HEINZ NIXDORF INSTITUT
UNIVERSITÄT PADERBORN

Static Analysis for C++ with Phasar

Block 4

Philipp Schubert
`philipp.schubert@upb.de`

Ben Hermann
`ben.hermann@upb.de`

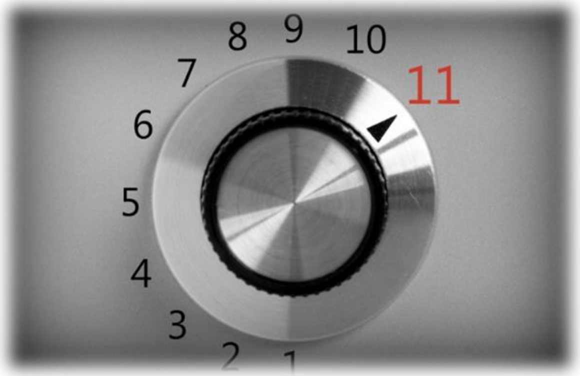
Eric Bodden
`eric.bodden@upb.de`

SOFTWARE TECHNIK

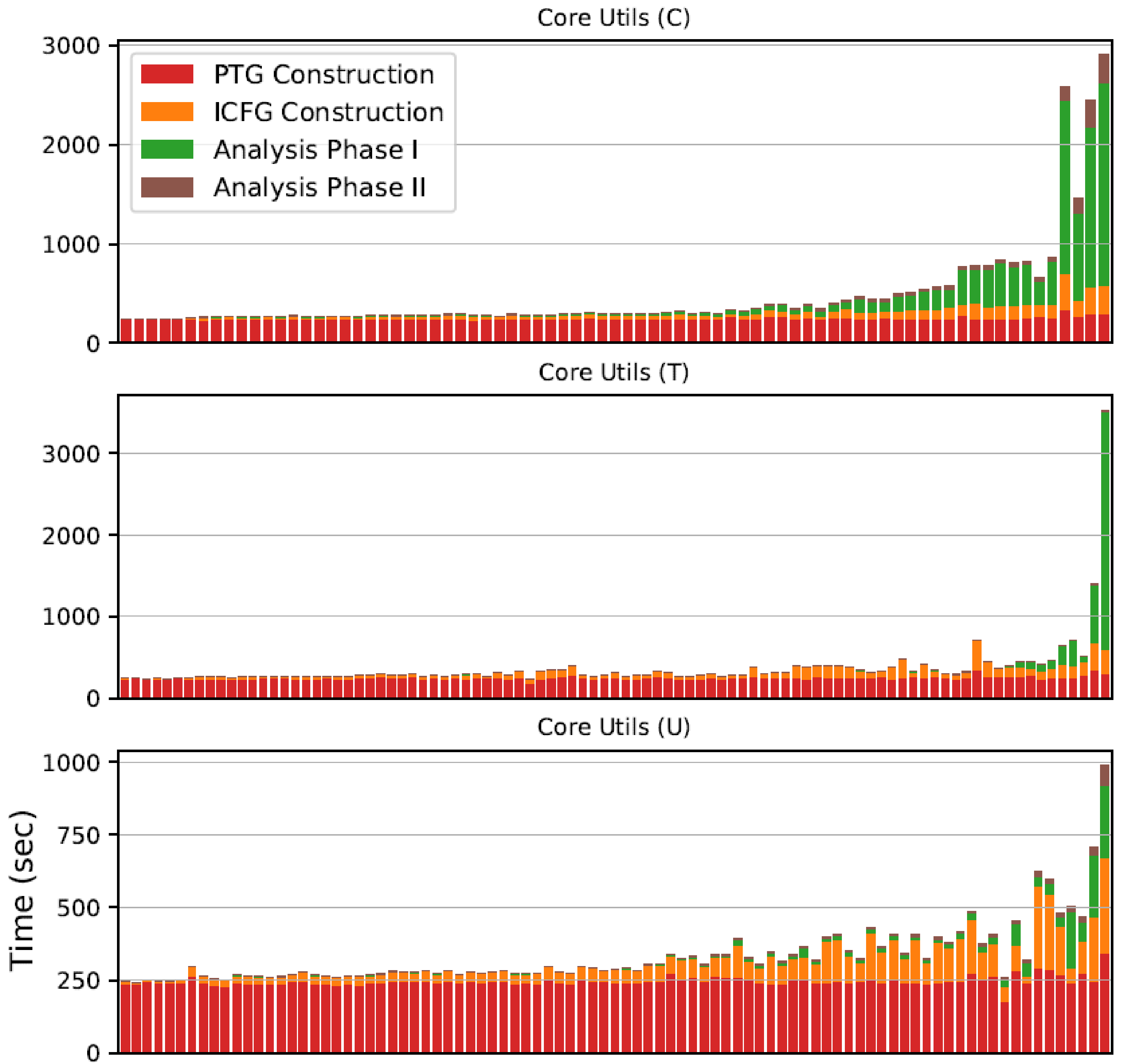
In this Block

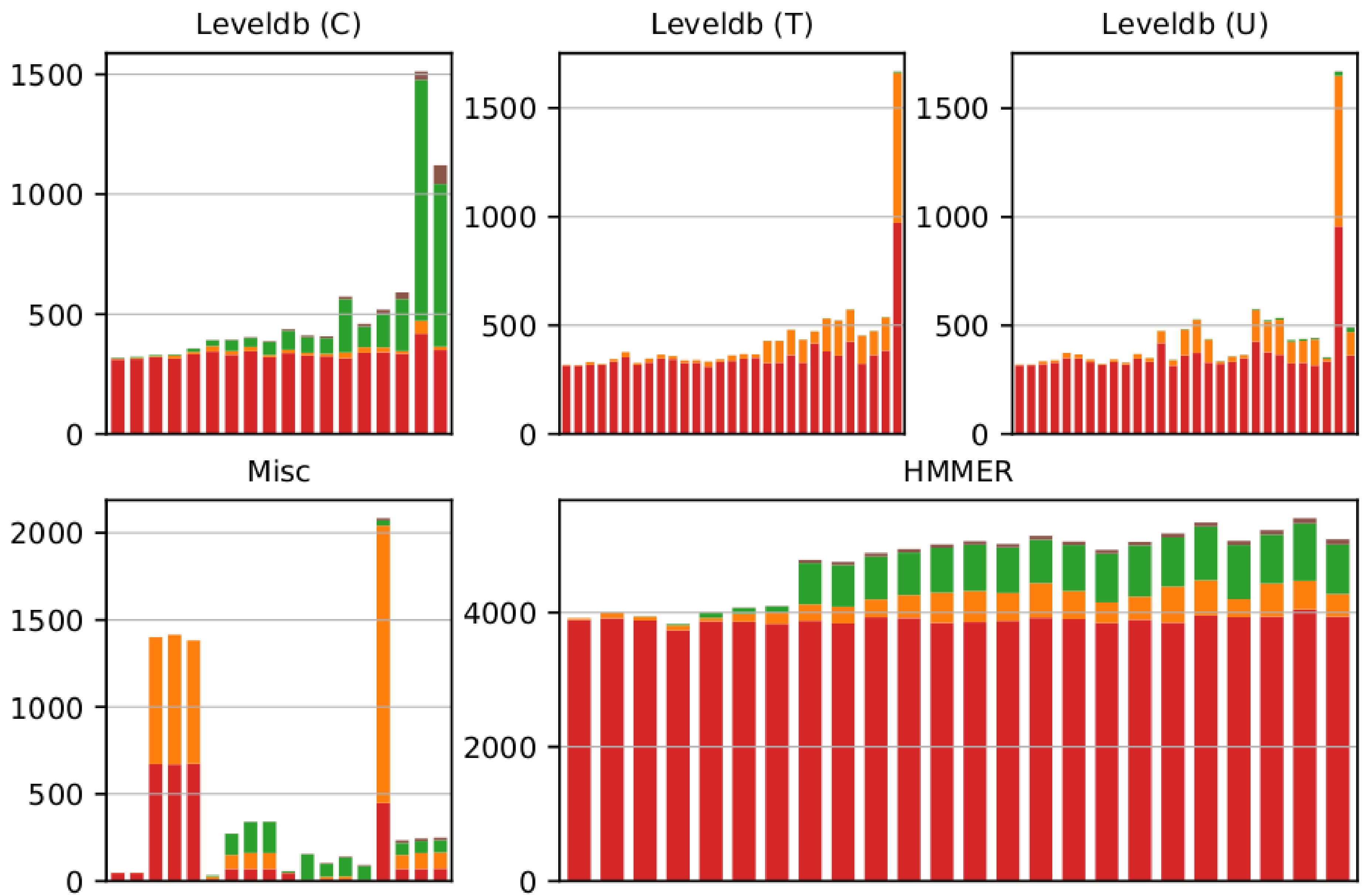
Measure your analysis

- Which portions of the runtime is spent where?
- Use PAMM (PerformAnce Measurement Mechanism) by specifying **`-DPHASAR_ENABLE_PAMM=ON`**
  - System is disabled by default
  - Defines functionality and a bunch of corresponding macros to measure different metrics
    - Timer, counter, histograms
  - Data is exported as **`json`**
  - Visualized using **`python`** and **`pandas`**
  - Allows for framework and analysis optimizations
  - Aids analysis understanding

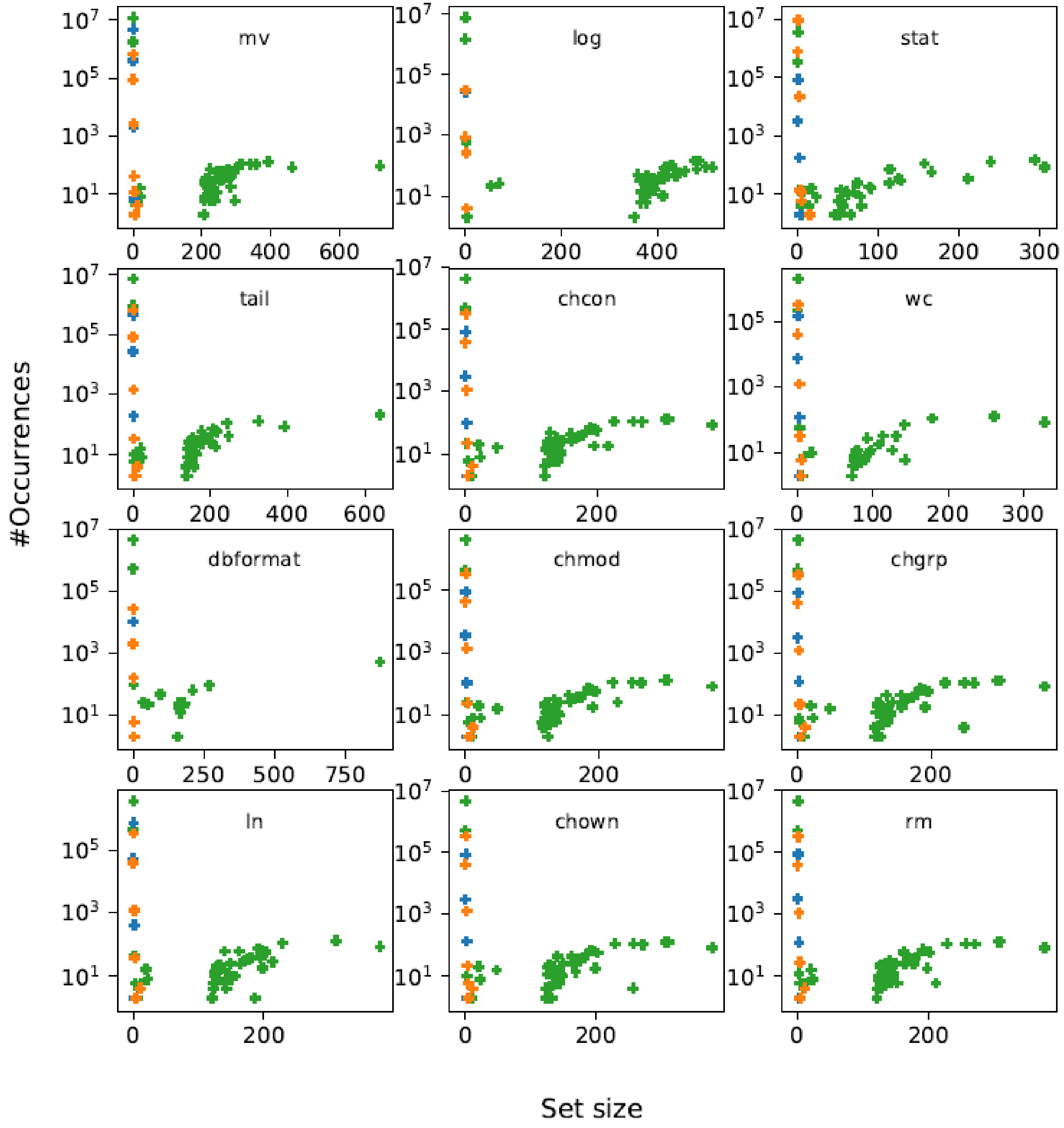


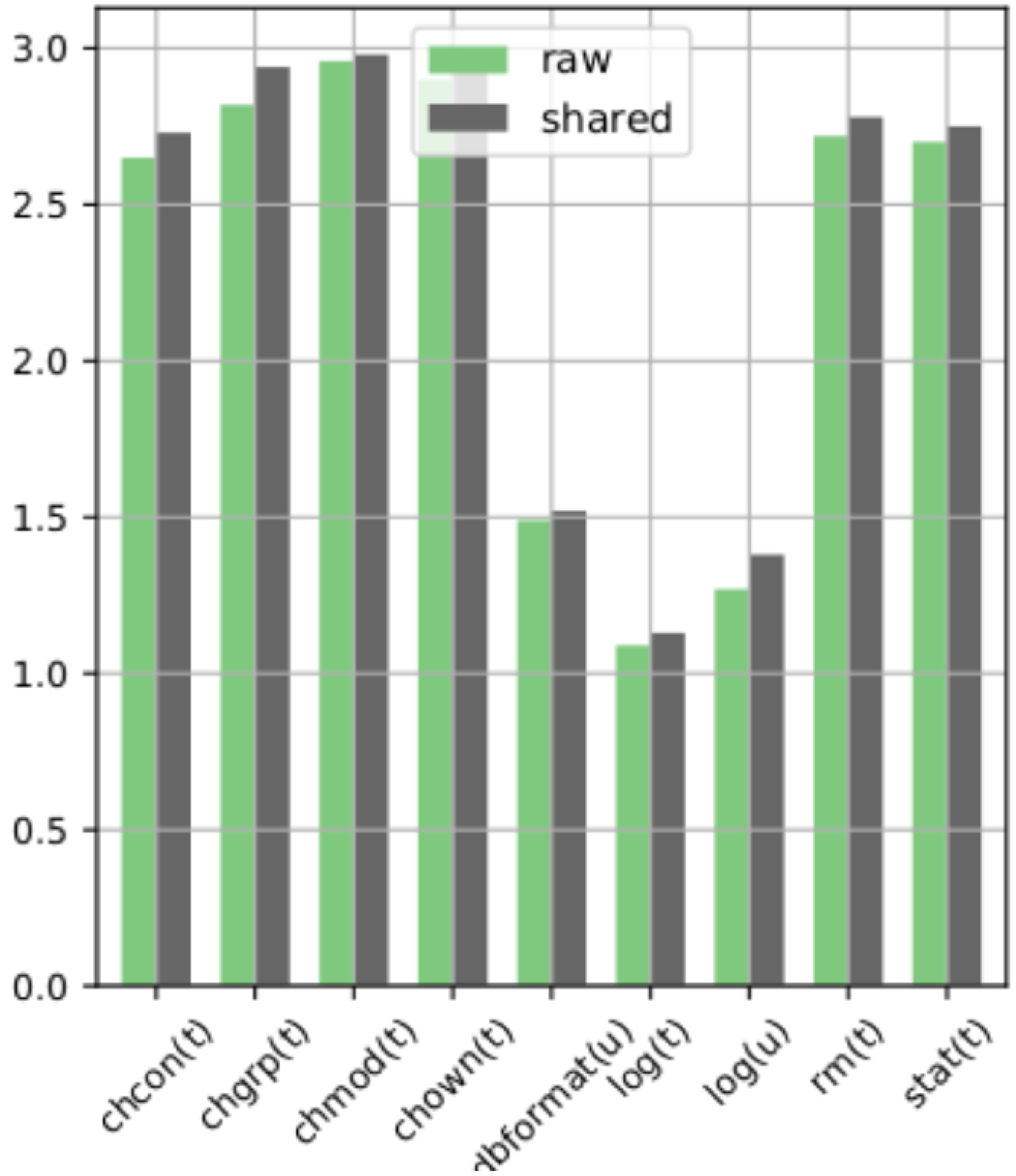
HEINZ NIXDORF INSTITUT
UNIVERSITÄT PADERBORN

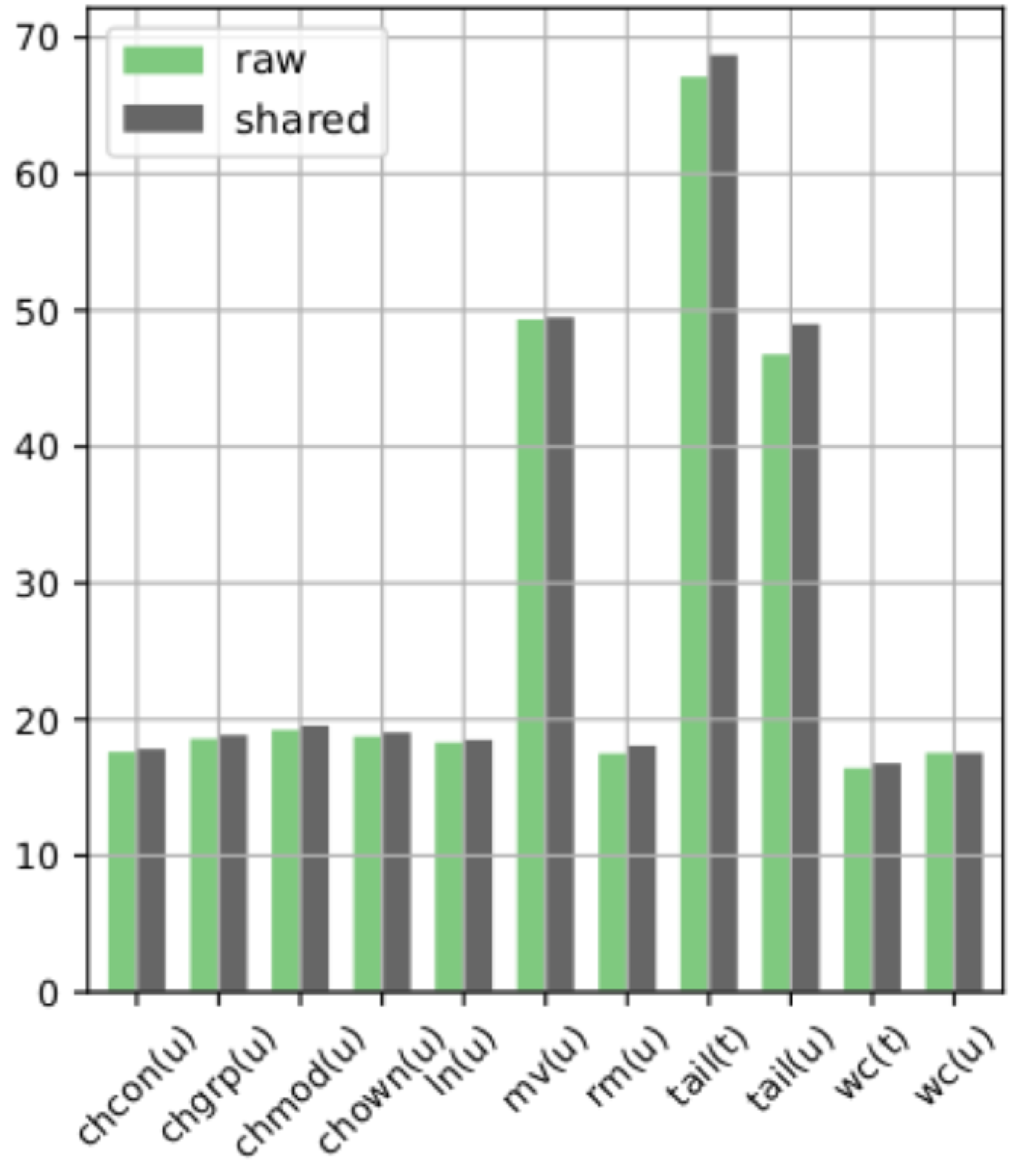

Core Utils (C)
PTG Construction
ICFG Construction
Analysis Phase I
Analysis Phase II
3000
2000
1000
0
Core Utils (T)
3000
2000
1000
0
Core Utils (U)
1000
750
500
250
0
Time (sec)

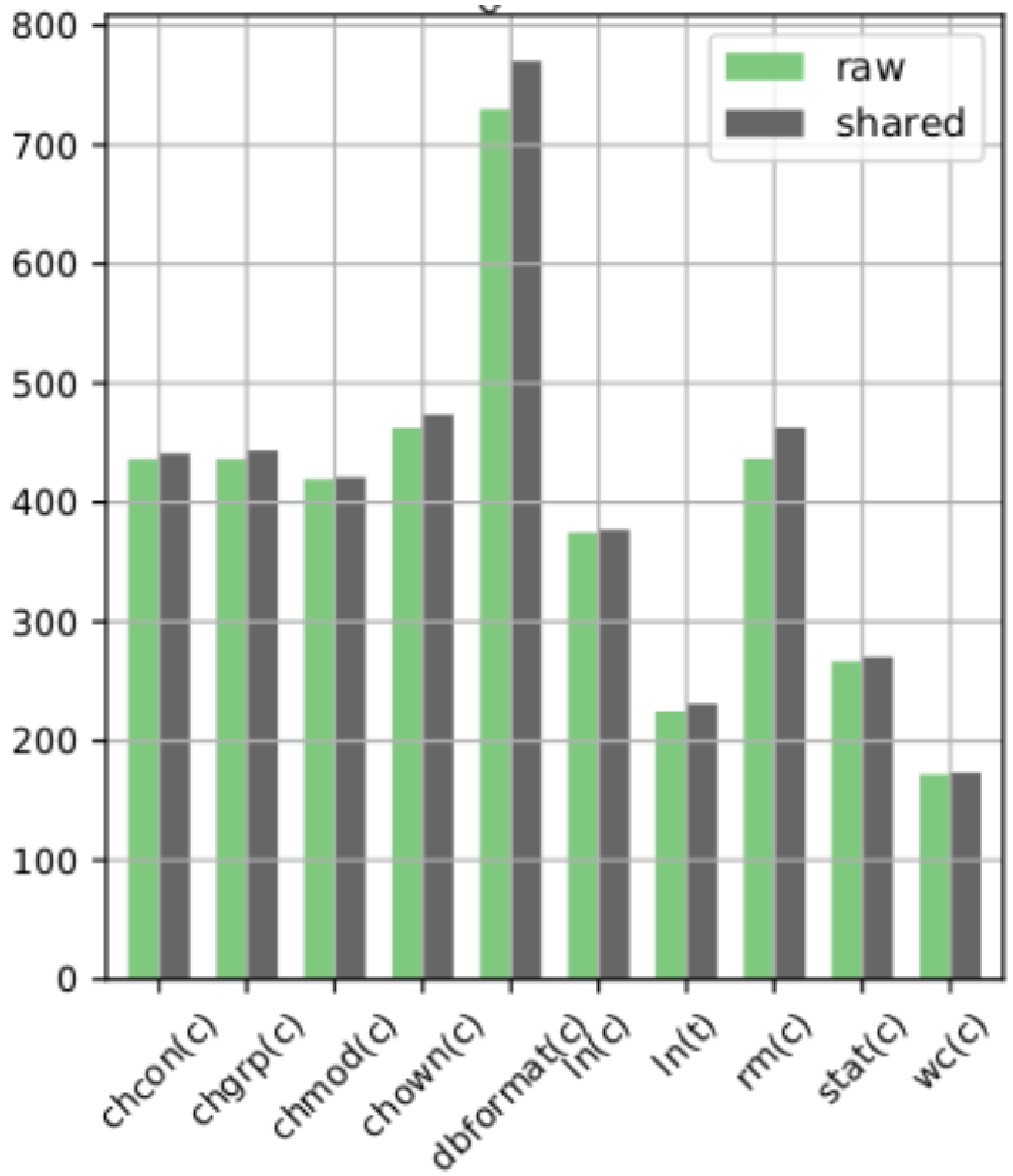


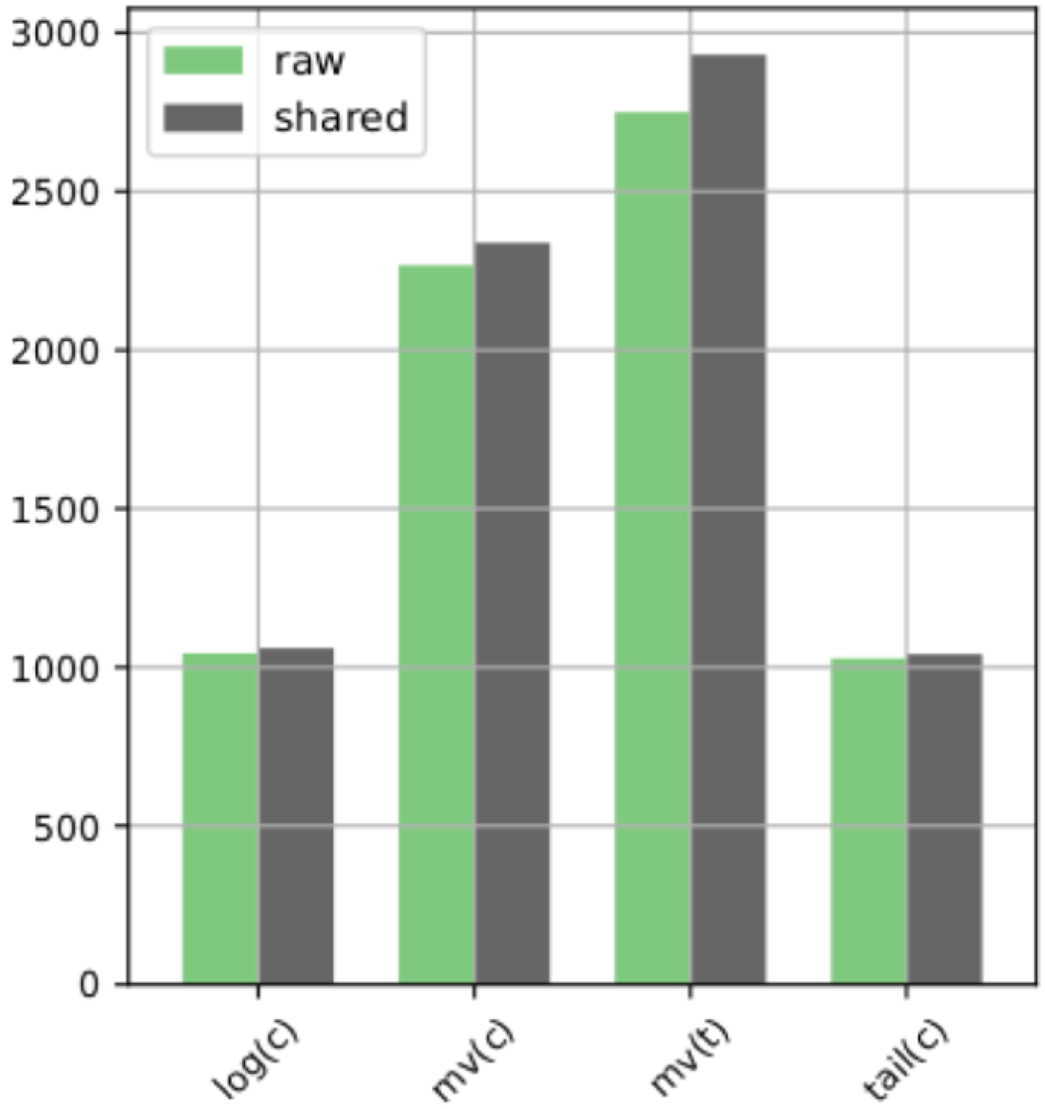
HEINZ NIXDORF INSTITUT
UNIVERSITÄT PADERBORN


mv
log
stat
tail
chcon
wc
dbformat
chmod
chgrp
ln
chown
rm
#Occurrences
Set size

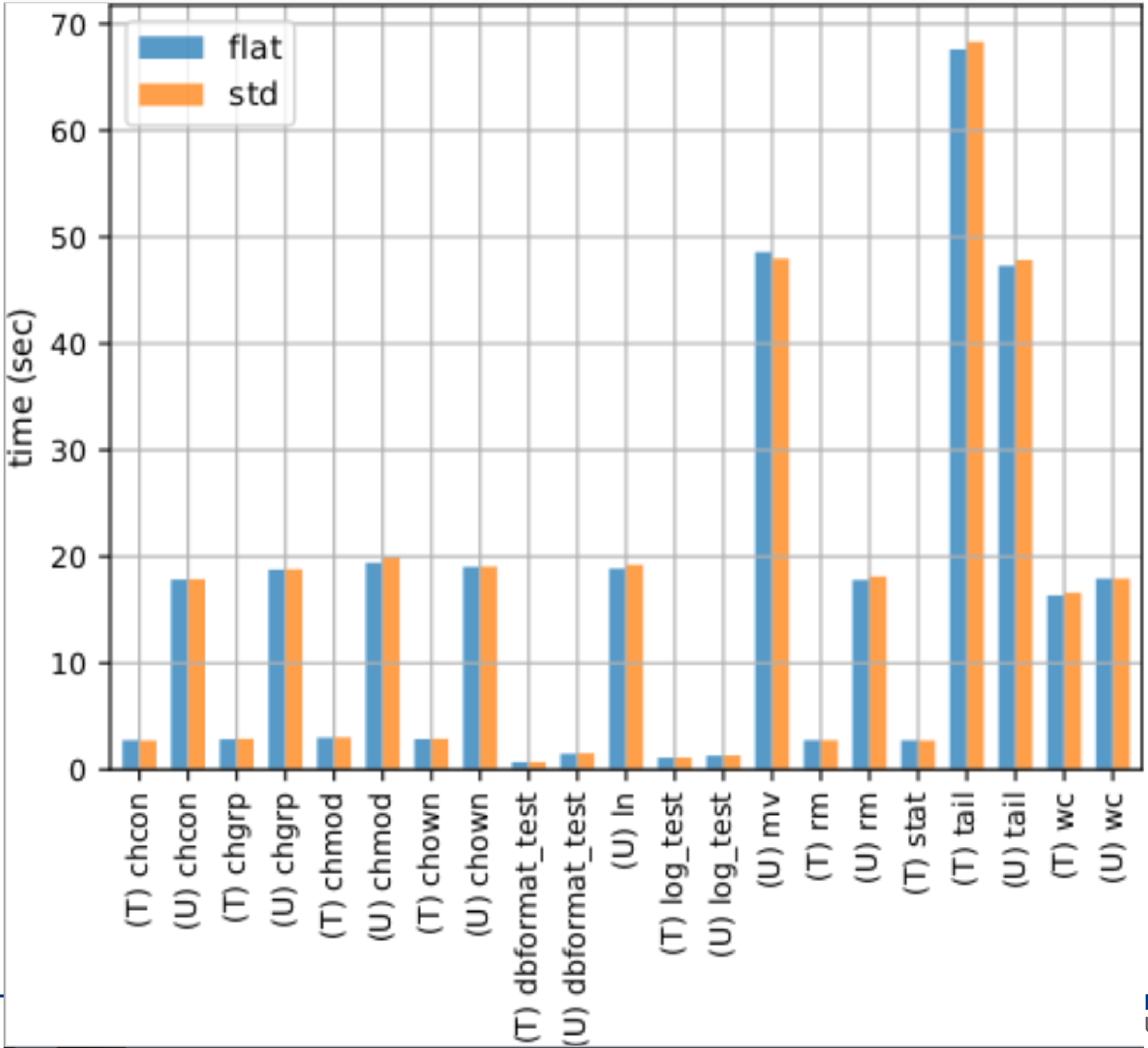

raw
shared
3.0
2.5
2.0
1.5
1.0
0.5
0.0
chcon(t)
chgrp(t)
chmod(t)
chown(t)
dbformat(u)
log(t)
log(u)
rm(t)
stat(t)

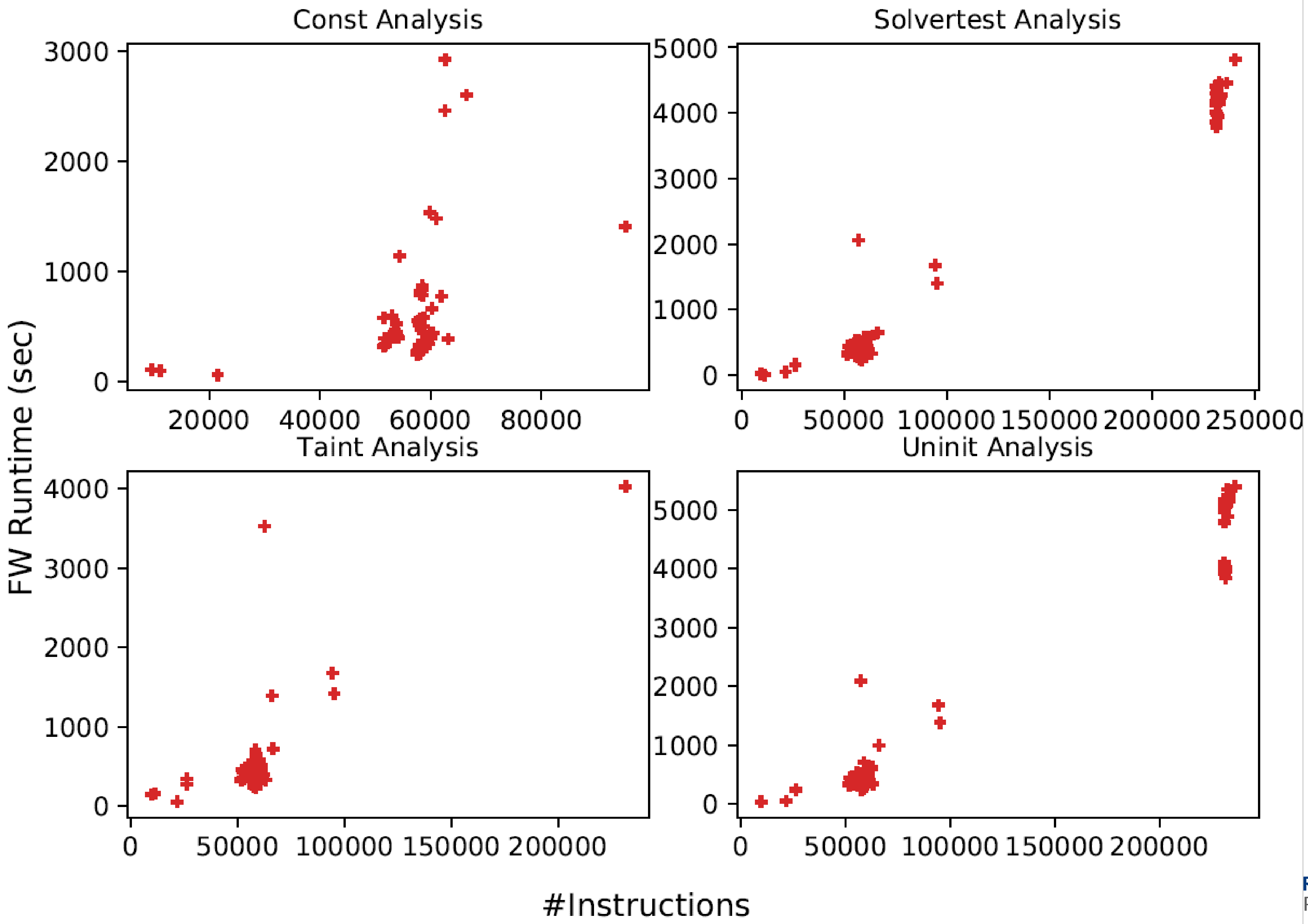

raw
shared
70
60
50
40
30
20
10
0
chcon(u)
chgrp(u)
chmod(u)
chown(u)
ln(u)
mv(u)
rm(u)
tail(t)
tail(u)
wc(t)
wc(u)





HEINZ NIXDORF INSTITUT
UNIVERSITÄT PADERBORN


flat
std
time (sec)
0
10
20
30
40
50
60
70
(T) chcon
(U) chcon
(T) chgrp
(U) chgrp
(T) chmod
(U) chmod
(T) chown
(U) chown
(T) dbformat_test
(U) dbformat_test
(U) ln
(T) log_test
(U) log_test
(U) mv
(T) rm
(U) rm
(T) stat
(T) tail
(U) tail
(T) wc
(U) wc



RF INSTITUT
PADERBORN

Analysis on real-world code

- Input language matters!
  - C or C++, something else?
  - LLVM provides all facilities to capture arbitrary source languages
    - C, C++, Objective C, Rust, Swift, …
  - It is all LLVM IR but …
    - language characteristics and complexity propagate into IR
    - E.g. indirect call-site: `%5 = call i32 %4(%struct.S* dereferenceable(4) %2, i32 5)`
      - In C: it's a function pointer, worst case → signature matching
      - In C++: oh, right?
        - Is it a function pointer or a virtual member function?
        - The odyssee begins → analysis time increases
        - More and more corner cases must be considered

Analysis on real-world code

- Remember C++´s special member functions
  - Boils down to IR, but must be considered
  - Keep semantics of source language in mind
- Easy to start, hard to finish
  - Target test code works
  - Production code segfaults
    - Find and handle bizarre corner cases
- Hard to debug
  - Size and amount of information
  - Visualization?
    - We are currently integrating one

ODR violation?

```
M1:
static void foo() {}  →   @foo
                      →   @_ZL3foov
M2:
static void foo() {}  →   same as above

M1 + M2 →   @_ZL3foo, @_ZL3foo.1
```

Analysis on real-world code

- What are your thoughts, results and observations?

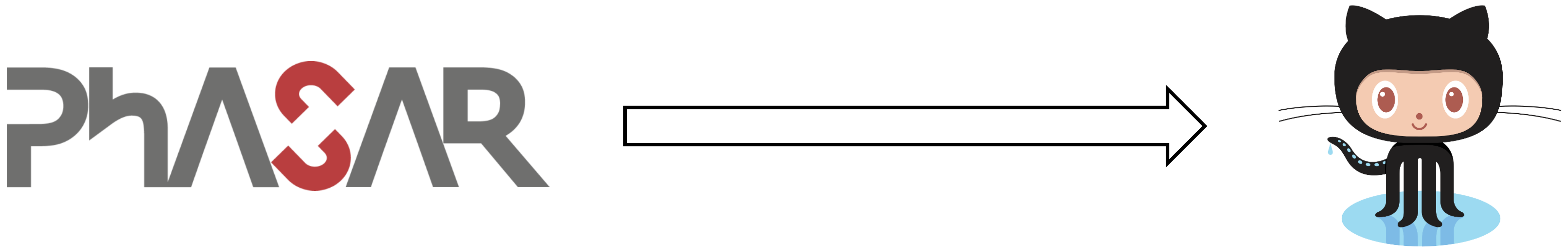
User demands

“Roads? Where we're going, we don't need roads.”

- What features might be useful in the future?
- Some features integrated soon:
  - Map results from IR back to source level
  - Use SVF framework for more precise pointer analysis
  - Offer code generator for analysis templates

Help us to develop Phasar on Github

- Give it a try
  - https://github.com/secure-software-engineering/phasar
  - https://phasar.org/
- Analyze some programs
- Write your own useful analyses, we provide the tools
- Create issues to track bugs, request features and more
- Create pull requests



Questions

Thank you for your attention

Questions?