Memory Safety for Embedded Devices with nesCheck

Daniele MIDI, Mathias PAYER, Elisa BERTINO
Purdue University
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

- COMPOSITION + PREPROCESSING: ncc
- SSA CONVERSION + TRANSFORMATION TO IR: clang
- TYPE INference: nesCheck opt pass
- METADATA CALCULATION + CHECKS REDUCTION: nesCheck opt pass
- INSTRUMENTATION: nesCheck opt pass
- TARGET PLATFORM COMPILATION: gcc
Static Analysis
foreach declaration of pointer variable $p$ do
classify($p$, SAFE);

foreach instruction $I$ using pointer $p$ do
$r \leftarrow$ 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

\[
\begin{align*}
\text{Types} & \quad \frac{\Gamma(x) = \tau \quad \tau \in \{\text{Safe, Seq, Dyn}\}}{\Gamma \vdash x : \tau} \\
\text{ArithT1} & \quad \frac{\Gamma \vdash e_1 : \tau \quad \tau \in \{\text{Safe, Seq}\}}{\Gamma \vdash e_1 + e_2 : \text{Seq}} \\
\text{ArithT2} & \quad \frac{\Gamma \vdash e_1 : \tau \quad \tau = \text{Dyn}}{\Gamma \vdash e_1 + e_2 : \text{Dyn}} \\
\text{IllegCast} & \quad \frac{(E, x) \Rightarrow l \ l : t \quad \text{incompatible}(t, t')}{\Gamma \vdash (t')x : \text{Dyn}}
\end{align*}
\]
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

```c
void f(int a) {
    int* p;
    metadata pmeta;
    if (a > 0)
        p = malloc(4 * sizeof(int));
        pmeta.size = 4 * sizeof(int);
    else
        p = malloc(20 * sizeof(int));
        pmeta.size = 20 * sizeof(int);
    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
Optimizations to reduce metadata table lookups:

Functions taking pointer parameters:

```c
void f(int* p) \rightarrow void f(int* p, metadata pmeta)
```

Functions returning pointers:

```c
int* f() \rightarrow \{int*, metadata\} f()
return p; \rightarrow return \{p, pmeta\};
```
Evaluation Results
Type Inference

### Pointer Percentage Averages

<table>
<thead>
<tr>
<th>Module</th>
<th>Safe</th>
<th>Seq</th>
<th>Dyn</th>
</tr>
</thead>
<tbody>
<tr>
<td>BaseStation</td>
<td>2793</td>
<td>439</td>
<td>215</td>
</tr>
<tr>
<td>Blink</td>
<td>2239</td>
<td>380</td>
<td>191</td>
</tr>
<tr>
<td>MultihopOscilloscope</td>
<td>6240</td>
<td>915</td>
<td>259</td>
</tr>
<tr>
<td>Null</td>
<td>1801</td>
<td>313</td>
<td>185</td>
</tr>
<tr>
<td>Oscilloscope</td>
<td>3216</td>
<td>502</td>
<td>222</td>
</tr>
<tr>
<td>Powerup</td>
<td>1808</td>
<td>315</td>
<td>185</td>
</tr>
<tr>
<td>RadioCountToLeds</td>
<td>3126</td>
<td>491</td>
<td>222</td>
</tr>
<tr>
<td>RadioSenseToLeds</td>
<td>3112</td>
<td>493</td>
<td>222</td>
</tr>
<tr>
<td>Sense</td>
<td>2239</td>
<td>383</td>
<td>191</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>2953</strong></td>
<td><strong>470</strong></td>
<td><strong>210</strong></td>
</tr>
</tbody>
</table>

**AVERAGES**

- Safe: 81%
- Seq: 13%
- Dyn: 6%
Checks Reduction

Average: 20% reduction
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

Average improvement: 41.13%

NAÏVE: no check reduction optimizations
NESCHECK: with full check reduction optimizations
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