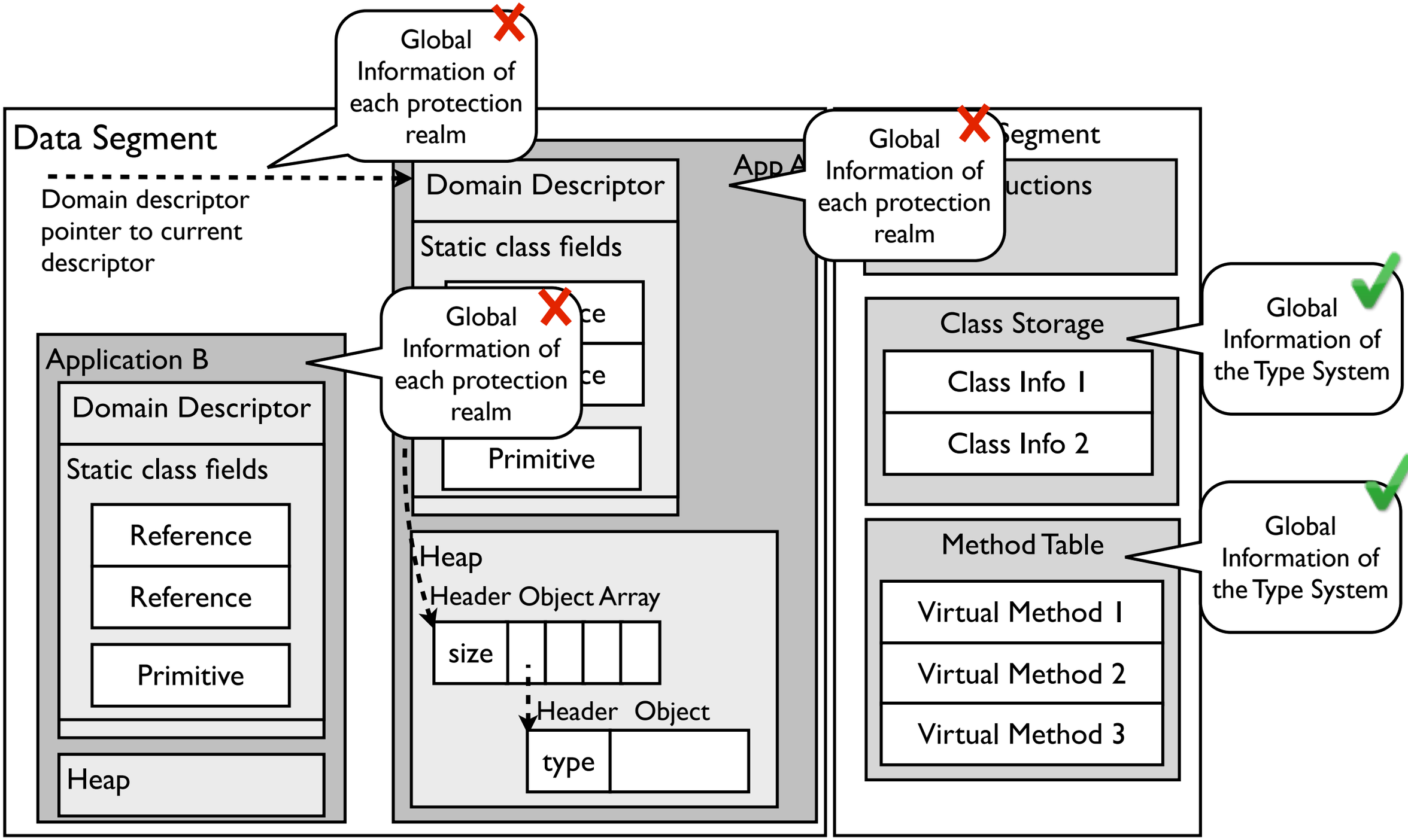
Runtime Information of KESO



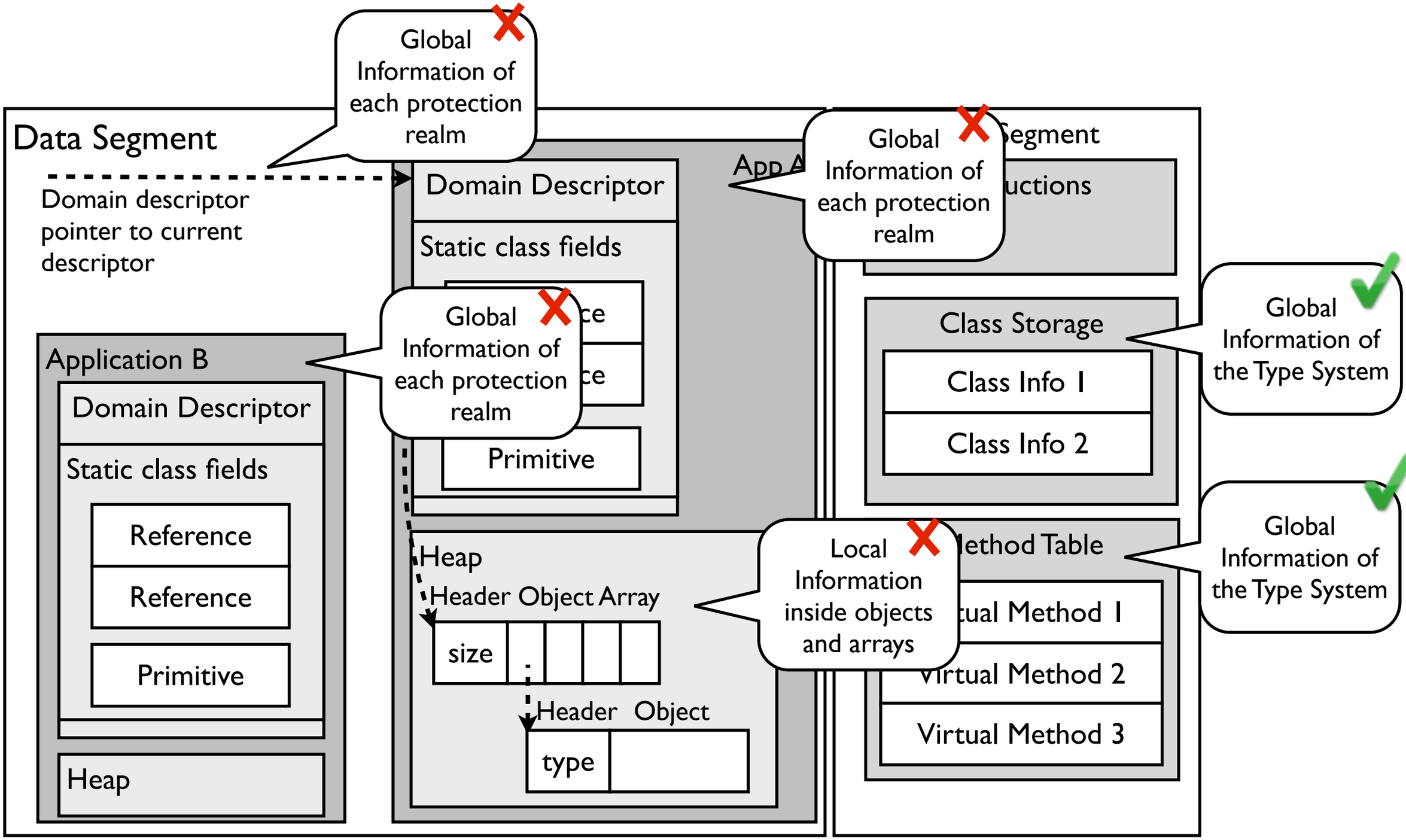
Runtime Information of KESO



Runtime Information of KESO

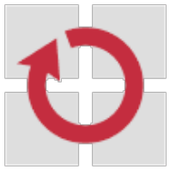


Runtime Information of KESO



What can go wrong?

- Wild references
- Corrupted type information in object headers
- Invalid current-domain pointer

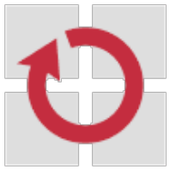


Wild References

- Internal implementation of Java reference: memory address
  - Corrupted reference = wild pointer

Wild References

- Internal implementation of Java reference: memory address
  - Corrupted reference = wild pointer
- Worst-case consequence: access to arbitrary memory location



Wild References

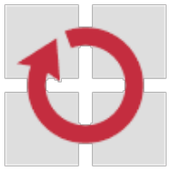
- Internal implementation of Java reference: memory address
  - Corrupted reference = wild pointer
- Worst-case consequence: access to arbitrary memory location

- Solution: validate references upon access
  - e.g. using parity/checksum
  - Type-safe language: all reference accesses are known ahead of time

Wild References

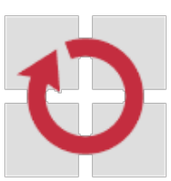
- Internal implementation of Java reference: memory address
  - Corrupted reference = wild pointer
- Worst-case consequence: access to arbitrary memory location

- Solution: validate references upon access
  - e.g. using parity/checksum
  - Type-safe language: all reference accesses are known ahead of time
- But when should we check references?



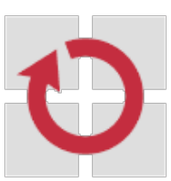
Dereference Check (DRC)

- References stay encoded all the time
- Checked, whenever they are dereferenced
- Advantage:
  - High error detection rate
- Drawbacks:
  - Checked more often than necessary
  - Higher error detection time



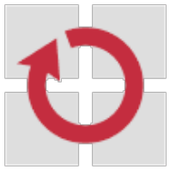
Load Reference Check (LRC)

- Checked at loads from memory
- Static reference fields, object ref. fields, reference arrays
- Advantage:
  - Faster than DRC
  - Shorter error detection time
- Drawbacks:
  - May wrongly assume correct value

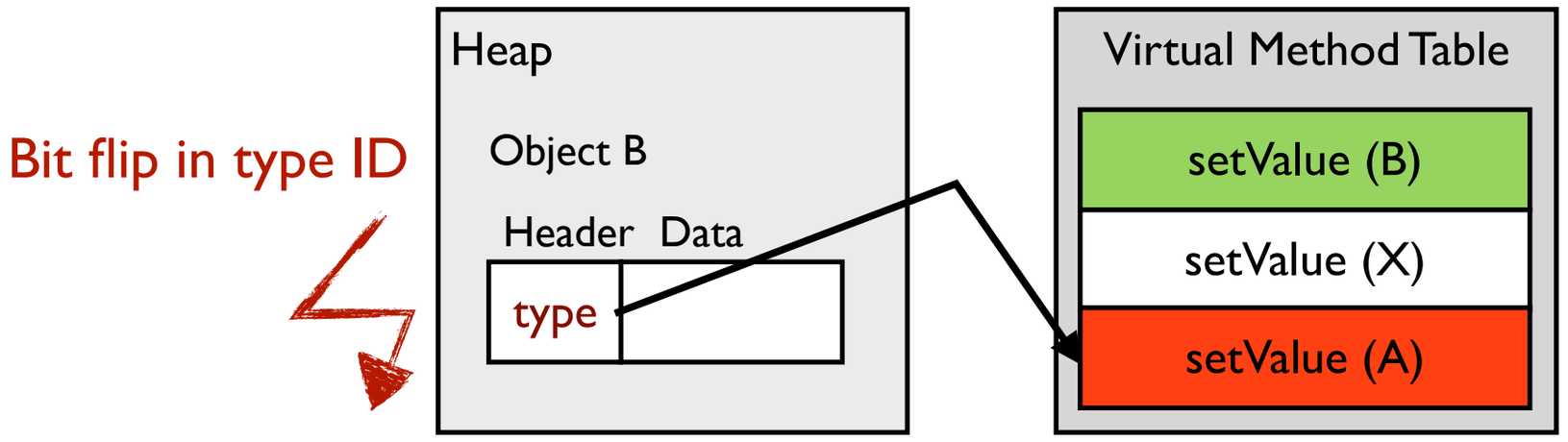


Header-Only Check (HOC)

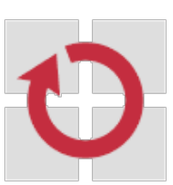
- Check header only upon dereference
- Advantage:
  - Faster than LRC
- Drawbacks:
  - May wrongly assume correct value in case of traditional memory layout
  - Requires special memory layout to reduce false negatives, higher memory consumption



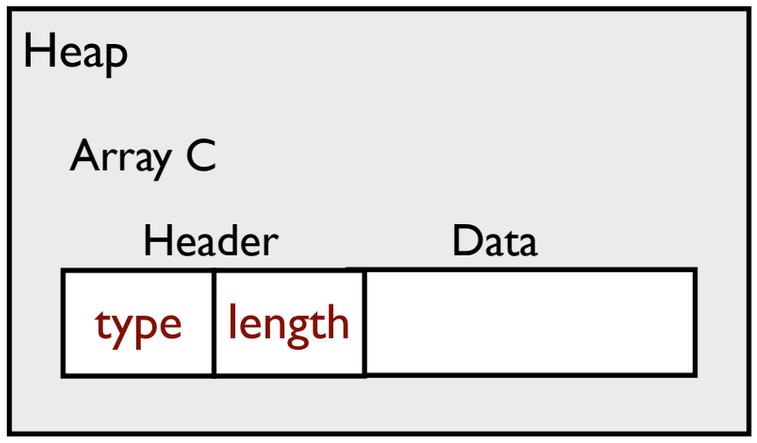
Corrupted Object Meta-Information



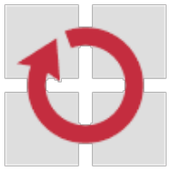
- Corruption of meta-information in object header possible
- Critical instructions:
  - Direct use of type ID: `checkcast, instanceof`
  - Table lookup with type ID as index: `invokevirtual, invokeinterface`
- Worst-case consequences: invalid cast, invocation of wrong method
- Solution: validate object header upon access
  - e.g. using parity/checksum



Corrupted Array Meta-Information



- Type ID: same issues as with plain objects
- Corrupted array length: bounds check may no longer work as expected
  - False positive: Out-of-bounds exception in spite of correct index
  - **False negative: No exception in spite of incorrect index!**
- Worst-case consequence: access to arbitrary array index
- Solution: validate array length before bounds check
  - e.g. using parity/checksum



Array Accesses

Array Accesses

- Results of data-flow analysis are used to address several cases:

Array Accesses

- Results of data-flow analysis are used to address several cases:
  1. Index and size are constant, no bounds check needed

Array Accesses

- Results of data-flow analysis are used to address several cases:
  1. Index and size are constant, no bounds check needed

```
int [] array = new int[30];
...
array[5]=9;
```

Array Accesses

- Results of data-flow analysis are used to address several cases:
  1. Index and size are constant, no bounds check needed
  2. Size constant, index variable, simple bounds check needed

Array Accesses

- Results of data-flow analysis are used to address several cases:
  1. Index and size are constant, no bounds check needed
  2. Size constant, index variable, simple bounds check needed

```
int [] array = new int[30];

...

/* index is dynamically derived from input
source */

/* simple array bounds check needed */

array[index]=9;
```

Array Accesses

- Results of data-flow analysis are used to address several cases:
  1. Index and size are constant, no bounds check needed
  2. Size constant, index variable, simple bounds check needed
  3. Access is within bounds, but either index or size is not constant, size information has to be checked: **Extended array bounds check (EBC)**

Array Accesses

- Results of data-flow analysis are used to address several cases:
  1. Index and size are constant, no bounds check needed
  2. Size constant, index variable, simple bounds check needed
  3. Access is within bounds, but either index or size is not constant, size information has to be checked: **Extended array bounds check (EBC)**

```
int [] array = new int[inputValue];
...
for(int i=0; i<array.length; i++) {
  /* Extended Array Bounds Check */
  array[i]=i;
}
```

Array Accesses

- Results of data-flow analysis are used to address several cases:
  1. Index and size are constant, no bounds check needed
  2. Size constant, index variable, simple bounds check needed
  3. Access is within bounds, but either index or size is not constant, size information has to be checked: **Extended array bounds check (EBC)**
  4. Neither index nor size is constant: **EBC**

Invalid Current-Domain Pointer

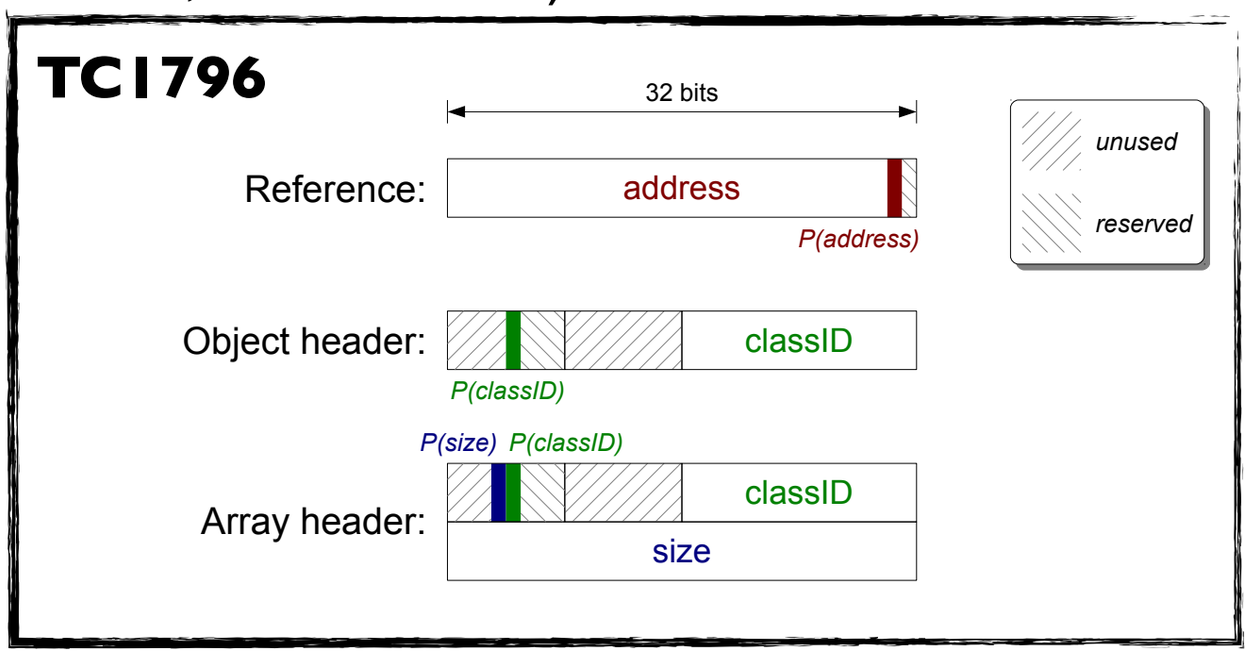
- Current-domain pointer can be corrupted
- Critical instructions:
  - `putstatic`, `getstatic`
  - Memory-management operations
  - ...
- Worst-case consequence: arbitrary memory accesses
- Solution: validate current-domain pointer upon access
  - e.g. using parity/checksum

Efficient Implementation

- Static programming model allows collecting extensive information ahead of time
  - Can be exploited by advanced high-level optimizations
- Optimizations support our purposes (attack-surface reduction, efficiency gains)
  - Constant propagation & folding: replaces (vulnerable) variables with constants, reduces register pressure
  - Runtime-check elimination: elides null-, bounds, **integrity checks**
  - Devirtualization: elides object-header checks
- Data-flow analysis: array accesses

Efficient Implementation (2)

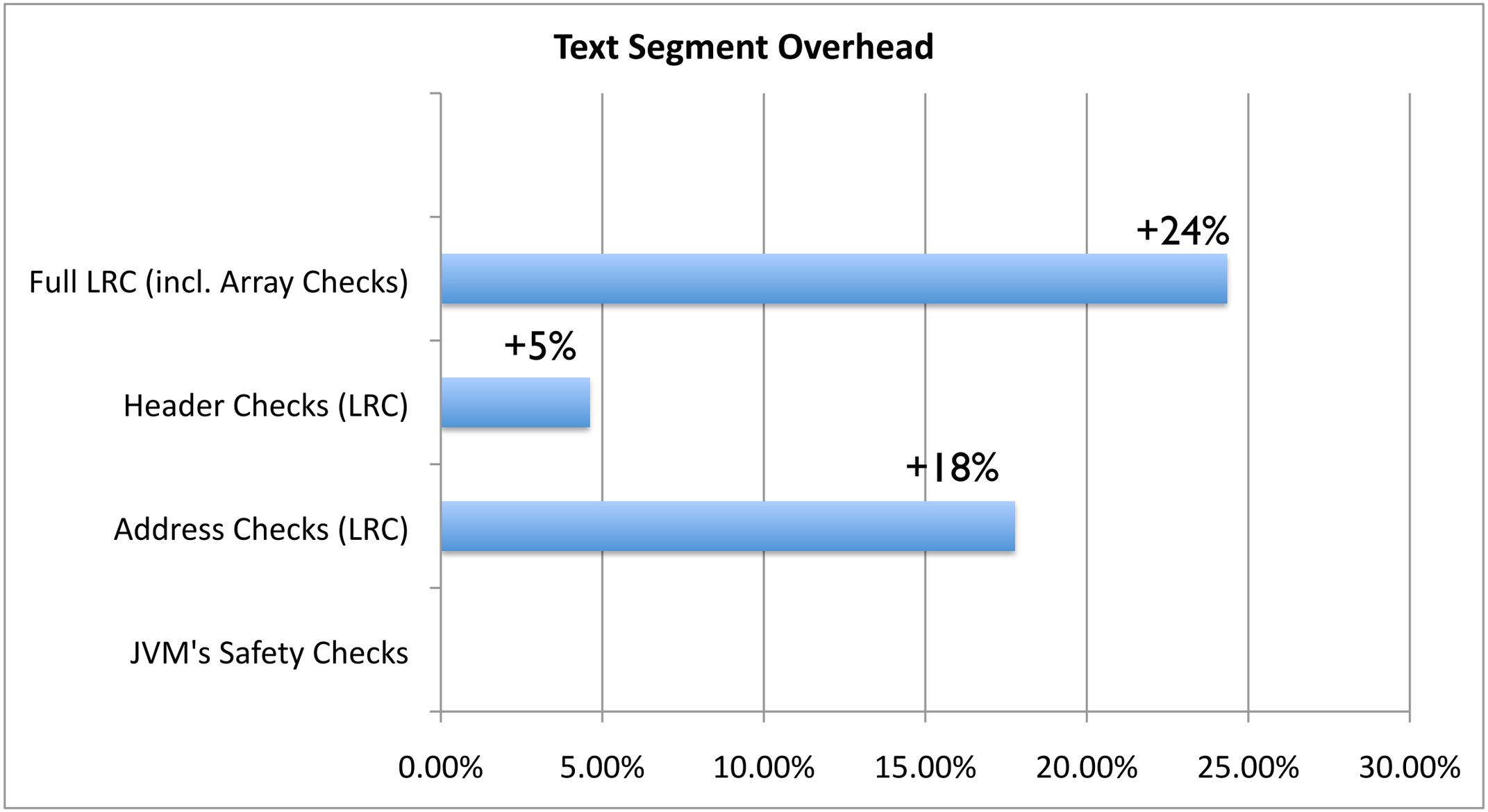
- Trusted memory base: ROM
  - Use data-flow analysis to derive constant data
  - Constant **application** and **system data** can be located in ROM
  - Immediate accesses are safe
- Incorporation of platform-specific features
  - Alignment and address layout: object headers
    - TriCore TC1796 (32-bit, 1 MiB SRAM, 2 MiB ROM)
    - AVR ATmega8535 (8-bit, 512 B SRAM, 8 KiB ROM)
  - Memory protection unit (MPU)
    - Safety net
    - Use hardware exceptions



Evaluation

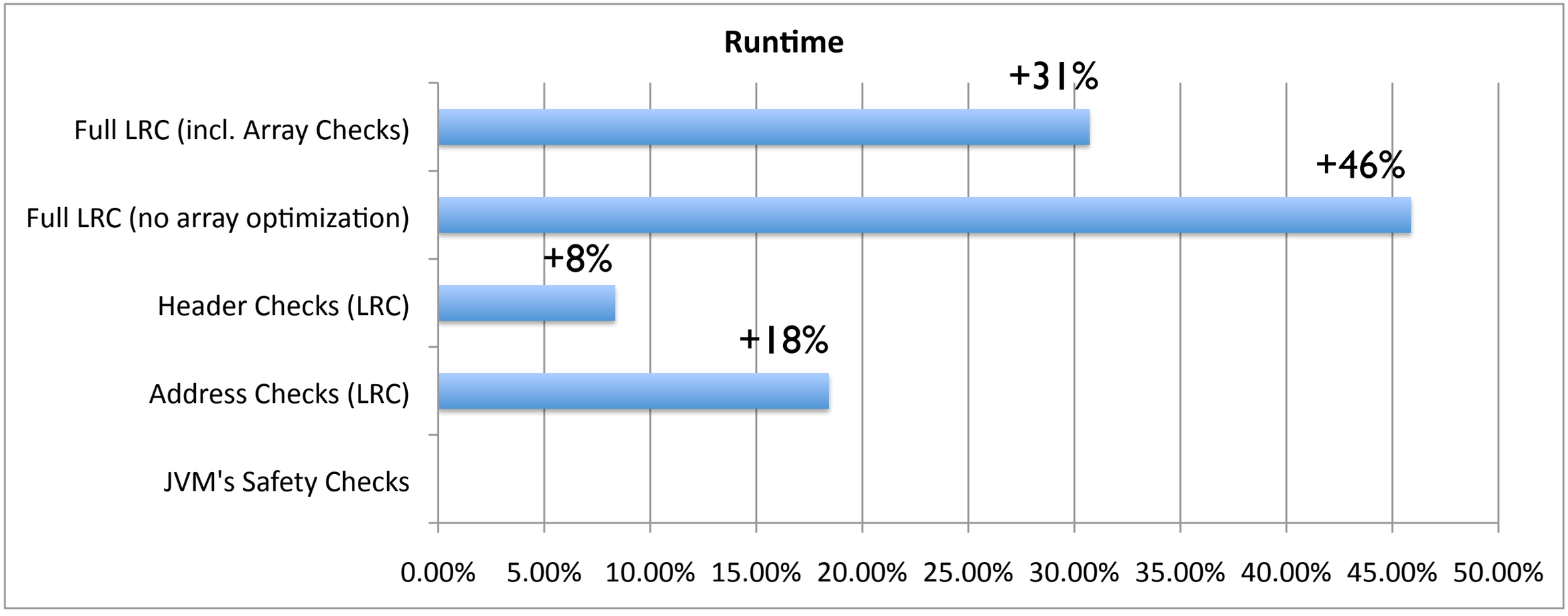
- Real-time Java application benchmark Collision Detector (CDx)
- Built KESO CDx variants
  - No runtime checks
  - Null and array bound checks (**Safety checks enabled**)
  - **LRC** (Reference and type integrity checking prior to safety checks)
- Measured overhead (memory footprint, runtime)
- Injected faults in main computation loop with *Fail** tool
  - Fault space pruning
  - Grouped domain data by compiler's reachability analysis

Evaluation - Footprint @ TC1796



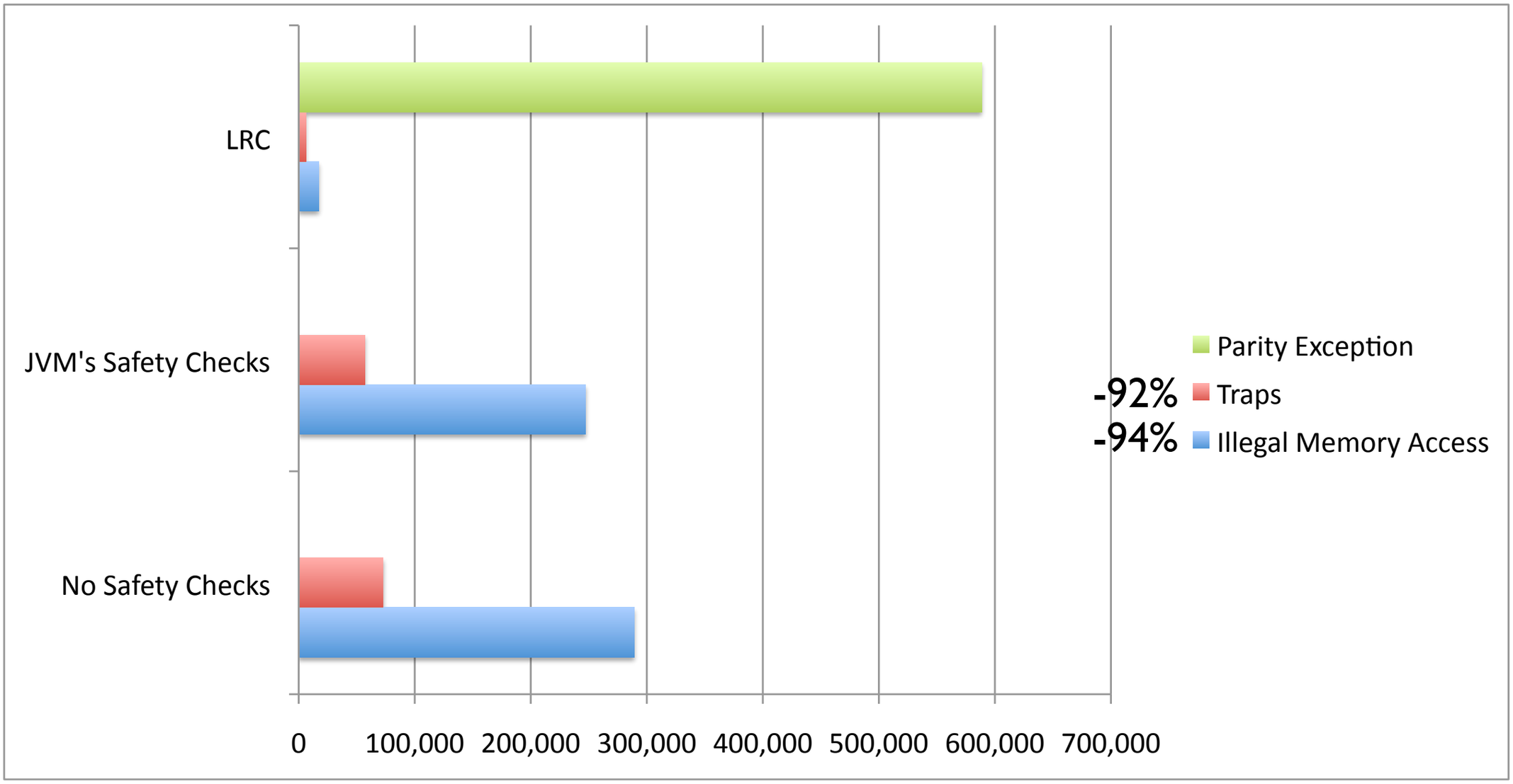
- No size changes in .data or .bss segment

Evaluation - Runtime @ TC1796



- Example: Deactivated array optimization (always EBC)

Evaluation - Fault Injection



- More numbers can be found in the paper

Conclusion

What can be done to preserve type safety and software isolation in the face of soft errors?

- Protect global type-system information (method table, class table, current-domain pointer)
- Check global and local references (addresses, type information)

Can the characteristics of Java be useful to achieve this goal?

- Separation of references and data values, type information
  - Explicit protection of references
  - Runtime-structural information for automated protection of application data
- Software-based memory protection

Future Work

- Improve program and system analyses and code generation
  - Loop unrolling to elide array bounds checks
  - Object header elision due to address layout peculiarities
    - Only address checking needed
- ROM allocation: Application data protection and runtime check elision
- Fragmented allocation
  - Use of software or hardware exceptions
  - No address checking needed, only header checks
- Incorporation of hardware characteristics (ROM/MPU/Memory layout)

Questions?

- http://www4.cs.fau.de/Research/KESO/
- KESO: distributed under the terms of the GNU LGPL, version 3