Memory Safety for Embedded Devices with nesCheck

Daniele MIDI, Mathias PAYER, Elisa BERTINO
Purdue University

AsiaCCS 2017

Ubiquitous Computing and Security

Sensors and WSNs are pervasive

Small + cheap → smart thermostats, production pipelines, “precision” agriculture

Internet of Things as generalization

Smart embedded systems + Internet-based services

Security is paramount

Stringent requirements on:

- end-to-end system reliability
- trustworthy data delivery
- service availability

Wireless Sensor Networks (WSNs)

WSNs must be functional at any time.

But…

Unreliable medium

Constrained resources

Unattended environment

➔ Transient/permanent failures

Motivations & Premises

Low-level languages + no memory protection
NesC suffers same problems as C

Common techniques not applicable!
Very constrained platform, no virtual memory, high overhead, ...

High modularity + whole program analysis
Allows language-based techniques

Not all checks are needed
Some can be verified statically

nesCheck

Static Analysis + Dynamic Instrumentation

Automatically catch memory bugs,

provide sound memory safety guarantees

while minimizing performance overhead.

APPLICATIONS: Automatic hardening of embedded software, consumer and corporate devices, …

Memory Safety Goals

Bugs [static]

Find all statically-provable bugs → report errors

Violations [static]

Find all violations → report warnings

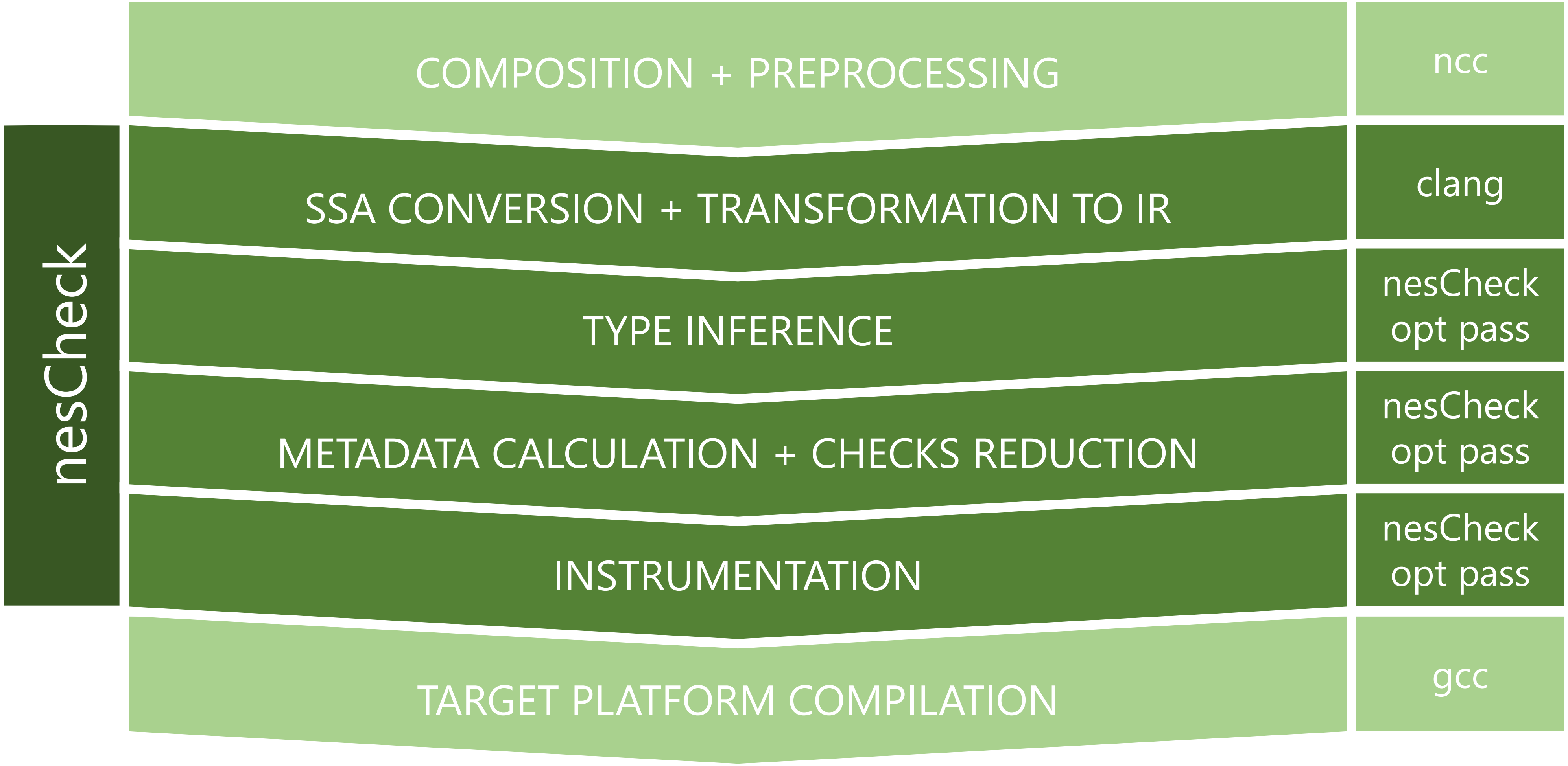
Checks reduction [static]

Statically determine “safe” violations

Runtime checks [dynamic]

Instrument remaining violations, catch all memory errors at runtime.

nesCheck Toolchain



Static Analysis

Type System and Inference Engine

Safe

Sequence

Dynamic

```
foreach declaration of pointer variable p do
    classify(p, SAFE);
foreach instruction I using pointer p do
    r ← result of(I);
    if I performs pointer arithmetic then
        classify(p, SEQ);
        classify(r, SAFE);
    if I casts p to incompatible type then
        classify(p, DYN);
        classify(r, DYN);
```

Operational Semantics | Type Inference

$$\text{Types}\ \frac{\Gamma(x) = \tau \quad \tau \in \{\texttt{Safe}, \texttt{Seq}, \texttt{Dyn}\}}{\Gamma \vdash x : \tau}$$

$$\text{ArithT1}\ \frac{\Gamma \vdash e_1 : \tau \quad \tau \in \{\texttt{Safe}, \texttt{Seq}\} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 + e_2 : \texttt{Seq}}$$

$$\text{ArithT2}\ \frac{\Gamma \vdash e_1 : \tau \quad \tau = \texttt{Dyn} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 + e_2 : \texttt{Dyn}}$$

$$\text{IllegCast}\ \frac{(E, x) \Rightarrow_l l : t \quad incompatible(t, t')}{\Gamma \vdash (t')x : \texttt{Dyn}}$$

Metadata

In-memory metadata
One instance per variable at any time

Explicit metadata variable
Logical variables across basic blocks

Metadata table entry
In-memory runtime information

```
void f(int a) {
1  int* p;
   metadata pmeta;
   if (a > 0)
2    p = malloc(4 * sizeof(int));
     pmeta.size = 4 * sizeof(int);
   else
3    p = malloc(20 * sizeof(int));
     pmeta.size = 20 * sizeof(int);
4  check(p[3], pmeta) && p[3] = 13;
}
```

Dynamic Instrumentation

Dynamic Checks Instrumentation

For any violating pointer dereference

Before `GetElementPointer` LLVM instruction:

- If pointer access was classified **SAFE** by static analysis, **skip check**.
- **Prepare bounds check:** `if (!checkBounds(p, offset, pmeta)) { trapFunction(); }`
- Check always `false`? → **Skip check**
  (e.g., `p[i]` for p with fixed length >= 3 and `i` inferred as 2)
- Check always `true`? → **Report memory bug**
  (e.g., `p[i]` for p with fixed length < 3 and `i` inferred as 2)
- **Add bounds check.**

Checks reduction

Based on type tracking and pointer usage

When propagated metadata results in constant check

Dynamic Checks Instrumentation

Optimizations to reduce metadata table lookups:

Functions taking pointer parameters:

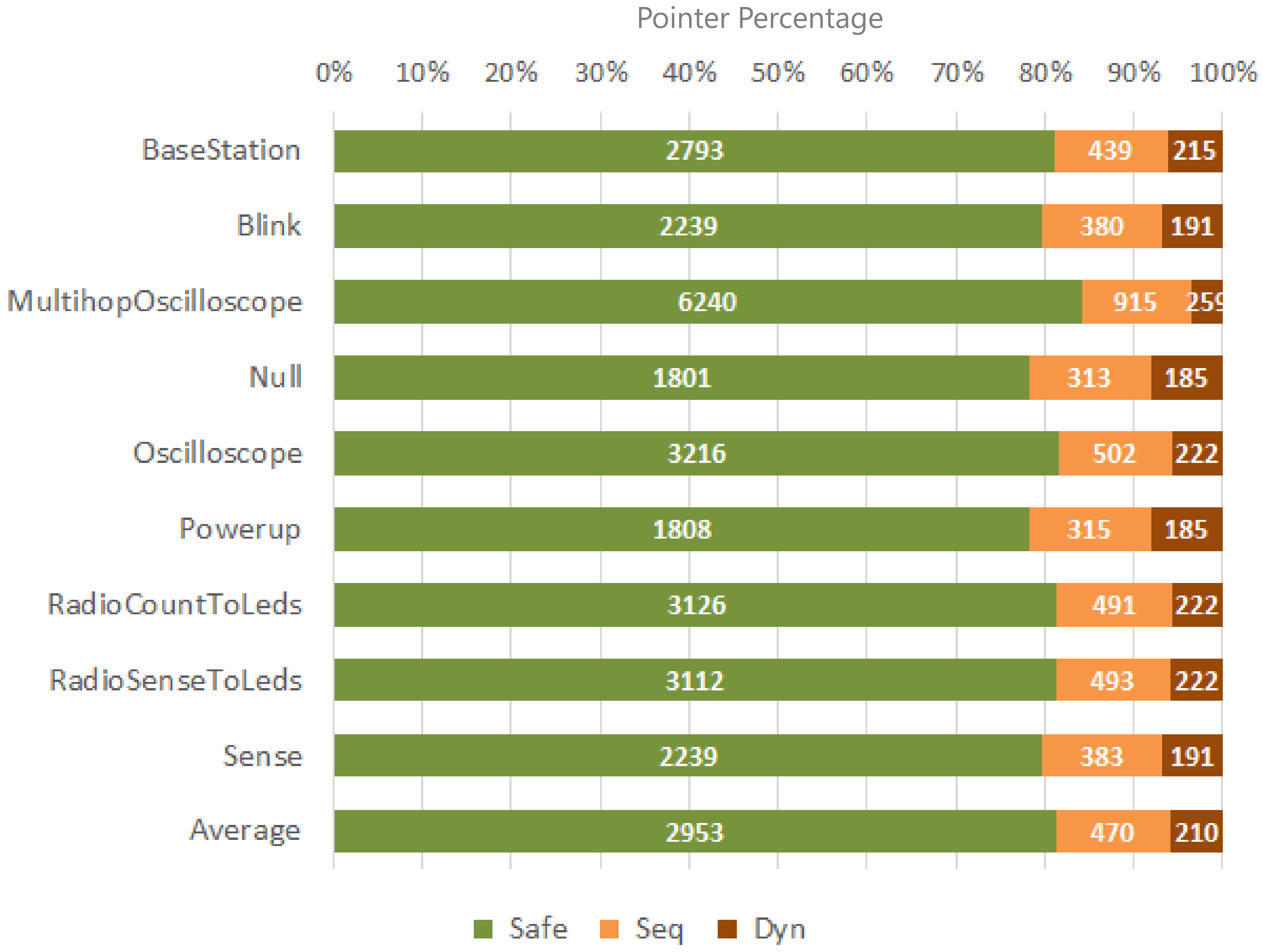
```
void f(int* p) → void f(int* p, metadata pmeta)
```

Functions returning pointers:

```
int* f() → {int*, metadata} f()
return p; → return {p, pmeta};
```

Evaluation Results

Type Inference



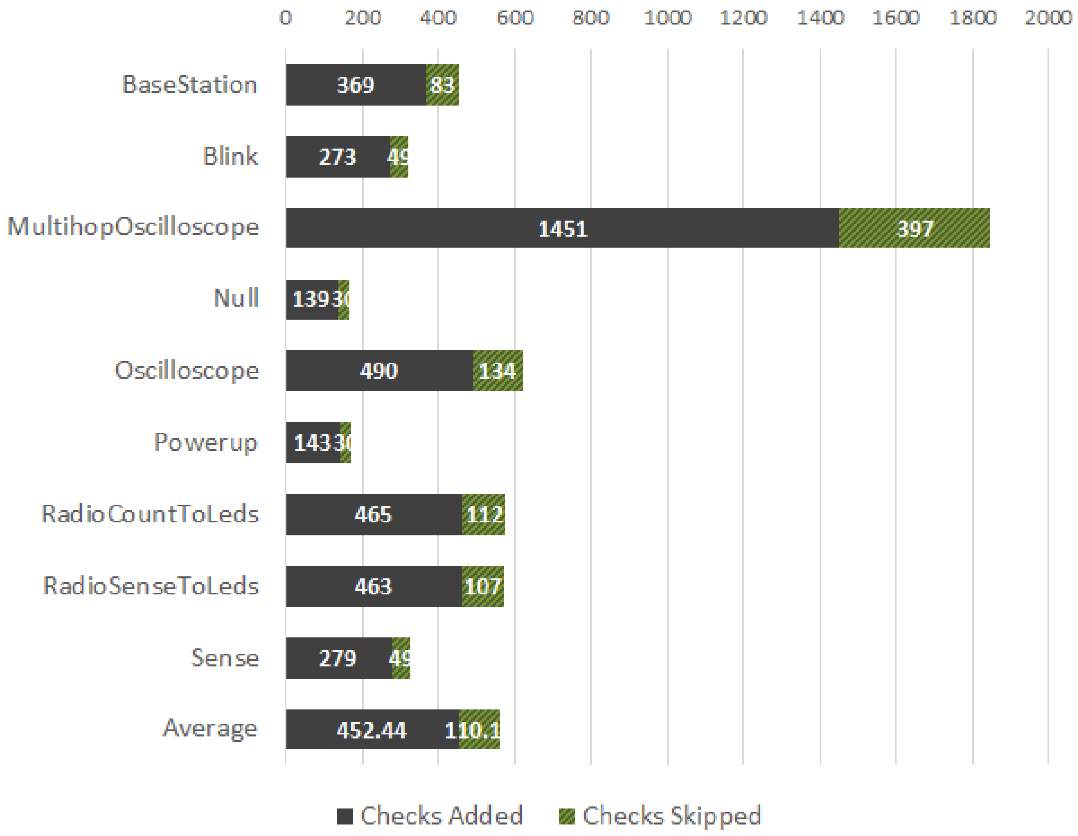
AVERAGES

Safe: 81%

Seq: 13%

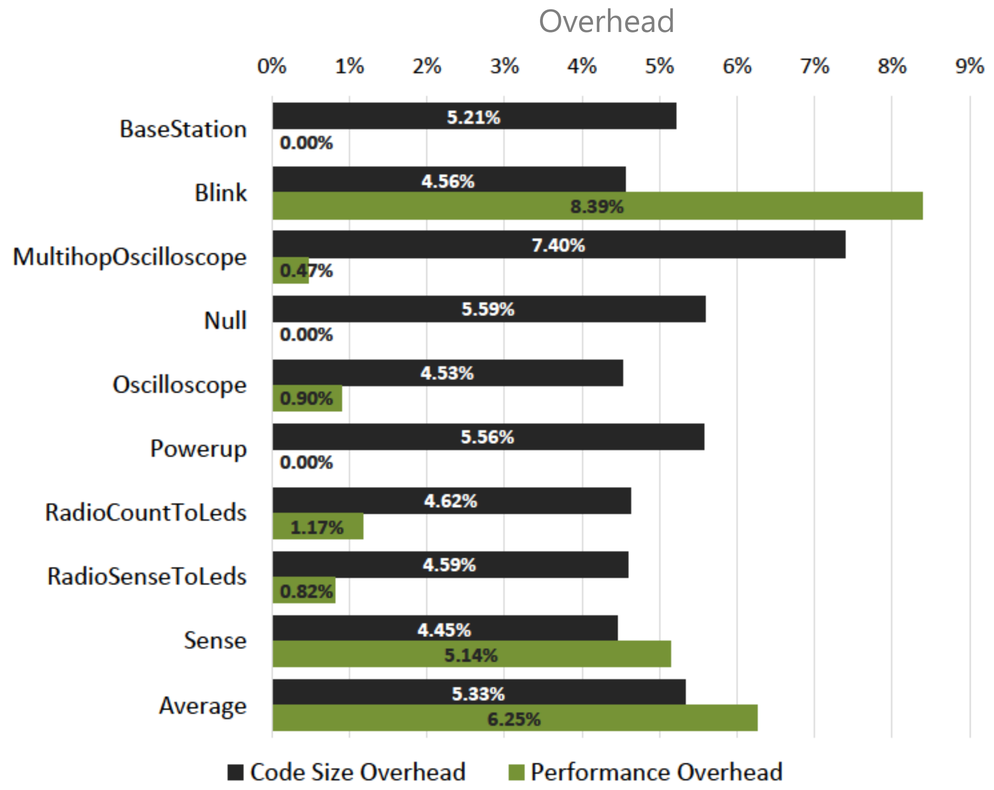
Dyn: 6%

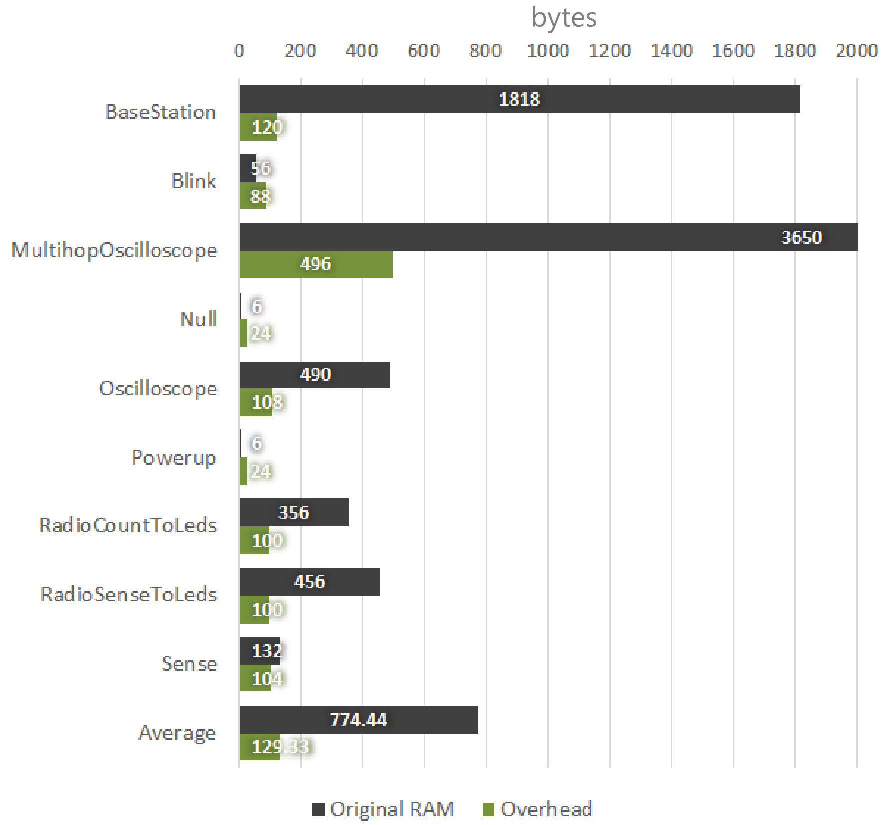
Checks Reduction



Average: 20% reduction

Code Size, Performance, and Memory Overhead

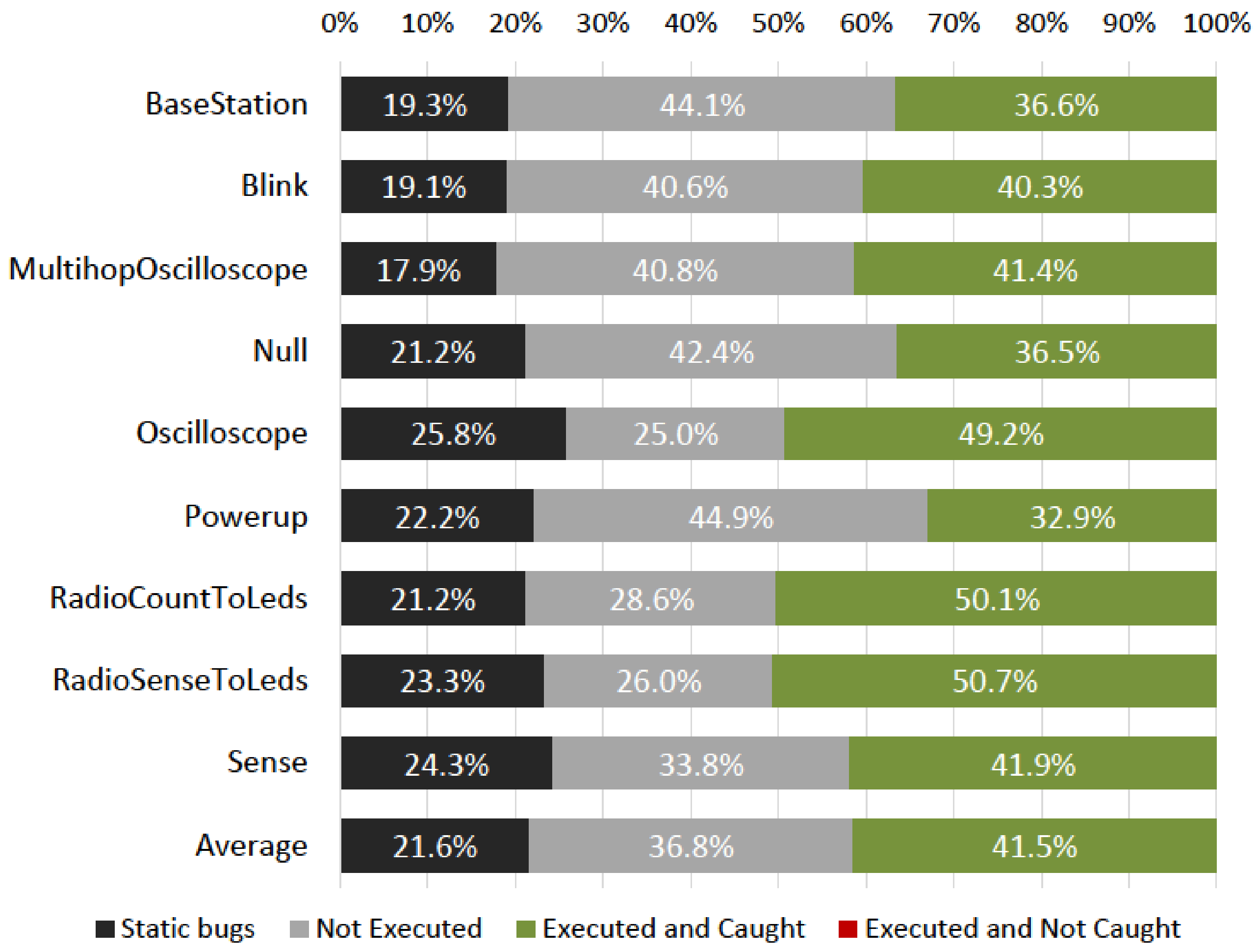




Code size 5%, performance 6%

As low as 7%, always <10kb

Fault Injection



AVERAGES

Static: 21.6%

Not Run: 36.8%

Dynamic (caught): 41.5%

Uncaught: 0%

State of the Art

CCured

Removes checks of SAFE pointers only

SoftBound

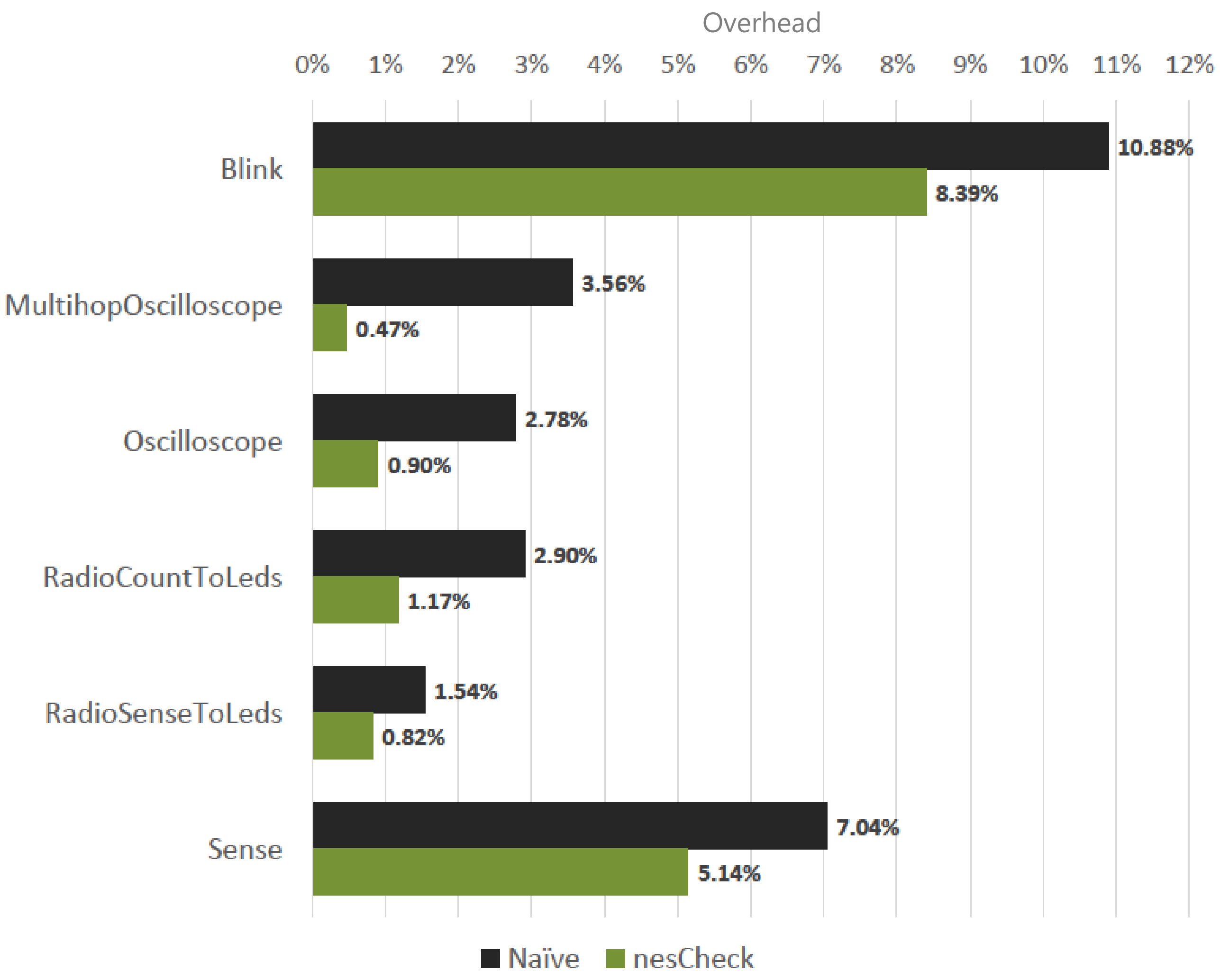
Instruments all pointers

SafeTinyOS

Requires extensive annotations or exclusion of entire components

Relies on Deputy source-to-source compiler

Naïve vs. Optimized Improvement



NAÏVE:
no check reduction optimizations

NESCHECK:
with full check reduction optimizations

Average improvement:
41.13%

Conclusion

nesCheck

Type system for pointer types: safe, seq, dyn

Statically prove pointer operations safe

Protect potentially unsafe operations at runtime

APPLICATIONS: Automatic hardening of embedded software, consumer and corporate devices, ...

https://github.com/HexHive/nesCheck