Raccoon: Closing Digital Side-Channels through Obfuscated Execution

Ashay Rane, Calvin Lin, Mohit Tiwari

The University of Texas at Austin

Secure code?

```
if (secret_bit == 1) {
    z = (msg * z * z) mod n;
} else {
    z = (z * z) mod n;
}
```

Instruction Pointer

Instruction count

Address trace

Cache usage

Branch predictor

EM radiation

Power

In fact, very insecure!

Serious Attacks

Side-channel attacks have been shown to:

- Leak private encryption keys.
- Reveal private medical information.
- Assist in reverse-engineering closed-source processors.

An Example Solution

```
if (secret_bit == 1) {
    z = (msg * z * z) mod n;
} else {
    z = (z * z) mod n;
    NOP;
    NOP;
}
```

Solution steps:

1. Pad else-block with NOPs to normalize time.

An Example Solution

```
if (secret_bit == 1) {
    z = (msg * z * z) mod n;
} else {
    read msg_dummy;
    z = (z * z) mod n;
    NOP;
    NOP;
}
```

Solution steps:

1. Pad else-block with NOPs to normalize time.
2. Introduce dummy variables. Use Oblivious RAM.

An Example Solution

```
if (secret_bit == 1) {
    z = (msg * z * z) mod n;
} else {
    read msg_dummy;
    z = (z * z) mod n;
    NOP;
    NOP;
}
```

Problems with this solution:

1. Close limited number of side-channels.
2. Forced to use ORAM for scalar variables.
3. Has to disable compiler and hardware optimizations (branch prediction, hardware caching).

Many Point Solutions

Each solution closes only a **narrow set of side-channels**.

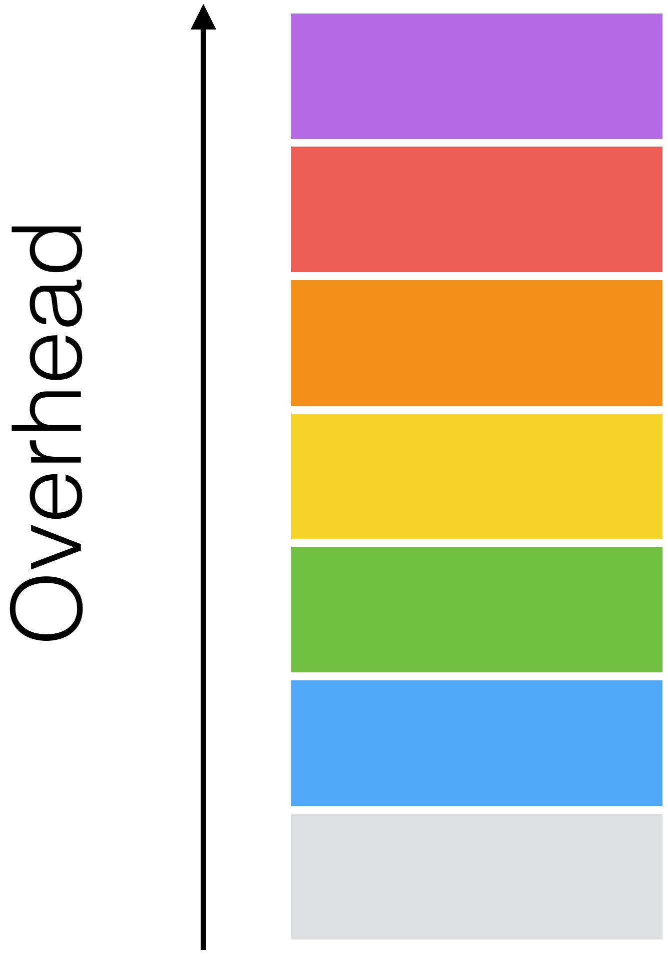
- Defenses against cache side-channel:
  [HPCA-09], [USENIX-12a], [USENIX-12b], [ISCA-07], [S&P-09], [MICRO-08], [S&P-11], [CCS-12].

- Defenses against address-trace side-channel:
  [ASPLOS-04], [CCS-13a], [STC-12], [ICS-03], [HASP-13], [HPCA-14], [ASPLOS-15], [CCS-13b].

- Defenses against power side-channel:
  [CHES-01a], [ACISP-01], [CHES-00], [DATE-05], [PKC-02], [ACISP-04], [CHES-01b], [CT-RSA-03].

Drawbacks of Point Solutions

- **High overhead** from composing multiple point solutions together.



Drawbacks of Point Solutions

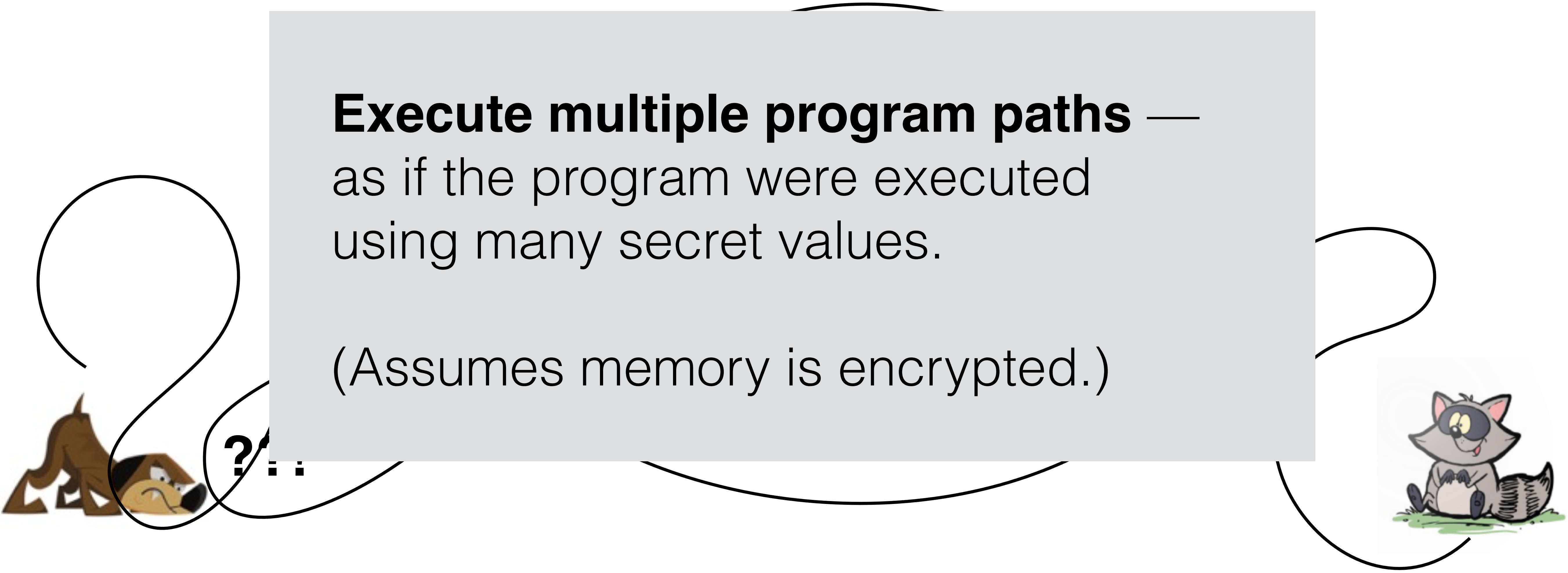
- **High overhead** from composing multiple point solutions together.
- Individual point solutions may **negate each other's defenses**.

Example: Tradeoff between instruction count and execution time.

Changing instruction count also changes execution time.

Our Solution

Do what Raccoons do!



Execute multiple program paths — as if the program were executed using many secret values.

(Assumes memory is encrypted.)

Our Solution: **Raccoon**

Execute multiple program paths —
as if the program were executed
using many secret values.

```
if (secret_bit == 1) {
    z = (msg * z * z) mod n;
} else {
    z = (z * z) mod n;
}
```

Our Solution: **Raccoon**

Execute multiple program paths — as if the program were executed using many secret values.

```
if (secret_bit == 1) {
    z = (msg * z * z) mod n;
} else {
    z = (z * z) mod n;
}
```

Our Solution: **Raccoon**

Execute multiple program paths —
as if the program were executed
using many secret values.

```
if (secret_bit == 1) {
    z = (msg * z * z) mod n;
} else {
    z = (z * z) mod n;
}
```

Adversary sees `secret_bit == 1`.

Our Solution: **Raccoon**

Execute multiple program paths —
as if the program were executed
using many secret values.

```
if (secret_bit == 1) {
    z = (msg * z * z) mod n;
} else {
    z = (z * z) mod n;
}
```

Adversary sees `secret_bit == 1` **and** `secret_bit == 0`.

Our Solution: **Raccoon**

Execute multiple program paths —
as if the program were executed
using many secret values.

```
if (secret_bit == 1) {
    z = (msg * z * z) mod n;
} else {
    z = (z * z) mod n;
}
```

Adversary sees `secret_bit == 1` **and** `secret_bit == 0`.

Key Insight

1. Many side-channels emerge from variations in source-level behavior.

   — Branch predictor state affected by program path.
   — Memory access trace affected by data-flow and control-flow.
   — Instruction count governed by program path.

2. Program behavior is captured by control flows and data flows.

3. Hence, normalizing / randomizing control flows and data flows closes many side-channels.

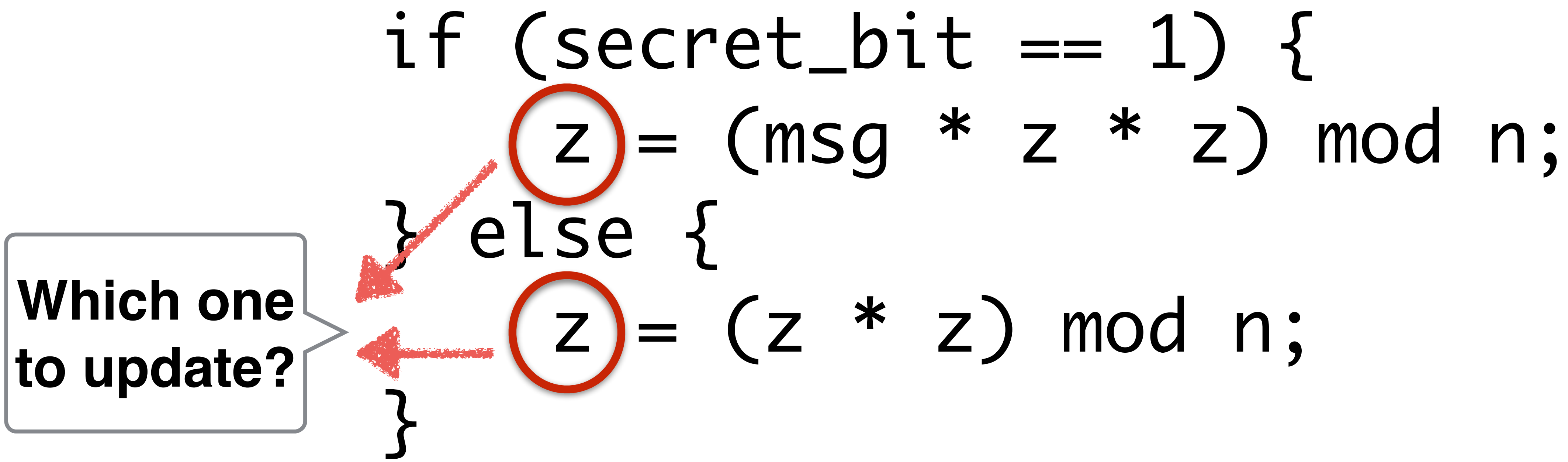
Raccoon's Approach

- Raccoon's static analysis limits explored branches, thus prevents path explosion.
- Raccoon does not *simulate* paths,
  Raccoon *executes* actual instructions.

```
if (secret_bit == 1) {
    z = (msg * z * z) mod n;
} else {
    z = (z * z) mod n;
}
```

Raccoon’s Approach

- Raccoon’s static analysis limits explored branches, thus prevents path explosion.
- Raccoon does not *simulate* paths, Raccoon *executes* actual instructions.



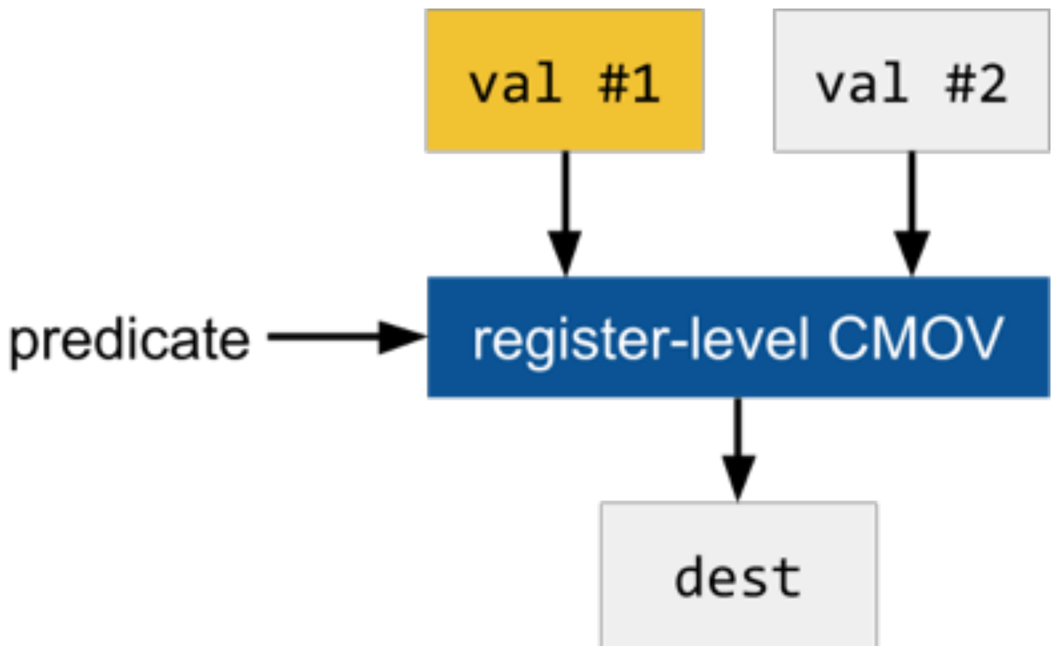
```
if (secret_bit == 1) {
    z = (msg * z * z) mod n;
} else {
    z = (z * z) mod n;
}
```

Which one to update?

Transaction-Like Buffers

- Raccoon buffers `write` operations from each path into a separate transaction buffer.

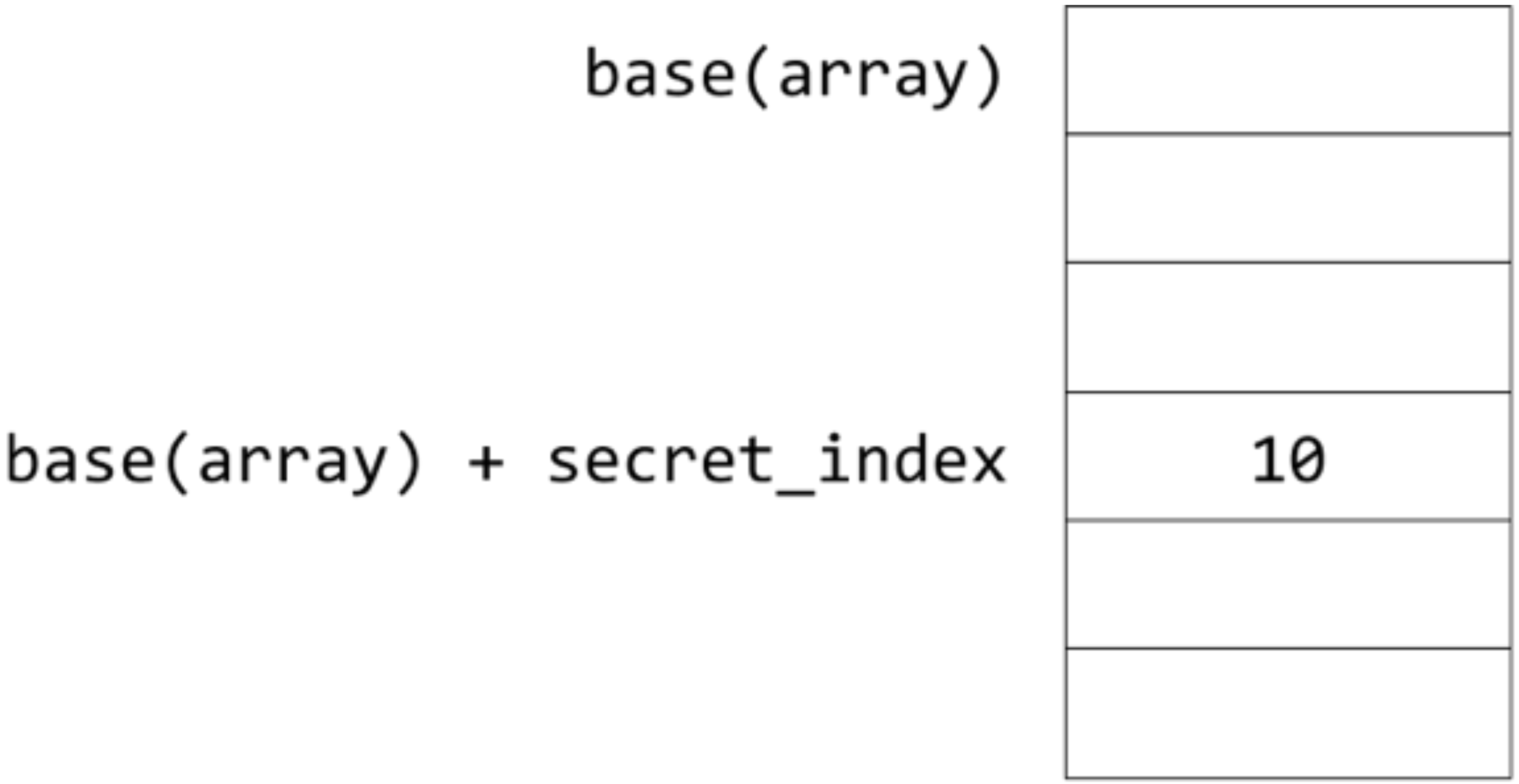
- After both paths finish execution, one buffer is **secretly** saved to memory, other is discarded.



Other Defenses in Raccoon

Simply following multiple program paths is insufficient!

```
array[secret_index] = 10;
```



Memory address reveals value of `secret_index`.

Array Access Obfuscation

- **Option 1:** Normalize address trace.
  Access entire array for every load/store operation.

- **Option 2:** Randomize address trace.
  Obfuscate addresses using ORAM.

  Raccoon includes software version of Path ORAM.

Benefits of Raccoon

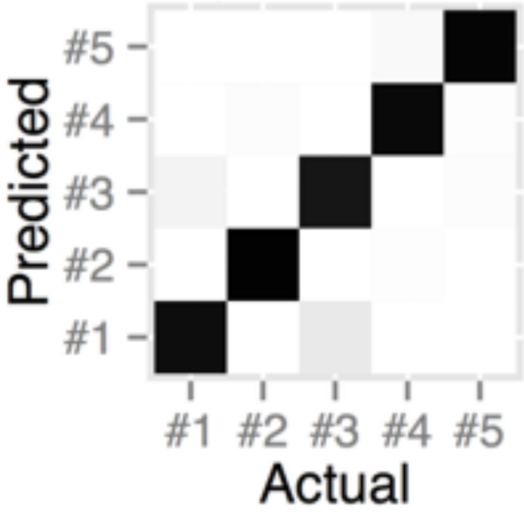
- Expands the threat model:
  Defends against broader class of side-channels.
  Code is no longer secret, only data. Hence no need to hide code.
- Does not require special-purpose hardware:
  Hence backward compatible with older processors.
- Expands the allowed set of language features,
  when compared with prior work.

Security Evaluation

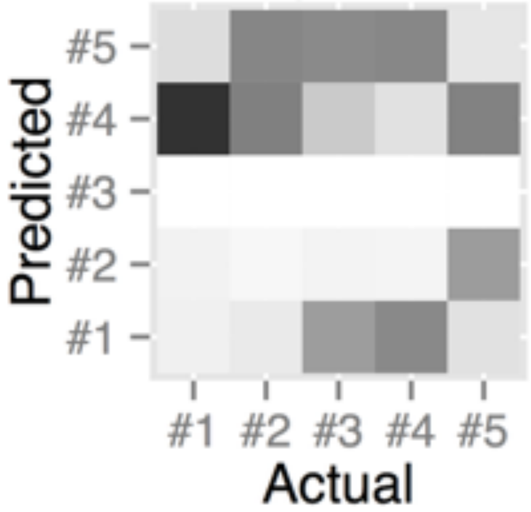
- Security of obfuscated code.
  Verified that control flows and data flows are correct and
  do not leak information.

- Security of obfuscation code.
  Verified using inference rules over LLVM IR that obfuscation code
  does not leak secrets.

- Demonstration of side-channel defense.
  — Adversary process snoops instruction pointer using `/proc`.
  — Adversary runs k-fold cross-validation, plots confusion matrix.



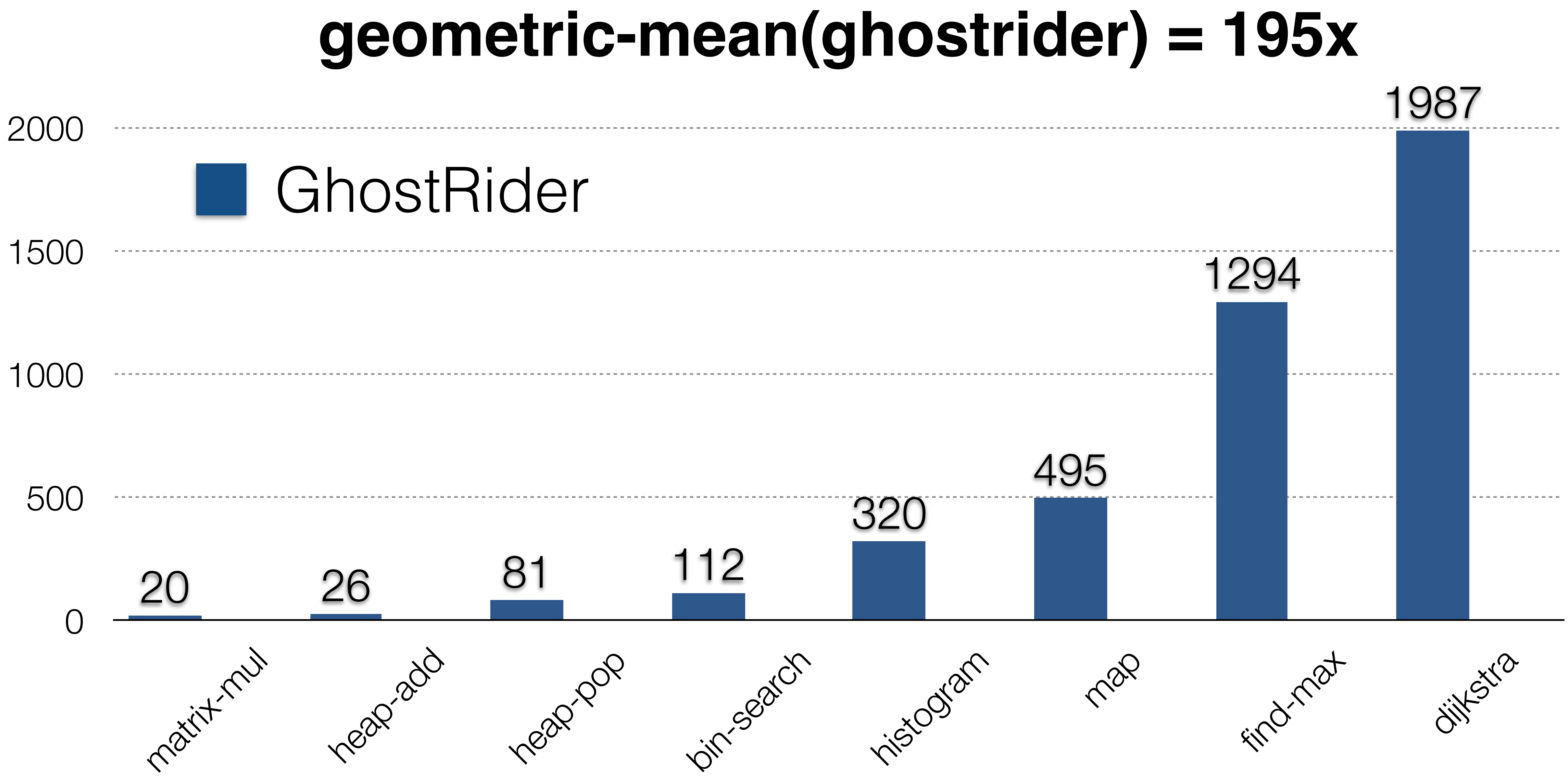
After obfuscation →



Performance Evaluation

- We evaluate performance overhead on a modern (Intel Xeon) processor.
- To place our work in the context of well-renowned related work,
  we compare Raccoon with a competing solution called GhostRider.
- We use 15 programs (which includes 8 from GhostRider).

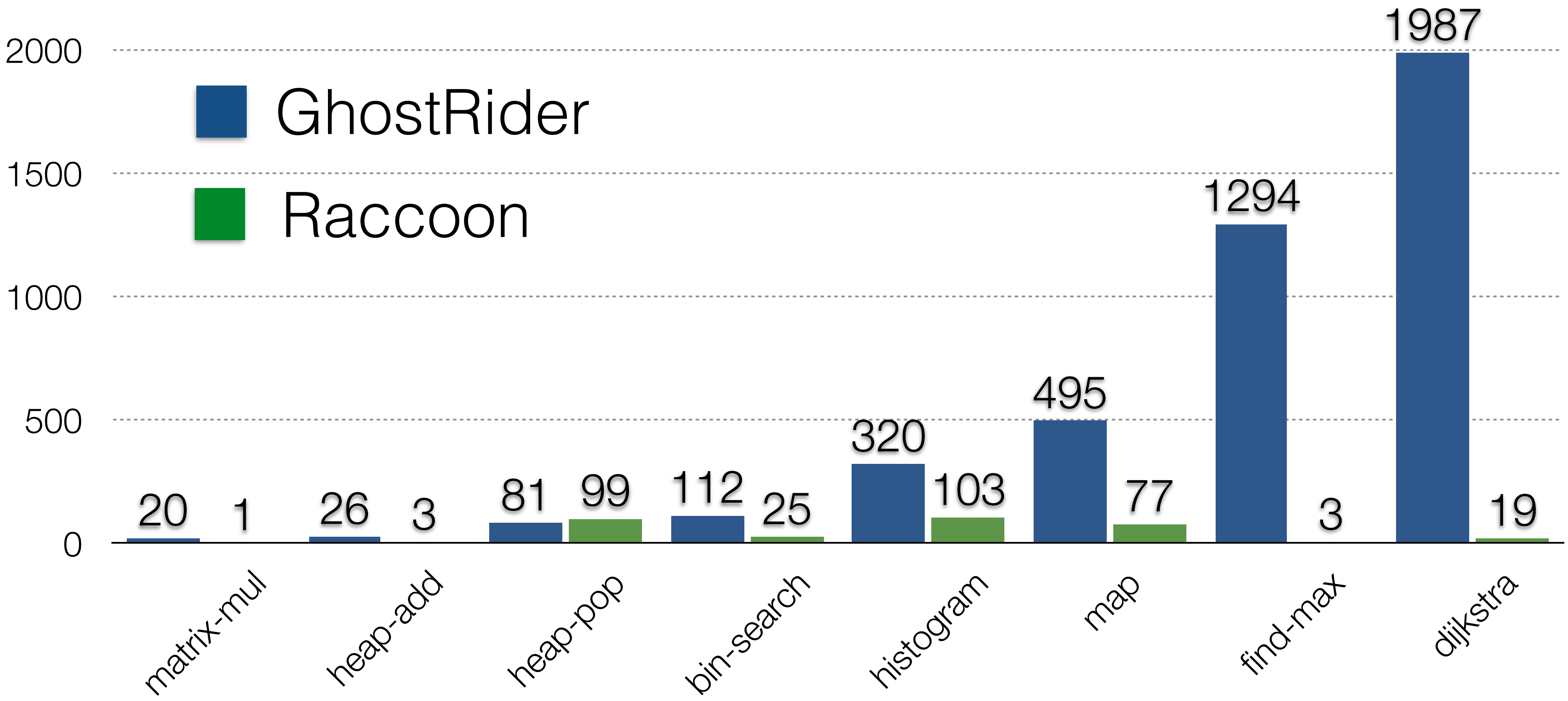
GhostRider (ASPLOS ’15)



GhostRider v/s Racccon

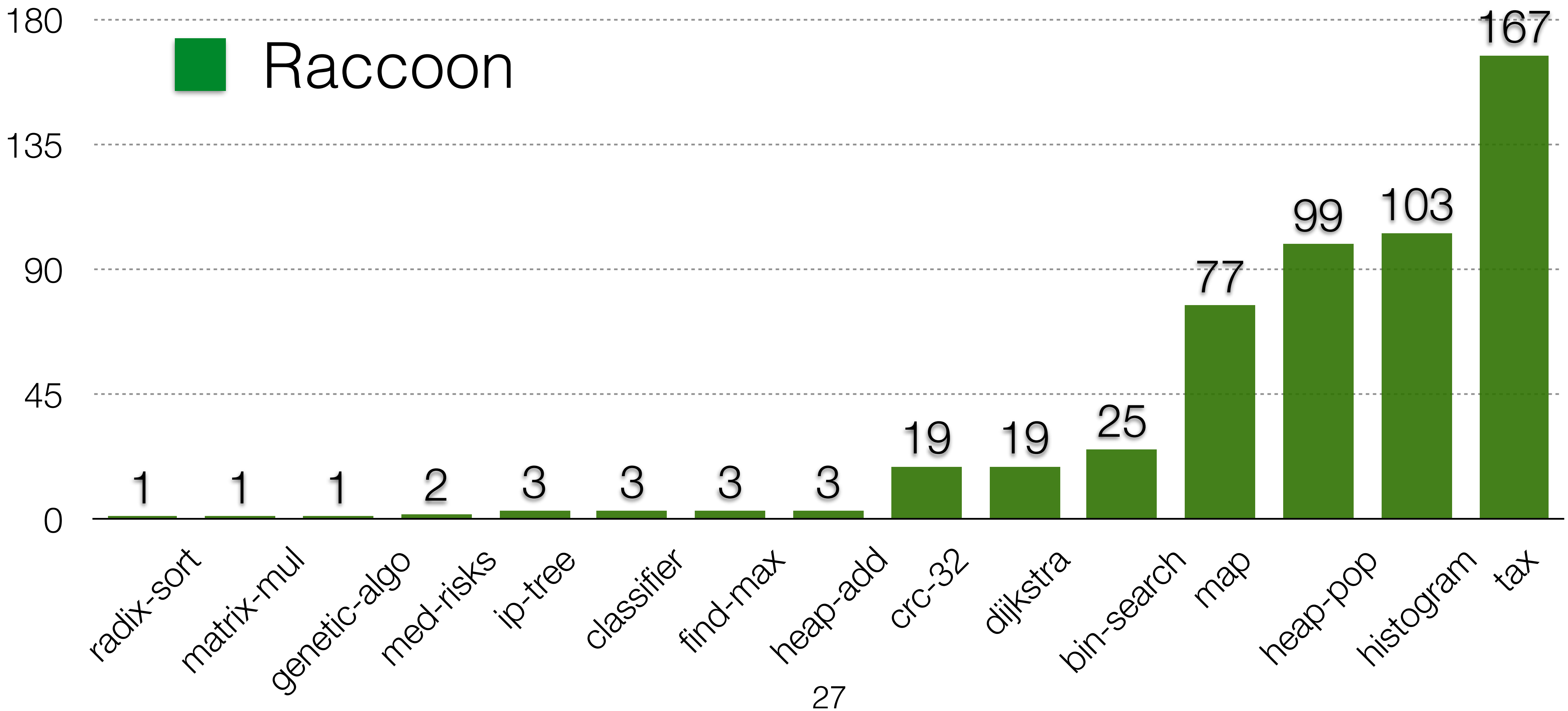
geometric-mean(raccoon) = 16x

geometric-mean(ghostrider) = 195x



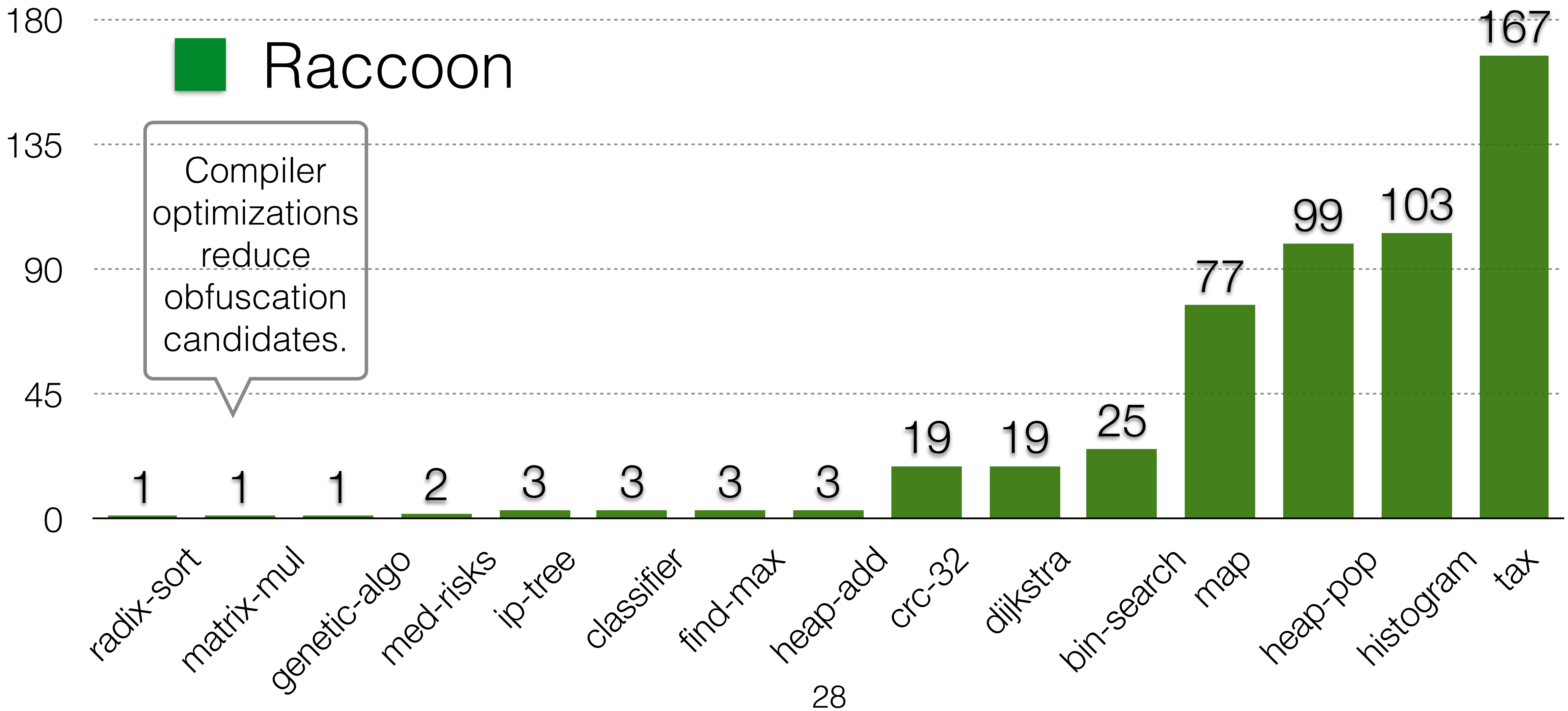
Raccoon’s Overhead

geometric-mean(raccoon) = 9x



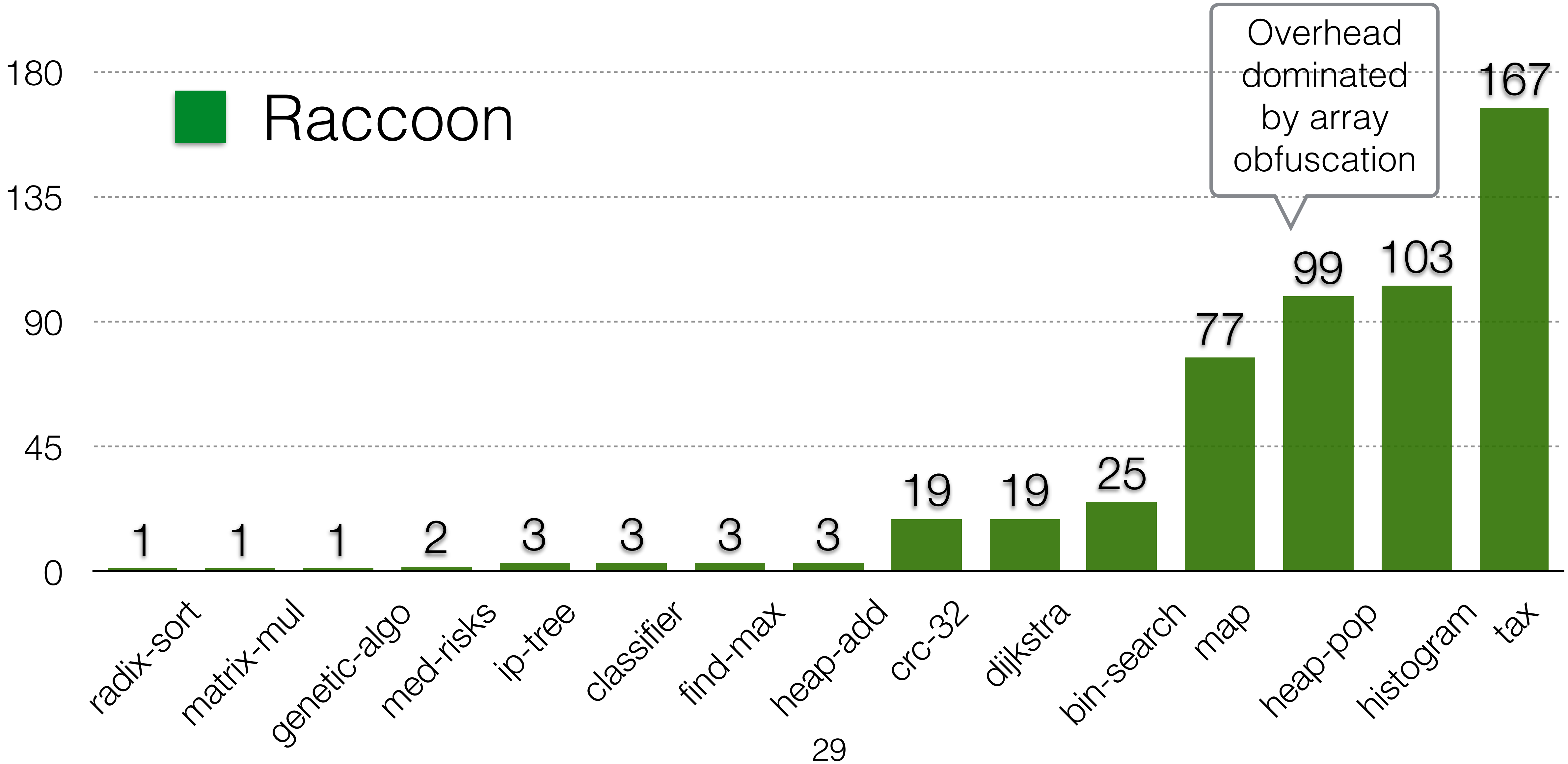
Raccoon’s Overhead

geometric-mean(raccoon) = 9x



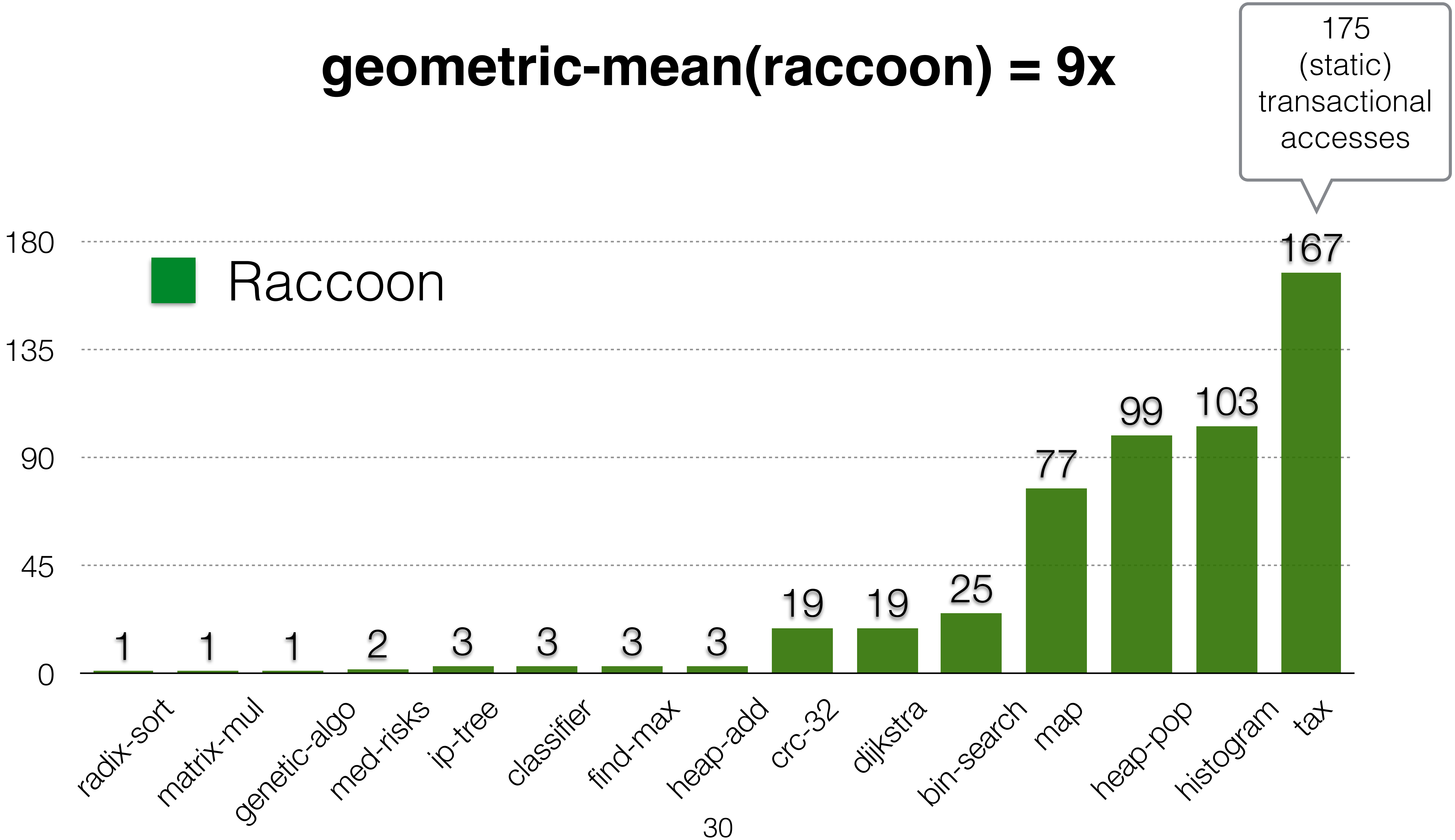
Raccoon’s Overhead

geometric-mean(raccoon) = 9x



Raccoon's Overhead

geometric-mean(raccoon) = 9x



Limitations of Raccoon

- Non-digital side-channels are outside of Raccoon's scope.
- Cannot obfuscate library calls and system calls.
- Cannot obfuscate all language features of the C99 standard:
  — Artifactual limitations: `break`/`continue`, secret loop trip counts.
  — Possibly-fundamental limitation: recursion.

There’s More!

Paper describes:

- How Raccoon does not introduce new termination-channel leaks.
- Obfuscation of multi-threaded programs.
- Optimizations to improve performance.
- Performance evaluation of software Path ORAM.
- Performance comparison with prior work.

Conclusions

- By applying program-level obfuscation, we can close many side-channels with one solution.
- This approach works in tandem with modern processors, instead of conflicting with them. Hence backward compatible with old hardware.
- Expands threat model, offers stronger protection.

Future Work

- Scale Raccoon to bigger programs:
  - Explore the use of special-purpose hardware.
  - Reduce overhead so that Raccoon's use is practical.

- Strengthen Raccoon's security guarantees:
  - Integrate fixed-time arithmetic libraries into Raccoon.
  - Increase the set of allowed language features.