High System-Code Security with Low Overhead

Jonas Wagner, Volodymyr Kuznetsov, George Candea, and Johannes Kinder

École Polytechnique Fédérale
de Lausanne

Royal Holloway,
University of London

High System-Code Security?

Today’s software is dangerous.

Example: OpenSSL
Overflow in ssl/t1_lib.c:3997 →

OpenSSL contains 53,073 memory accesses.
How to protect them all?

Protect *all* dangerous operations using sanity checks

✓ *Checks are automatically added at compile time*

✓ *No source code modification is needed*

```
*p = 42;
```


```
if (!isValidAddress(p)) {
  reportError(p);
  abort();
}
*p = 42;
```

Problem: Sanity checks cause high performance overhead

Tool	Avg. Overhead
AddressSanitizer (memory errors)	73%
SoftBound/CETS (full memory safety)	116%
UndefinedBehaviorSanitizer (integer overflows, type errors…)	71%
Assertions, code contracts, …	depends

Problem: Sanity checks cause high performance overhead

People use checks heavily for testing,
but disable them in production

Goal: checks in production

Insight: Checks are not all equal

Most of the *overhead* comes from a few expensive checks

Checks in hot code,
each executed many times

Most of the *protection* comes from many cheap checks

Checks in cold code

Our Approach: ASAP

As Safe As Possible

Lets users choose their *overhead budget* (e.g., 5%)

Automatically identifies sanity checks in software

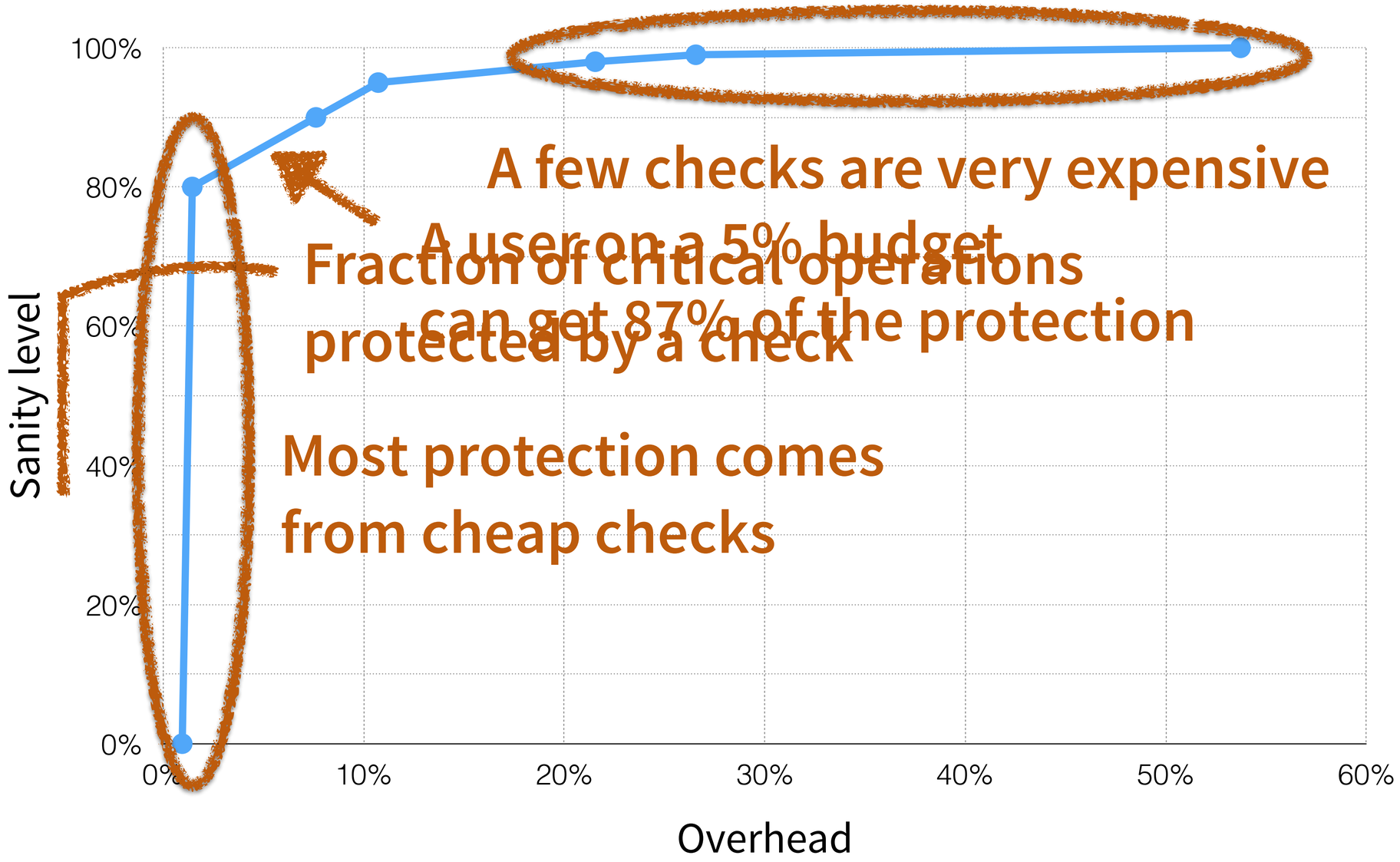
Analyzes the cost of every check

Selects as many checks as fit in the user's budget

Most of the *overhead* comes from a few expensive checks

Most of the *protection* comes from many cheap checks

ASAP Insight & Results



Outline

Introduction: What is ASAP?

Design

Key Algorithms

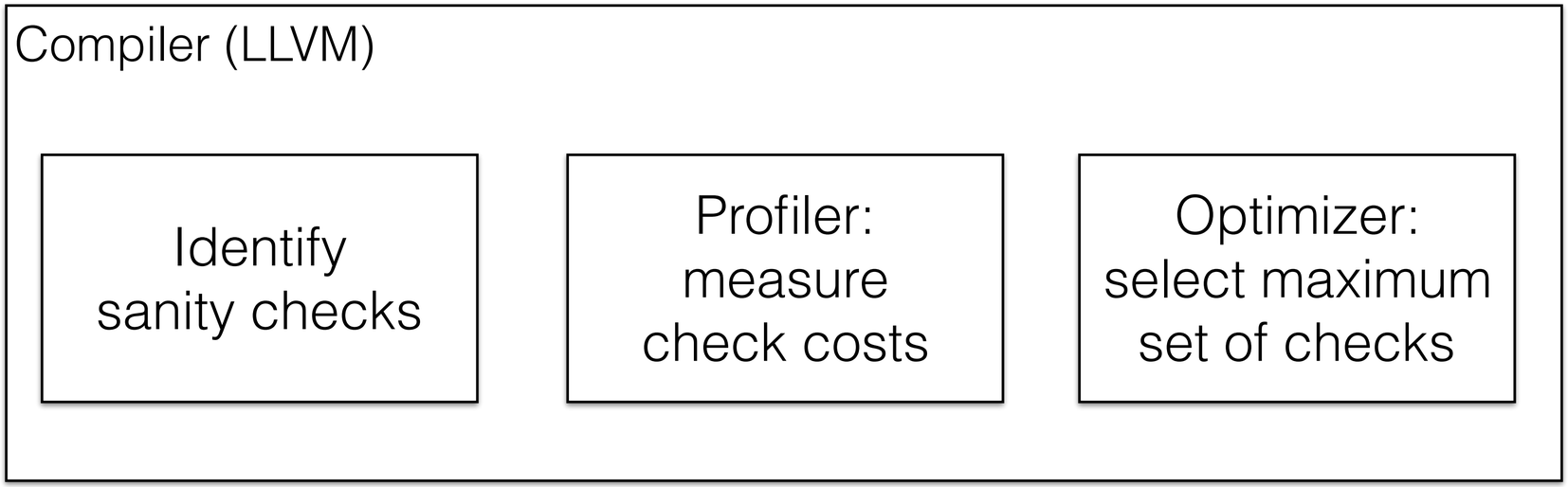
Results

Conclusion

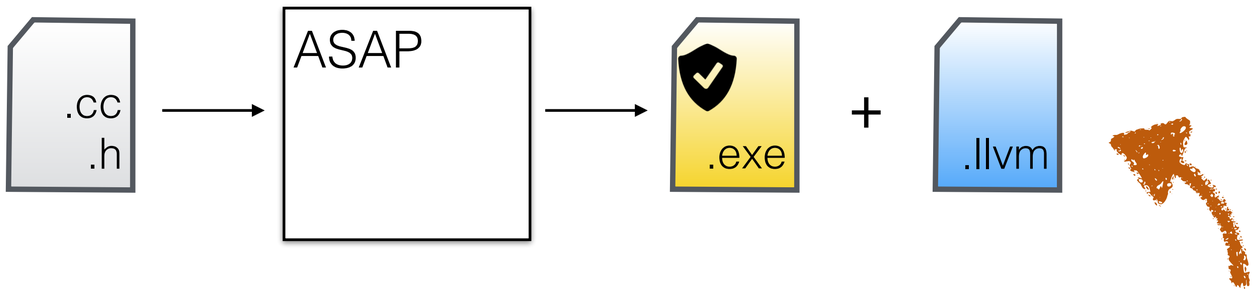
Design

ASAP is built into the compiler

- ✓ *Easy to use (set CC and CFLAGS)*
- ✓ *Compatible (parallel compilation, …)*



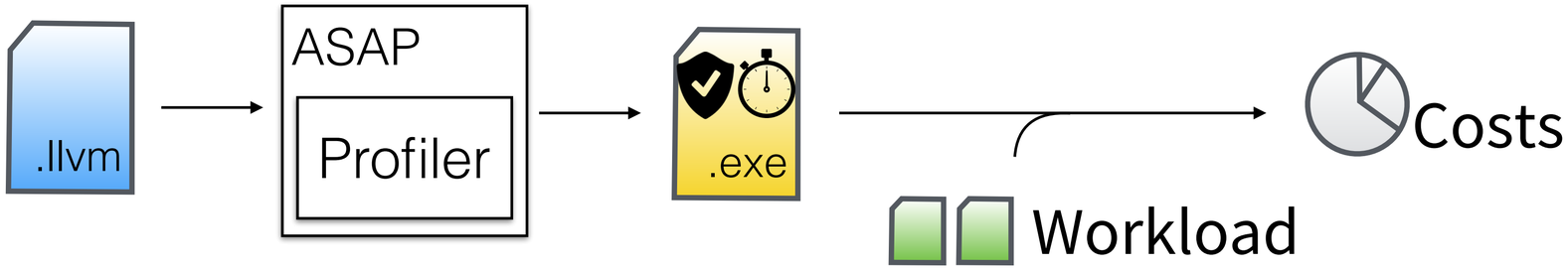
Users use ASAP like a regular compiler that adds checks



ASAP stores intermediate compiler output

Users use ASAP like a regular compiler that adds checks

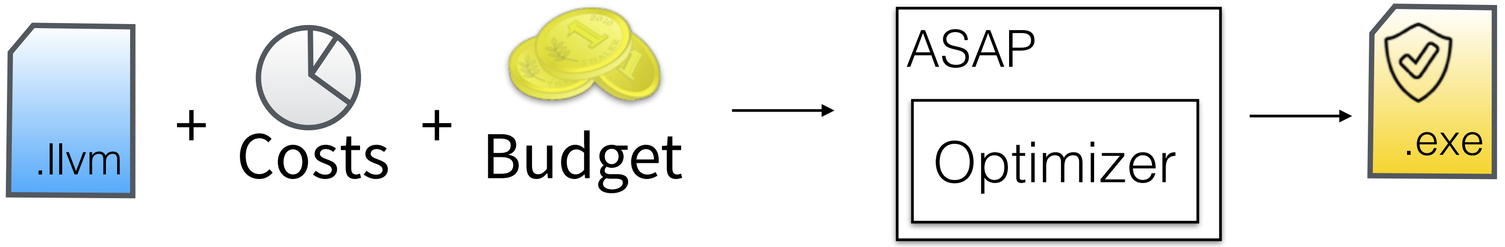
ASAP generates a program variant with profiling instrumentation. Users run this to measure check costs.



Users use ASAP like a regular compiler that adds checks

ASAP generates a program variant with profiling instrumentation. Users run this to measure check costs.

ASAP uses costs & budget to generate an optimized program



Outline

Introduction: What is ASAP?

Design

Key Algorithms

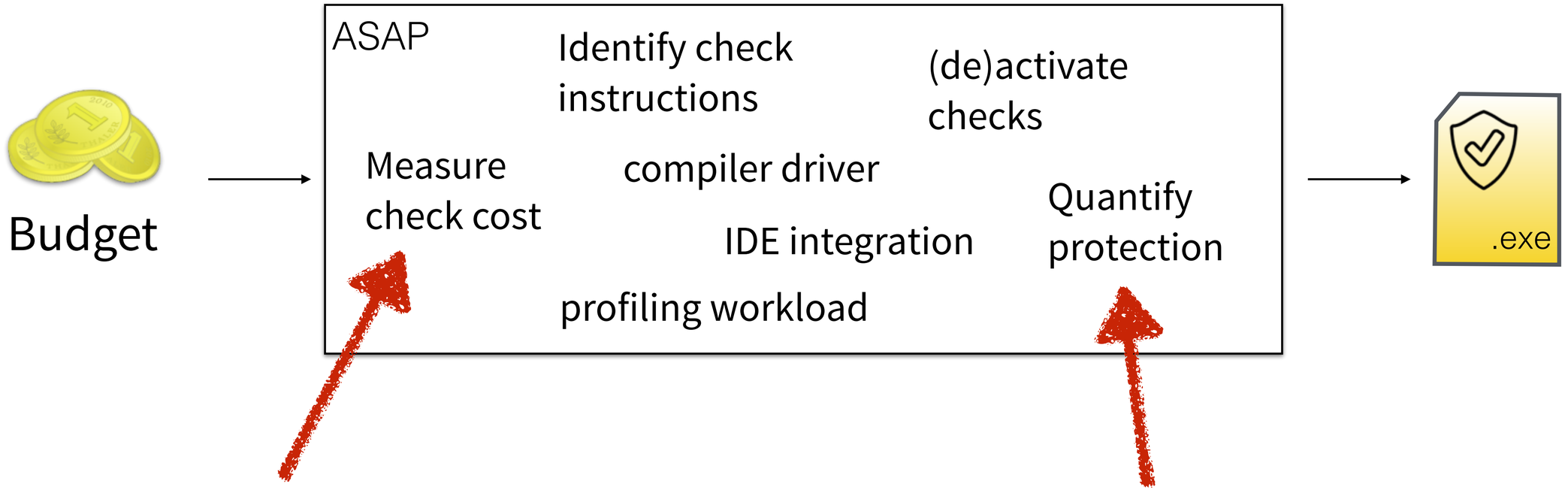
Results

Conclusion

Budget

ASAP

.exe



Can users trust ASAP to select checks that use the least CPU cycles?

If ASAP says you're 87% protected, what does this mean?

Measure Check Cost

```
...
if (!isValidAddress(p)) {
  reportError(p);
  abort();
}
*p = 42;
...
```

Measure Check Cost

```
...
prof1++;
if (!isValidAddress(p)) {
  prof2++;
  reportError(p);
  abort();
}
prof3++;
*p = 42;
...
```

1. Add **profiling counters**

Measure Check Cost

```
...
prof1++;
if (!isValidAddress(p)) {
  prof2++;
  reportError(p);
  abort();
}
prof3++;
*p = 42;
...
```

1. Add **profiling counters**
2. Identify **check instructions**

Measure Check Cost

```
...
prof1++;
if (!isValidAddress(p)) {
  prof2++;
  reportError(p);
  abort();
}
prof3++;
*p = 42;
...
```

1. Add **profiling counters**
2. Identify **check instructions**
3. Use static model
   of **cycles per instruction**

Measure Check Cost

```
...
prof1++;
if (!isValidAddress(p)) {
  prof2++;
  reportError(p);
  abort();
}
prof3++;
*p = 42;
...
```

1. Add **profiling counters**
2. Identify **check instructions**
3. Use static model of **cycles per instruction**
4. Compute cost for check in CPU cycles:

$$\sum_{i \in \text{check}} \text{prof}(i) \cdot \text{cycles}(i)$$

Precise cost in CPU cycles

Budget

ASAP

Measure check cost

Identify check instructions

(de)activate checks

compiler driver

IDE integration

profiling workload

Quantify protection

.exe

If ASAP says you're 87% protected, what does this mean?

ASAP quantifies protection using the *sanity level*

We would like to know the *effective protection level*

Methodology: measure how many bugs/vulnerabilities are effectively prevented

ASAP quantifies protection using the *sanity level*

We would like to know the *effective protection level*

Methodology: measure how many bugs/vulnerabilities are effectively prevented

Experiment 1

Source code of Python 2.7

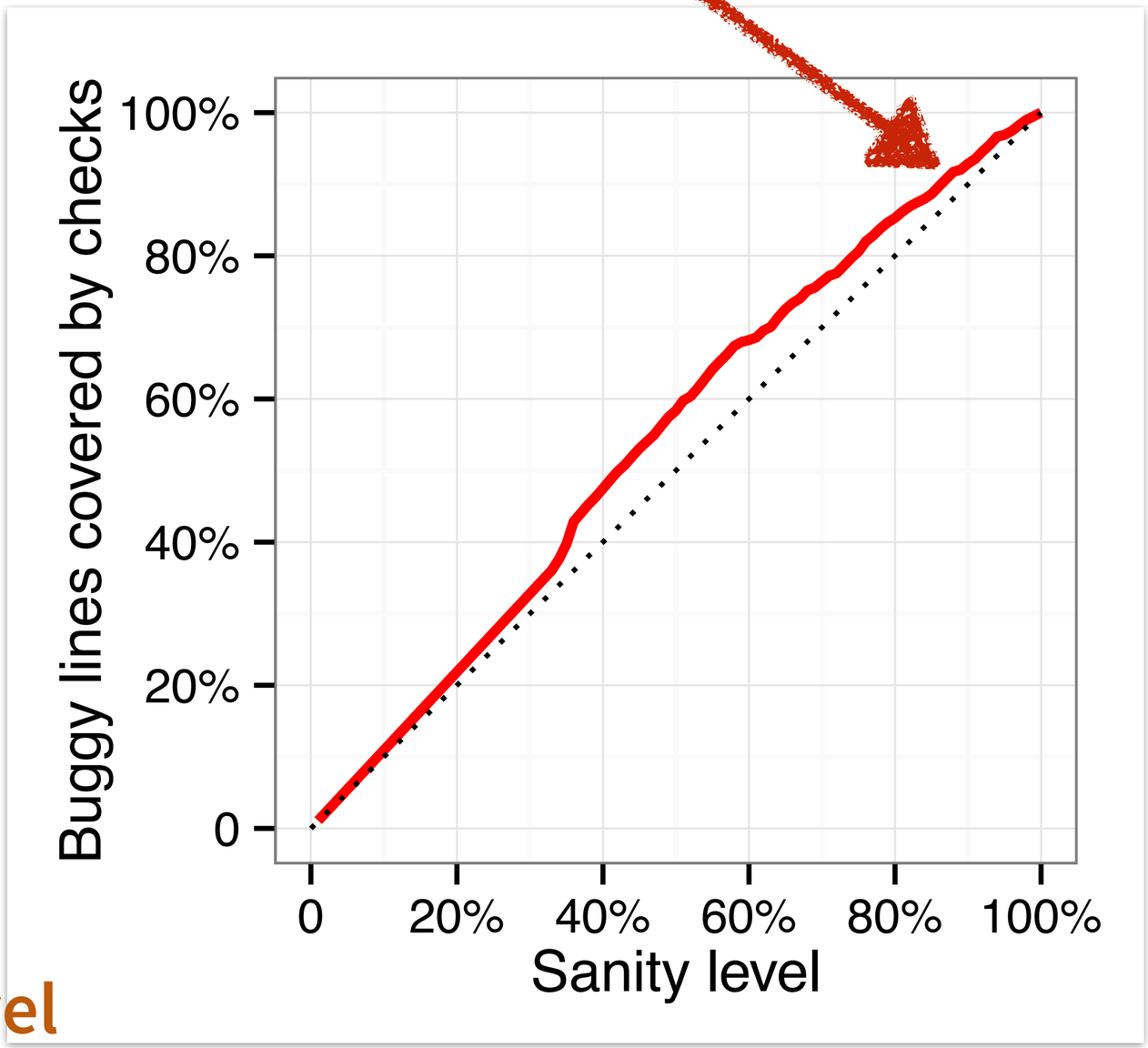
Bug: line that has received a patch between version 2.7.1 and 2.7.8

Sanity level 87% ≈ 91% protection

ASAP quantifies protection using the *sanity level*

We would like to know the *effective protection level*

Methodology: measure how many bugs/vulnerabilities are effectively prevented



Effective Protection ≥ sanity level

Experiment 2

Known bugs

Project	Bugs
Python 3.4	3
OpenSSL	1
RIPE Benchmarks	10

All of these are in cold code

Experiment 2

Known bugs

Project	Bugs
Python 3.4	3
OpenSSL	1
RIPE Benchmarks	10

All of these are in cold code

Experiment 3

Vulnerabilities from CVE DB

Analyze 145 vulnerabilities from 2014

- *Memory errors*
- *Open source*
- *Patch available*
- *Error location known*

83% of these are in cold code

Checks in cold code provide real value

Outline

Introduction: What is ASAP?

Design

Key Algorithms

Results

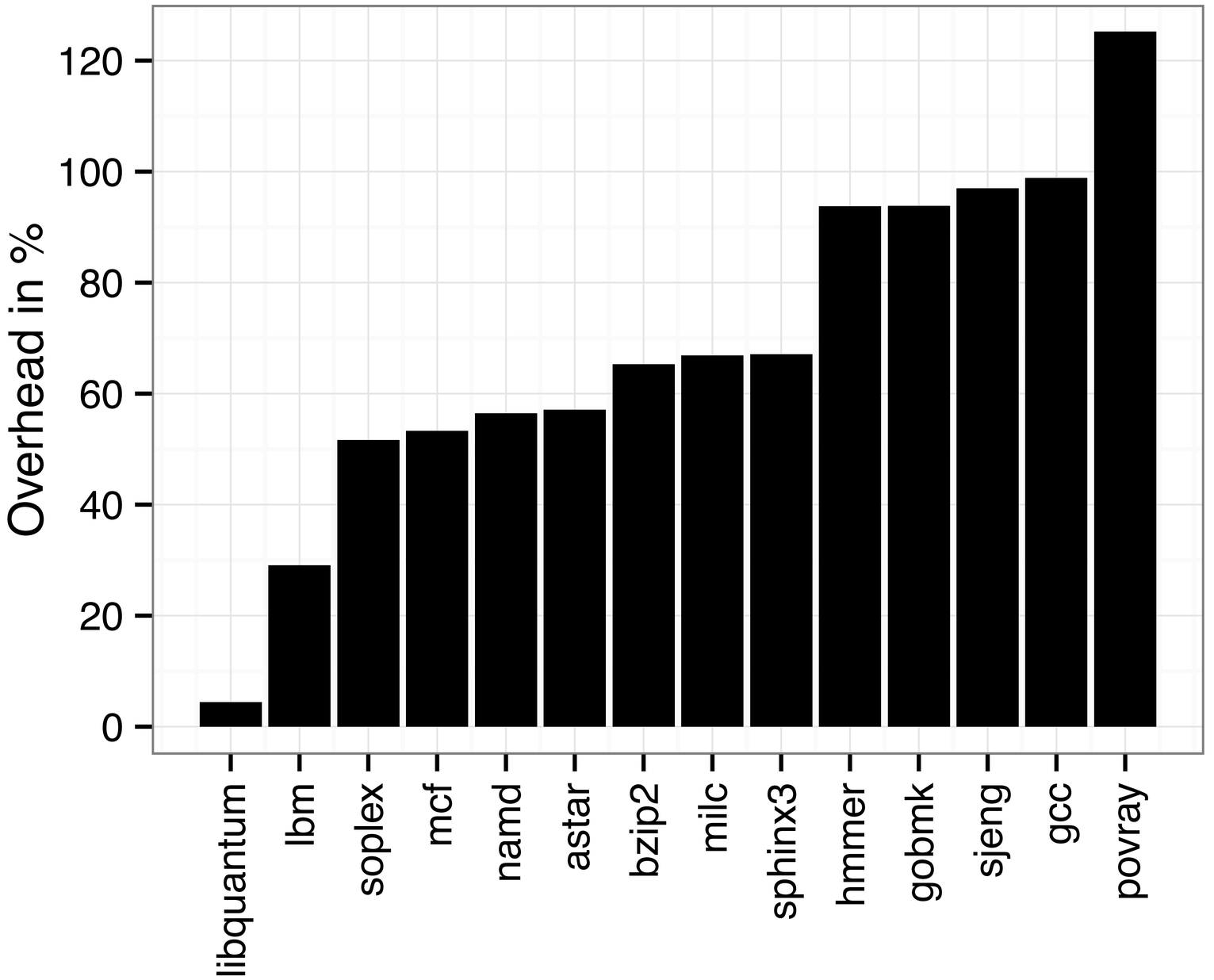
Conclusion

Results

Overhead for:
SPEC benchmarks
AddressSanitizer

100%

sanity level

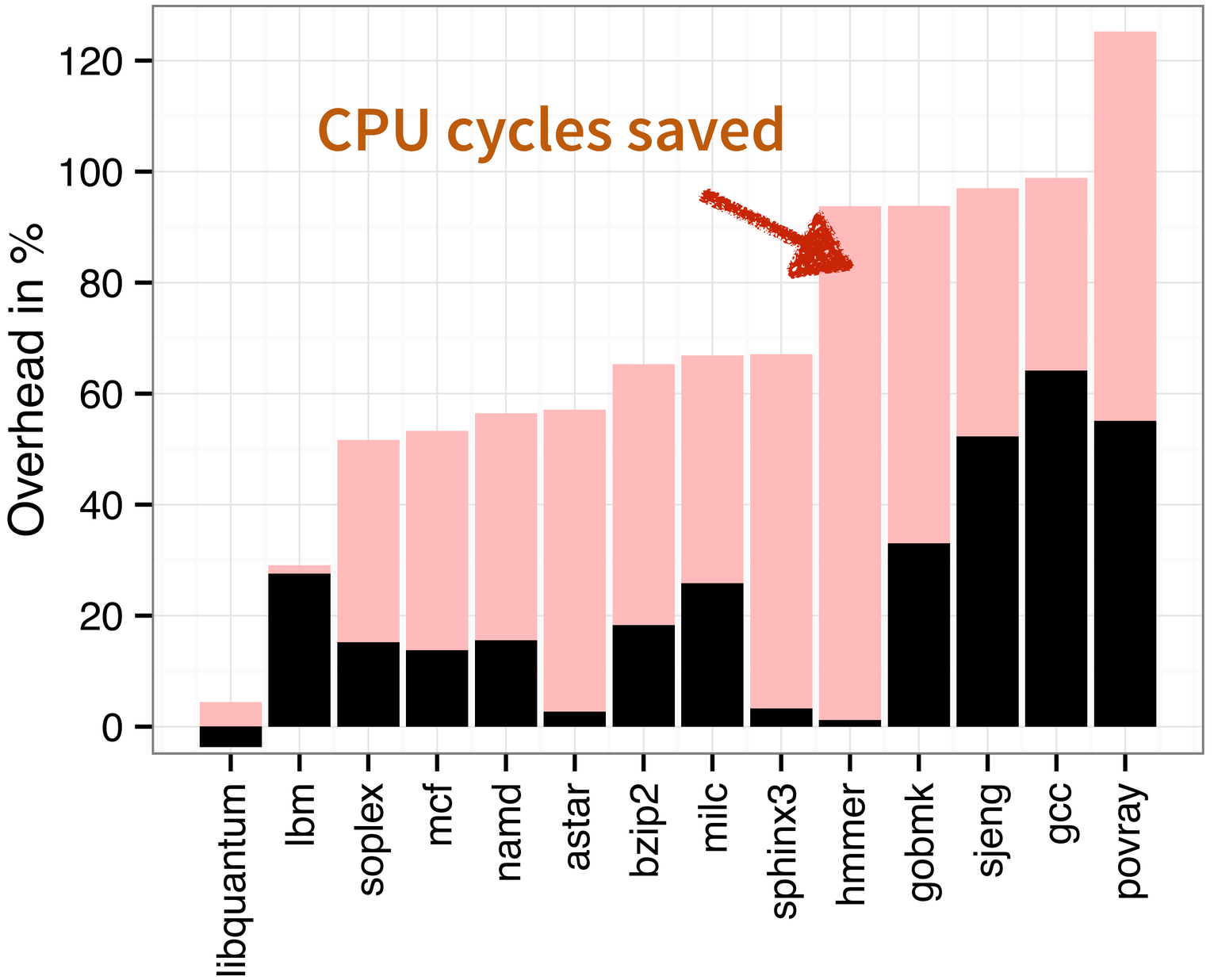


Results

Overhead for:
SPEC benchmarks
AddressSanitizer

95%

sanity level

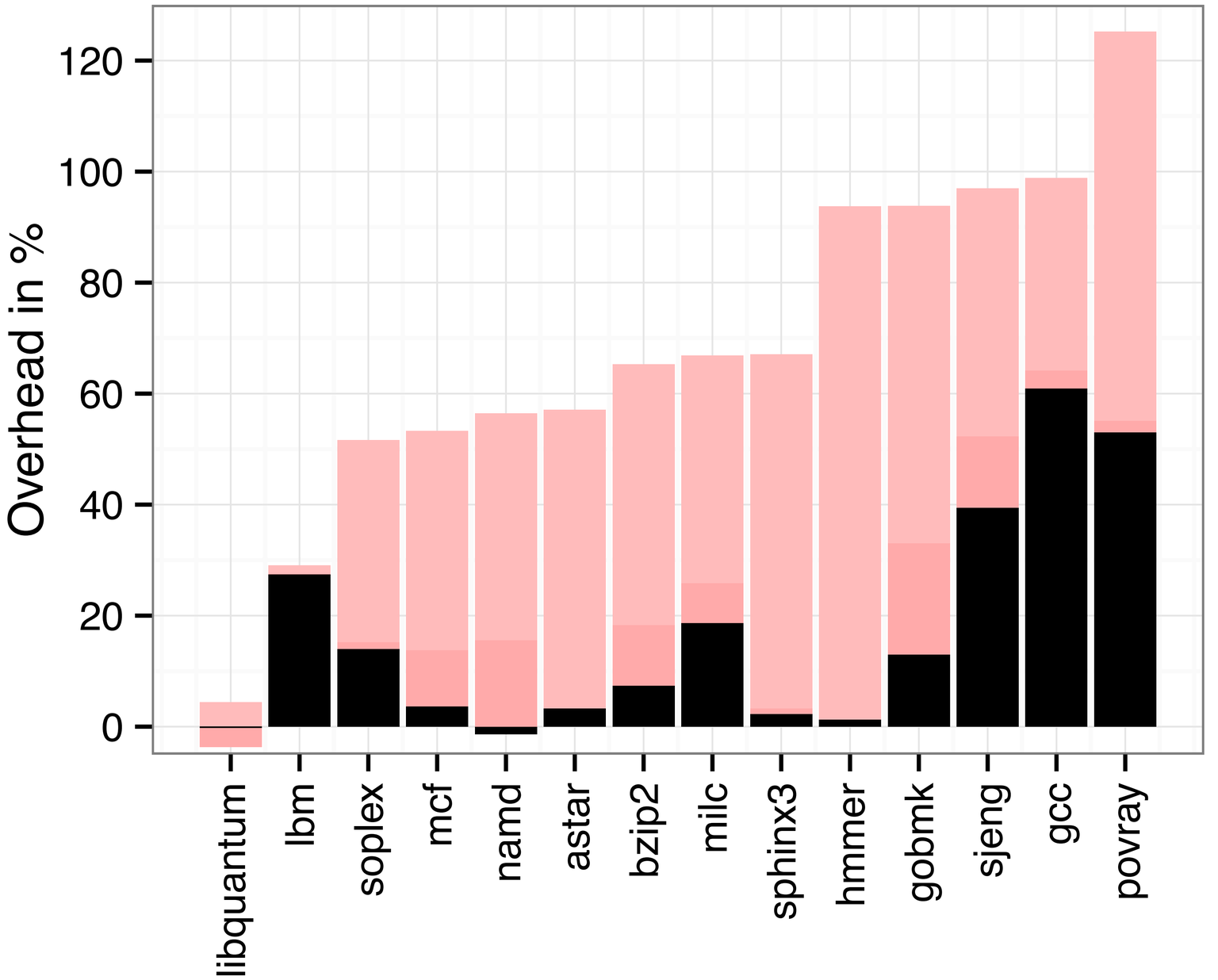


Results

Overhead for:
SPEC benchmarks
AddressSanitizer

90%

sanity level

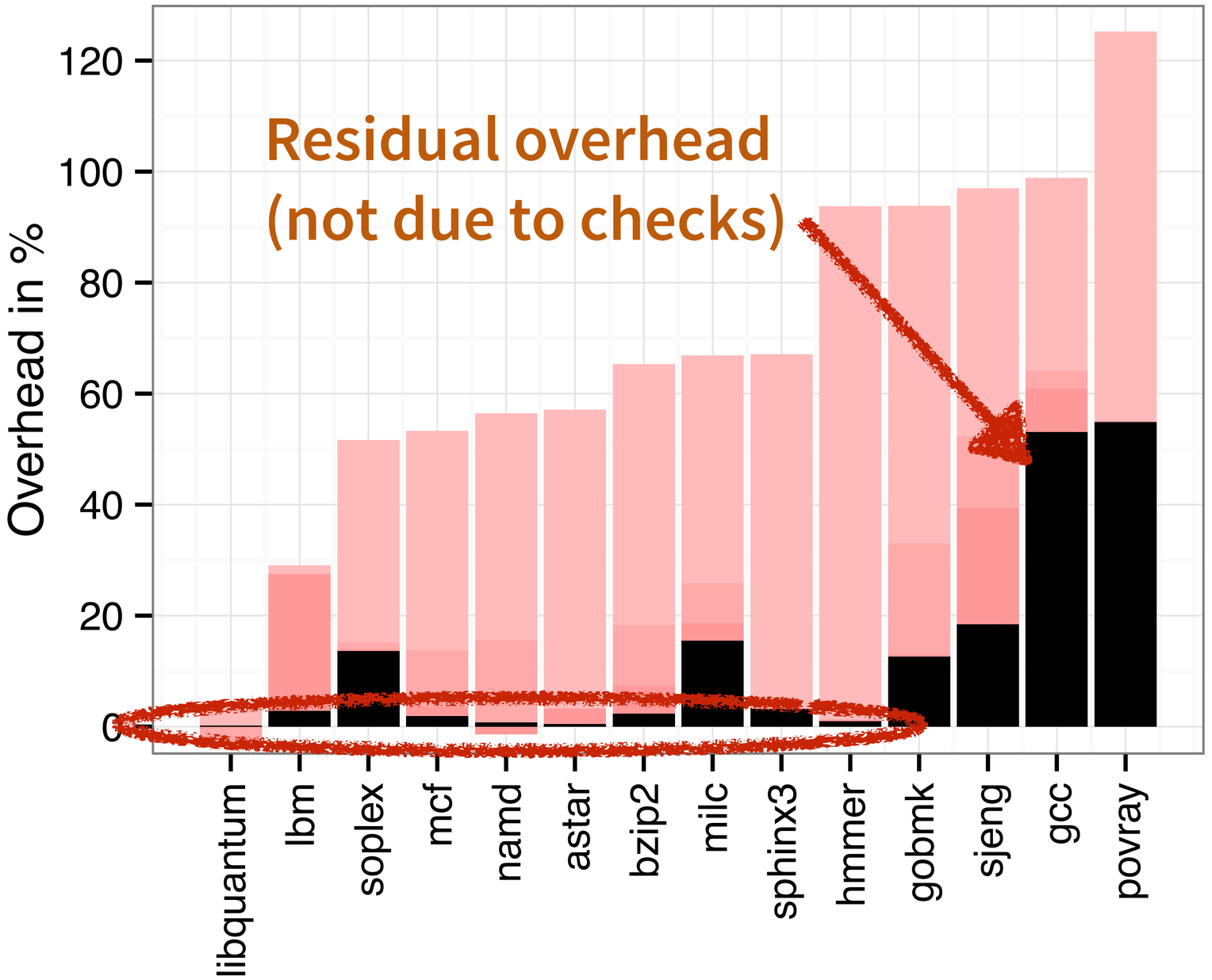


Results

Overhead for:
SPEC benchmarks
AddressSanitizer

80%

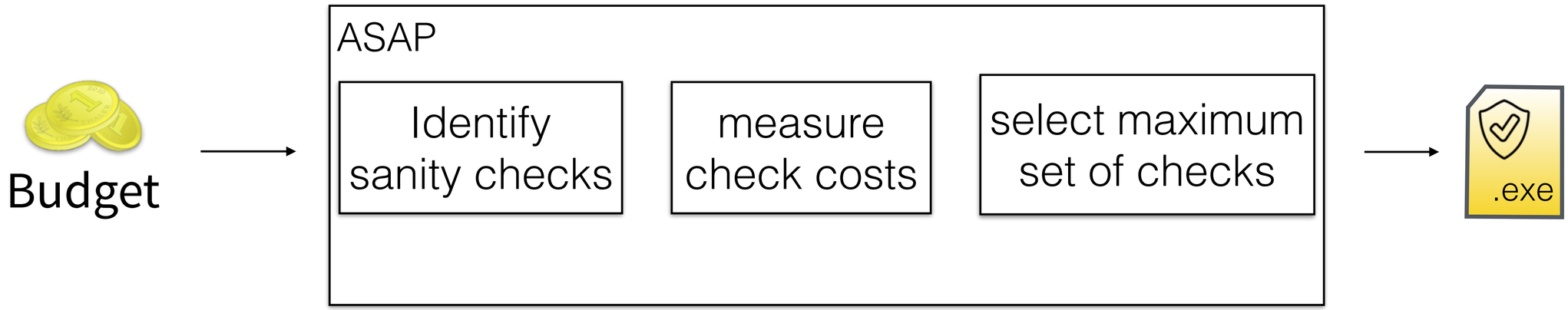
sanity level



Conclusion

Protect your software!
dslab.epfl.ch/proj/asap

Run-time checks deliver strong protection at ~~high cost~~



Most of the *overhead* comes from a few expensive checks

Most of the *protection* comes from many cheap checks

Backup slides

Sanity Check in LLVM