RapidPatch: Firmware Hotpatch for Embedded Devices

Yi He*, Zhenhua Zou, Kun Sun, Zhuotao Liu, Ke Xu, Qian Wang, Chao Shen, Zhi Wang, Qi Li
MCU-based embedded devices are everywhere

Energy Conserving

Cortex-M MCU

2 weeks

Cortex-A SoC

18 hours

Low Cost (< $1)

Real Time

Real-Time Servo Motor Control

<table>
<thead>
<tr>
<th>OS</th>
<th>CPU</th>
<th>Mem</th>
<th>Storage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux</td>
<td>&gt; 520 MHz</td>
<td>&gt; 128 Mb</td>
<td>&gt; 256 Mb</td>
</tr>
<tr>
<td>RTOS</td>
<td>64 ~ 240 MHZ</td>
<td>128 ~ 512 Kb</td>
<td>&lt; 2Mb</td>
</tr>
</tbody>
</table>

Resource-Constrained
Their firmware updates are delayed

Previous work\textsuperscript{[1]} indicates that 28.25\% (385,060 / 1,362,906) IoT devices with at least one N-Day vulnerabilities are exposed to the attackers.

The obstacles of patching these devices

Patch Development

- Upstream Library Developers
- Upstream RTOS Developers
- Third-Party Libs
- RTOS

Patch Write

- Merge source code patch for multiple codebases
- NUTTX Real-Time Operating System
- Zephyr

Patch Propagation

- Downstream IoT Developers
- Too many patches need to be tested on the fragmentation devices
- Need to reboot for installing the updates

Patch Test

Patch Deploy
Our solution: new patching workflow

RapidPatch Workflow: one patch for all the heterogeneous devices with the same vulnerability.
Solutions for patch development

Obstacle-1: Patch Writing
Obstacle-2: Patch Propagation
Obstacle-3: Patch Testing
Obstacle-4: Patch Deploy

Previous: Merge C source code patch to generate binary patch for each type of device

Now: Use a single eBPF bytecode patch for all the devices
Binary Patch vs Bytecode Patch

Need to prepare patches with different architectures.

Why use eBPF as the VM-based language? Why not Lua, Python, Javascript, …?

1. Simple and efficient
2. Flexible, use C grammar
The basic idea of eBPF patch

Skip or replace the vulnerable C function with eBPF code

Buggy Program

Func 1
...
Vulnerable Function
Func n

Patch Runtime

run eBPF VM

Switch to and restore from patch runtime
Solutions for patch testing

Obstacle-1: Patch writing

Obstacle-2: Patch Propagation

Obstacle-3: Patch Testing

Obstacle-4: Patch Deploy

The safety is ensured by software fault isolation (SFI) of the patch runtime.
Which patches have no side effects?

**Filter Patch:**

```c
1 int pico ICMPv6_send_ер со nу (struct pico_frame *echo) {
2   // ... omit
3   if (echo->transport_len < PICO ICMP6HDR_ECHO_REQUEST_SIZE) {
4     return -1; // invalid packet
5   }
6   /* bug: When the echo->transport_len is less than the
7      PICO ICMP6HDR_ECHO_REQUEST_SIZE, the memcpy len will
8      arithmetic underflow here */
9   memcpy(reply->payload, echo->payload, (uint32_t)
10      (echo->transport_len - PICO ICMP6HDR_ECHO_REQUEST_SIZE));
11 }
```

(a). The C Source Code Patch

```c
#include "ebpf_helper.h"
const int PICO ICMP6HDR_ECHO_REQUEST_SIZE = 8;
uint64_t filter(stack_frame *frame) {
  uint8_t *echo = (uint8_t *)(frame->r0);
  uint16_t *transport_len_ptr = (uint16_t *)(echo + 38);
  uint16_t transport_len = (uint16_t *)(transport_len_ptr);
  if (transport_len >= PICO ICMP6HDR_ECHO_REQUEST_SIZE) {
    return retv_return(OP_PASS, 0);
  }
  return set_return(OP_DROP, -1);
}
```

(b). The eBPF Filter Patch

**Code-Replace Patch:**

```c
1 void shell_spaces_trim(char *str){
2   // ...
3   for (u16_t j = i + 1; j < len; j++) {
4      // ...
5      memmove(&str[i + 1], &str[j], len - i + 1);
6      i += len - j + 1;
7   }
8   // ...
9 }
```

(a). The C Source Code Patch

```c
void ebpf_spaces_trim(stack_frame *frame) {
  // ...
  for (u16_t j = i + 1; j < len; j++) {
    // ...
    C_CALL(FUNC_memmove,
      &str[i + 1], &str[j],
      len - j + 1);
  }
}
```

(b). The eBPF Code Replace Patch

No. It calls C functions.

Yes. It only reads the memory.
Patch Verifier and SFI

eBPF Patch

1. Contain write instructions?
2. Contain C function callings?

Yes

No

Waive Tests

Need Tests

Filter Patch

Code-Replace Patch

SFI:
Prevent write
Restrict loop iterations

Add loop limitation for code with unbounded loops.
Solutions for patch deploy

Obstacle-1: Patch writing
Obstacle-2: Patch Propagation
Obstacle-3: Patch Testing
Obstacle-4: Patch Deploy

Live Patch without reboot or delay the real-time tasks.

Industry devices  Medical devices
Challenges for hotpatching embedded devices

Approach-1: A/B Scheme

- Switch to new firmware

- Old Firmware
- New Firmware

Limitations:
1. The storage is insufficient.
2. Still need to reboot during switch.

Approach-2: Inline Hook

- Modify the first instruction to a jmp trampoline.

Limitation:
The code of embedded devices are stored in ROM (nor-flash) and modifying is time consuming which break the real-time constraint.
What is real-time constraint?

While (true) {
    process_task();
    sleep_for_us(2000);
}

In Windows/Linux, this code has very large jitter.

Desired Loop Time

Jitter

1. Have very precise hardware timer.
2. All high priority tasks should not be prevent by low priority tasks.
3. Have hard deadline for some tasks.

Real-time devices:

Control commands should be executed in time
Patch Deploying under real-time constraint

Method-1: Compile-time instrument

```c
int buggy_function() {
    check_has_patch();
    // ...
}
```

Method-2: MCU patching feature

Method-3: hardware breakpoint based KPorbe

<table>
<thead>
<tr>
<th>Patch Point</th>
<th>Support Device</th>
<th>#Patch</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed Patch Points</td>
<td>Function Begin</td>
<td>All</td>
</tr>
<tr>
<td>FPB</td>
<td>Basic Block</td>
<td>Only Cortex-M3/M4</td>
</tr>
<tr>
<td>KProbe</td>
<td>Basic Block</td>
<td>Cortex-M3~M55(all), RISC-V</td>
</tr>
</tbody>
</table>

The patch task should not block the motor control task.
Put it together

1. Get source code patch
2. Merge code to firmware codebase
3. Generate binary patch via differencing
4. Test patch on devices
5. Reboot devices to update

Bytecode Patch

Patch Verifier

Hotpatching

Directly use the upstream bytecode patch for heterogeneous devices.

Waive test and can deploy at any time with Live Patch Technology.
RapidPatch Implementation

Source Code:
https://github.com/IoTAccessControl/RapidPatch

<table>
<thead>
<tr>
<th>Module</th>
<th>#LoC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Patch Control</td>
<td>~ 1200 C</td>
</tr>
<tr>
<td>Patch Core</td>
<td>~ 2200 C</td>
</tr>
<tr>
<td>Libebpf</td>
<td>~ 3400 C</td>
</tr>
<tr>
<td>Patch Tool</td>
<td>~ 700 Python</td>
</tr>
<tr>
<td>Patch Verifier</td>
<td>~ 1200 Python</td>
</tr>
</tbody>
</table>

IoT Development Kits:
Usability Evaluation

90% of the CVEs can be patched.

56% of them do not need test.

76% of the patches for high risk CVEs do not need test.

Failed Cases:
1. change too many functions.
2. change the marco or inline functions.

Can be used for MCUs with different architectures, such as Cortex-M, Xtensa, RISC-V

<table>
<thead>
<tr>
<th>Device MCU</th>
<th>Arch</th>
<th>Frequency</th>
<th>Flash</th>
<th>SRAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>NRF52840</td>
<td>Cortex-M4</td>
<td>64MHz</td>
<td>1MB</td>
<td>256KB</td>
</tr>
<tr>
<td>STM32L475</td>
<td>Cortex-M4</td>
<td>80MHz</td>
<td>512KB</td>
<td>128KB</td>
</tr>
<tr>
<td>STM32F429</td>
<td>Cortex-M4</td>
<td>180MHz</td>
<td>2MB</td>
<td>256KB</td>
</tr>
<tr>
<td>ESP-WROOM32</td>
<td>Xtensa</td>
<td>240MHz</td>
<td>448KB</td>
<td>520KB</td>
</tr>
<tr>
<td>GD32VF103</td>
<td>RISC-V32</td>
<td>108MHz</td>
<td>128KB</td>
<td>32KB</td>
</tr>
</tbody>
</table>

Support various RTOSs, such as NuttX, FreeRTOS, Zephyr, and LiteOS
Patch Runtime Performance Evaluation

Delays incurred by different hotpatching strategies is about $1 \sim 4$ us.

<table>
<thead>
<tr>
<th></th>
<th>Fixed Patch Point</th>
<th>FPB</th>
<th>Debug Monitor</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>OP</td>
<td>Cycles</td>
<td>Time</td>
</tr>
<tr>
<td>No Patch</td>
<td>66</td>
<td>1.03</td>
<td>0</td>
</tr>
<tr>
<td>Pass (Continue)</td>
<td>66</td>
<td>1.03</td>
<td>395</td>
</tr>
<tr>
<td>Drop / Redirect</td>
<td>66</td>
<td>1.03</td>
<td>120</td>
</tr>
</tbody>
</table>

The average delays incurred by eBPF patch execution is less than 5 us.

<table>
<thead>
<tr>
<th>CVE</th>
<th># of BPF Instructions</th>
<th>eBPF Interpreter</th>
<th>eBPF-Jit</th>
<th>Memory (Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>8</td>
<td>27.3 $\mu$s</td>
<td>1.7 $\mu$s</td>
<td>56</td>
</tr>
<tr>
<td>c2</td>
<td>16</td>
<td>8.5 $\mu$s</td>
<td>1.6 $\mu$s</td>
<td>48</td>
</tr>
<tr>
<td>c3</td>
<td>100</td>
<td>133.3 $\mu$s</td>
<td>14.7 $\mu$s</td>
<td>260</td>
</tr>
<tr>
<td>c4</td>
<td>12</td>
<td>9.5 $\mu$s</td>
<td>2.0 $\mu$s</td>
<td>68</td>
</tr>
<tr>
<td>c5</td>
<td>14</td>
<td>23.5 $\mu$s</td>
<td>1.5 $\mu$s</td>
<td>48</td>
</tr>
<tr>
<td>c6</td>
<td>55</td>
<td>51.2 $\mu$s</td>
<td>4.4 $\mu$s</td>
<td>232</td>
</tr>
<tr>
<td>c7</td>
<td>46</td>
<td>26.8 $\mu$s</td>
<td>1.8 $\mu$s</td>
<td>188</td>
</tr>
<tr>
<td>c8</td>
<td>10+10</td>
<td>14.9 $\mu$s +16.2 $\mu$s</td>
<td>2.8 $\mu$s +2.7 $\mu$s</td>
<td>56+68</td>
</tr>
<tr>
<td>c9</td>
<td>10</td>
<td>28.1 $\mu$s</td>
<td>1.8 $\mu$s</td>
<td>52</td>
</tr>
<tr>
<td>c10</td>
<td>7</td>
<td>10.1 $\mu$s</td>
<td>1.4 $\mu$s</td>
<td>48</td>
</tr>
<tr>
<td>c11</td>
<td>7</td>
<td>9.5 $\mu$s</td>
<td>1.6 $\mu$s</td>
<td>48</td>
</tr>
<tr>
<td>c12</td>
<td>36</td>
<td>22.2 $\mu$s</td>
<td>3.9 $\mu$s</td>
<td>156</td>
</tr>
</tbody>
</table>
End-to-end Latency Evaluation

The overall request latency incurred by patch (KProbe) is less than 0.6% (JIT mode).

Compile-time instrument all the possible patching functions can bring 0.5% ~ 19% delay.
Limitations and Future Works

1. Can we automatically generate eBPF patch from C source patch?

2. Can we automatically identify the vulnerable function rather than manually verify the target Library version?

3. How to tolerate patches with logic bugs (incorrect patch)?
   Test cases with fork execution?

Future work: Implement fault isolation for all patches and used RapidPatch in real products (arm Linux).
Conclusion

★ It is challenging to hotpatch MCU-based embedded devices
  ○ need to prepare patches for too many heterogeneous devices
  ○ need to test patch on every types of devices
  ○ hotpatching without break the real-time constraint

★ RapidPatch: a new patch workflow for patching embedded devices
  ○ one patch for all the devices with the same vulnerability
  ○ multiple hotpatching strategies for different MCUs
  ○ most of the patches can waive tests
  ○ negligible overhead ( < 0.6%)
Related Work

★ IoT Firmware Update
  ○ Over-The-Air Update [ATC 19], [ICDCS 19], [ACSAC 20]

★ Hotpatching
  ○ HERA [NDSS 21], Android Live Patch [Security 17], [NDSS 18]

★ eBPF based system enhancement
  ○ System Observability [ATC 16], Performance Profiling [ATC 19]
  ○ Container Security [ATC 20]
  ○ JIT Implementation Formal Verification [OSDI 20]

We propose the first eBPF based architecture-independent patching mechanism.
Thanks for Listening
Q&A