Towards a Resilient Operating System for Wireless Sensor Networks

Hyoseung Kim    Hojung Cha
Yonsei University, Korea

2006. 6. 1.

Hyoseung Kim
hskim@cs.yonsei.ac.kr
Motivation (1)

- Problems: Application errors on sensor networks

  Diverse hardware devices
  Cooperate with large number of nodes
  Unattended operation for long-time

Kernel data corruption
Kernel code execution
Hardware control faults

Kernel fail & sensor node crash

No way to recover from system failure
Crashed sensor node is useless in the field

System safety like general purpose OS needed!
Motivation (2)

• Popular sensor nodes: Telos / Mica
  - No MMU, privileged mode, and exceptions

• Current sensor network systems
  - No system safety mechanism
Towards Resilient Sensor Networks

- **Ensure system safety at the operating system level**
  - *Applications*: we cannot write the code safely all the time
  - *Sensor node hardware*: does not easily detect application errors

- **Objectives**
  - Provide error-safe mechanism on resource-constrained sensor devices
  - Do not require any hardware supports
  - Do not require users to learn new programming language semantics
Previous Work

- **SFI (Software Fault Isolation)**
  - Requires MMU for segmentation and stack safety
  - Designed for fixed-length instruction architecture

- **Proof-carrying Code**
  - The automatic policy generator does not exist

- **Programming language approaches**
  - Cyclone, Control-C, Cuckoo
  - Users should be aware of the different usages of pointer/array
  - Requires hardware supports for stack safety

RETOS Architecture

Resilient, Expandable, and Threaded Operating System

Code (Flash Rom)
- Applications
- Static/dynamic code checking
- Modules
- Scheduler
- Module Manager
- Function Table
- Variable timer
- Erred apps manager
- System call

Data (RAM)
- User stack
- User data
- Module data
- Single kernel stack
- Kernel Data
RETOS Architecture

Resilient, Expandable, and Threaded Operating System

Code (Flash Rom)

- Applications
- Static/dynamic code checking

Kernel area

- Modules
- Module Manager
- Function Table

User area

- Scheduler
- Policy
- Priority
- Variable timer
- Erred apps manager
- System call

Data (RAM)

- User stack
- User data
- Module data
- Single kernel stack
- Kernel Data
RETOS Architecture

Resilient, Expandable, and Threaded Operating System

- Code (Flash Rom)
  - Applications
  - Multi-threads
  - Modules
  - Scheduler
    - Policy
    - Priority
  - Variable timer
  - Erred apps manager
  - System call

- Data (RAM)
  - User stack
  - User data
  - Module data
  - Single kernel stack
  - Kernel Data

- Static kernel
- Dynamic kernel
- User space
Ensuring System Safety

- **Principle**
  - Dual Mode operation & static/dynamic code checking

![Diagram](image)
Dual Mode Operation

• Why dual-mode is needed?
  – Static/dynamic code checking evaluates if the application modifies data or issues code in its allocated data
  – Preemption would invoke problems

• (ex) *Thread_A* is preempted by *Thread_B*

![Diagram showing kernel/user separation and code checking](image)
Static/Dynamic Code Checking

- Use data and code area within application itself
- Restrict direct hardware resource manipulation

```
#include <lo.h>
#include <syscalls.h>
#include <pthread.h>

... 

/* main thread */
int main ()
{
    while (1) {
        void (*func)(void);
        /* radio packet recv */
        radio_recv(PORT, &pkt);
        func = Foo_Func[pkt->id];
        /* do something*/
        func();
    }
}
```

```
main:  /* prologue: frame size - 44 */
    .L_FrameSize_main=4x2e
    .L_FrameOffset_main=0x2e
    mov  #(__stack-44), r1
    /* prologue end (size-2) */
    .L23:
    /* do something */
    call  &func
    mov  #110(6), r15
    call  #syscall10
    jmp  .L5
    br  #__stop_progExec___
    /* epilogue: frame size=44 */
    add  #44, r1
    br  #__stop_progExec___
    /* epilogue end (size=4) */
    /* function main size 168 (162) */
    .size main, Lfe6-main__
```

```
mov  #1792, r1
mov  #6, r15
push r15
mov  t0=0608,r15
cmp  12(r15),r1
jnc  $+8
cmp  14(r15),r1
jnc  $+6
call  #488
pop  r15
call  #494
mov  #1008, r13
clr r14
mov  #0, r15
push r15
mov  t0=0608,r15
cmp  12(r15),r1
jnc  $+8
cmp  14(r15), r1
jnc  $+6
call  #488
```
Implementation

• **RETOS implemented on TI MSP430 F1611**
  – 8MHz core clock, 10Kb RAM, 48Kb Flash ROM
  – Current version: 0.92 (May 24 2006)

• **Peripheral supports**
  – 2.4Ghz RF Module (Chipcon 2420)
  – Ultrasound, Humidity, and Light sensors

• **Applications: MPT (Multi-Party Object Tracking)**
Functionality Test (1)

- **Classify safety domain into four parts:**
  - Stack / Data safety → Stack and data area
  - Code safety → Control flow
  - Hardware safety → Immediate hardware control

<table>
<thead>
<tr>
<th>Hardware safety</th>
<th>Stack safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disable interrupt</td>
<td>General/Mutual recursive call</td>
</tr>
<tr>
<td>DINT(); → Detected by <strong>Static check</strong></td>
<td>void foo() { foo(); }</td>
</tr>
<tr>
<td>Flash ROM writing (memory mapped registers)</td>
<td>void foo() { bar(); } void bar() { foo(); }</td>
</tr>
<tr>
<td>FCTL1 = FWKEY+WRT; → Detected by <strong>Static check</strong></td>
<td>→ Detected by <strong>Dynamic check</strong></td>
</tr>
</tbody>
</table>
## Functionality Test (2)

<table>
<thead>
<tr>
<th>Data safety</th>
<th>Code safety</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Directly addressed pointers</strong></td>
<td><strong>Directly addressed function call</strong></td>
</tr>
</tbody>
</table>
| int *tmp = 0x400;  
  *tmp = 1;  
  → Detected by Static check | void (*func)(void) = 0x1000;  
  func();  
  → Detected by Static check |
| **Illegal array indexing** | **Corrupted return address**  
  (Buffer overflow) |
| /* array in heap area */  
  int array[10];  
  ...  
  for(i = 10; i > 0; i--) {  
    array[i-100] = i;  
  }  
  → Detected by Dynamic check | void func() {  
  int array[5], i;  
  for(i = 0; i < 10; i++)  
    array[i] = 0;  
  }  
  → Detected by Dynamic check |
Overhead Analysis (1)

- **Execution time running in user mode**
  - Programs, which require more memory access than complex arithmetic calculations, shows larger overhead

<table>
<thead>
<tr>
<th>Category</th>
<th>Computation Based</th>
<th>Memory Access Based</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPT_mobile</td>
<td>386566 → 394624 cycles/sec</td>
<td></td>
</tr>
<tr>
<td>MPT_backbone</td>
<td>40326 → 40508</td>
<td></td>
</tr>
<tr>
<td>sensing</td>
<td>1056 → 1096</td>
<td></td>
</tr>
<tr>
<td>r_recv</td>
<td>4704 → 5010</td>
<td></td>
</tr>
<tr>
<td>r_send</td>
<td>24815 → 26176</td>
<td></td>
</tr>
<tr>
<td>pingpong</td>
<td>1243 → 1347</td>
<td></td>
</tr>
<tr>
<td>surge</td>
<td>58723 → 62688</td>
<td></td>
</tr>
</tbody>
</table>
Overhead Analysis (2)

- **Application code size comparison**
  - Code size increased, independent of application type
  - Applications are inherently small in size, separated from kernel

![Bar chart showing code size comparisons](chart.png)

- MPT_mobile: 8726 → 10046 bytes
- MPT_backbone: 614 → 682 bytes
- sensing: 774 → 835 bytes
- r_recv: 902 → 1004 bytes
- r_send: 946 → 1014 bytes
- pingpong: 478 → 510 bytes
- surge: 1436 → 1610 bytes
Overhead Analysis (3)

- Dual mode operation

<table>
<thead>
<tr>
<th></th>
<th>Single mode (cycles)</th>
<th>Dual mode (cycles)</th>
<th>Overhead (cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>system call</strong> (led toggle)</td>
<td>264</td>
<td>302</td>
<td>38</td>
</tr>
<tr>
<td><strong>system call</strong> (radio packet send)</td>
<td>352</td>
<td>384</td>
<td>32</td>
</tr>
<tr>
<td><strong>timer interrupt</strong> (invoked in kernel)</td>
<td>728</td>
<td>728</td>
<td>0</td>
</tr>
<tr>
<td><strong>timer interrupt</strong> (invoked in user)</td>
<td>728</td>
<td>760</td>
<td>32</td>
</tr>
</tbody>
</table>

- Summary
  - Computational overhead & larger code size are inevitable to provide system safety in sensor node platform
  - However, they are considered as a fair trade-off compared to a system failure
Conclusions

• **Safety mechanism for wireless sensor networks**
  – Separate errant application from the kernel, “Kernel never die”
  – Automatically recover from serious errors
  – Useful in the real, large-scale sensor networks

• **Current status**
  – RETOS is being developed in our research group
  – Being ported to other processors (AVR)
    - *Mobiquitous 2006, San Jose, July 17-21*
Questions
Watchdog Timer

- Not easy to recognize and handle problems such as:
  - Memory access beyond the application area
  - Immediate hardware control

- Watchdog timer simply makes the system be restarted
  - One errant application can stop all other applications
  - Disturbance of long-term operations of sensor applications
Multithread vs. Event-driven

- **Concurrency for computation based applications**
  - Run-to-completion vs. preemptive time-sliced thread

- **Poor software structure on event-driven model**
  - One conceptual function split into multiple functions
    - Loss of control structures / local variables

- **Simplified synchronization methods on thread model**
  - Like “atomic” operation in nesC
Limitations

- **Library codes**
  - We assume the libraries always operate safely
  - In real situation: If a user passes an invalid address to `memcpy()`?
  - Make wrapper functions that checks address parameters

- **Divide by zero**
  - Hardware multiplier/divider equipped processor: instruction level checking
  - Emulated: library level checking

- **Intentionally skips the code checking sequences**
  - User authentication on code updating would be required
## Sensor OS Comparison

<table>
<thead>
<tr>
<th></th>
<th>System Safety (kernel-user separation)</th>
<th>Light-weight dynamic reprogramming</th>
<th>Programming Model</th>
<th>Programming Language</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>TinyOS</strong></td>
<td>No</td>
<td>No</td>
<td>Event-driven</td>
<td>nesC</td>
</tr>
<tr>
<td><strong>SOS</strong></td>
<td>No</td>
<td>Yes</td>
<td>Event-driven</td>
<td>Standard C</td>
</tr>
<tr>
<td><strong>MANTIS</strong></td>
<td>No</td>
<td>No</td>
<td>Preemptive Multithreading</td>
<td>Standard C</td>
</tr>
<tr>
<td><strong>RETOS</strong></td>
<td>Yes</td>
<td>Yes</td>
<td>Preemptive Multithreading</td>
<td>Standard C</td>
</tr>
</tbody>
</table>