RapidPatch: Firmware Hotpatch for Embedded Devices

Yi He*, Zhenhua Zou, Kun Sun, Zhuotao Liu, Ke Xu,
Qian Wang, Chao Shen, Zhi Wang, Qi Li

Paper

Slide

Code

清華大學
Tsinghua University

MCU-based embedded devices are everywhere

Energy Conserving

Cortex-M MCU

2 weeks

Cortex-A SoC

18 hours

Low Cost (< $1)

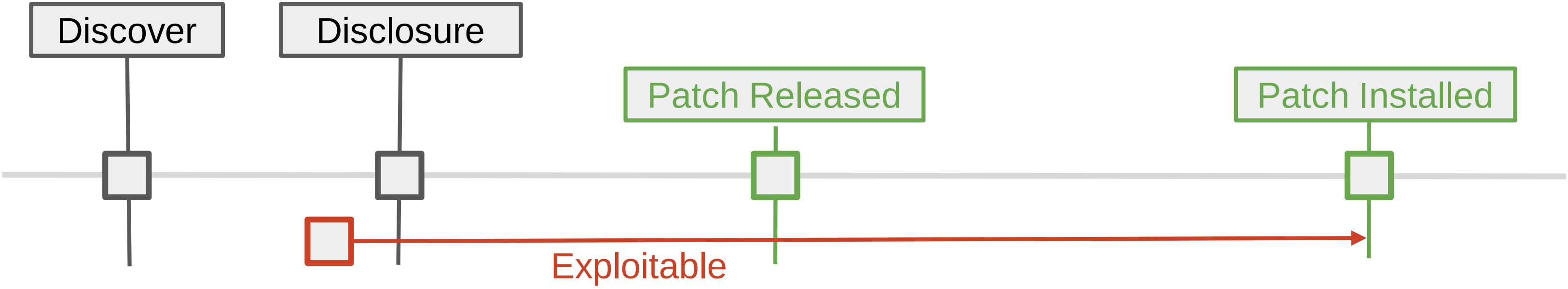
Real Time

Real-Time Servo Motor Control

Resource-Constrained

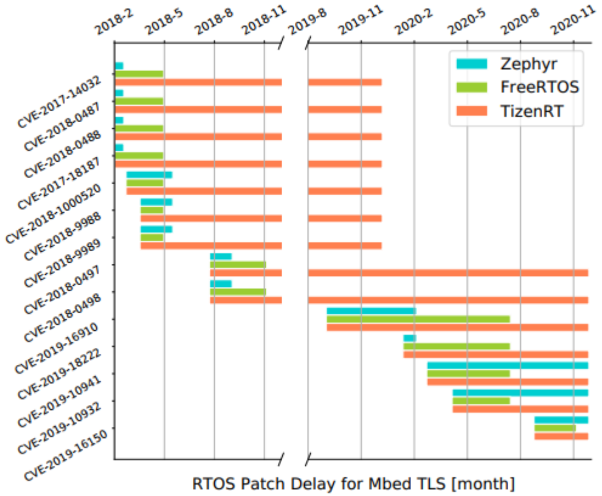
OS	CPU	Mem	Storage
Linux	> 520 MHz	> 128 Mb	> 256 Mb
RTOS	64 ~ 240 MHZ	128 ~ 512 Kb	< 2Mb

Their firmware updates are delayed



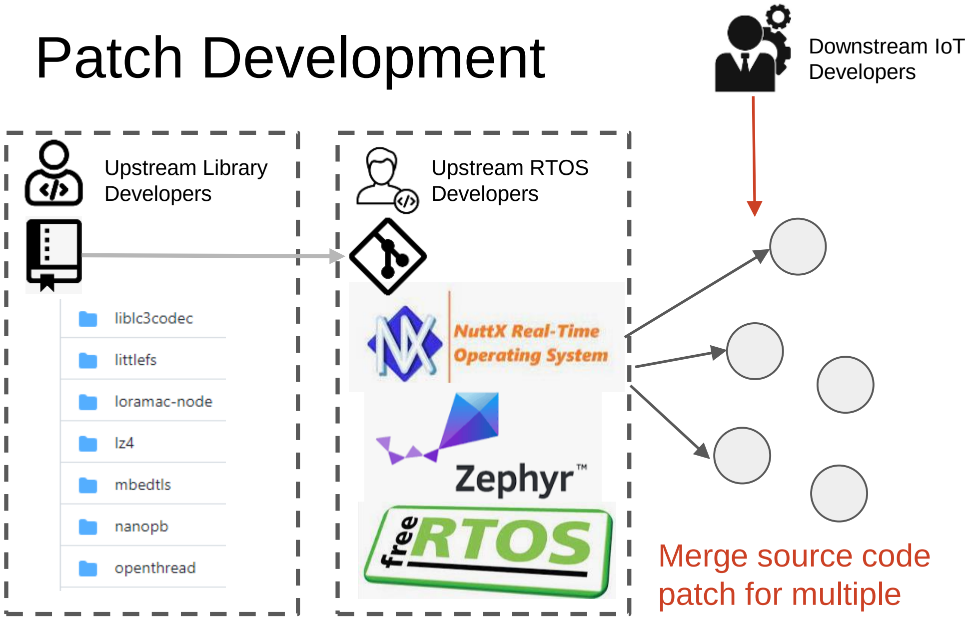
Previous work[1] indicates that 28.25% (385,060 / 1,362,906) IoT devices with at least one N-Day vulnerabilties are exposed to the attackers.

[1]. A Large-scale Empirical Study on the Vulnerability of Deployed IoT Devices. Binbin Zhao et al. TDSC 2020.



The obstacles of patching these devices

Patch Development



Merge source code patch for multiple codebases

Third-Party Libs

RTOS

Patch Write

Patch Propagation

Patch Propagation

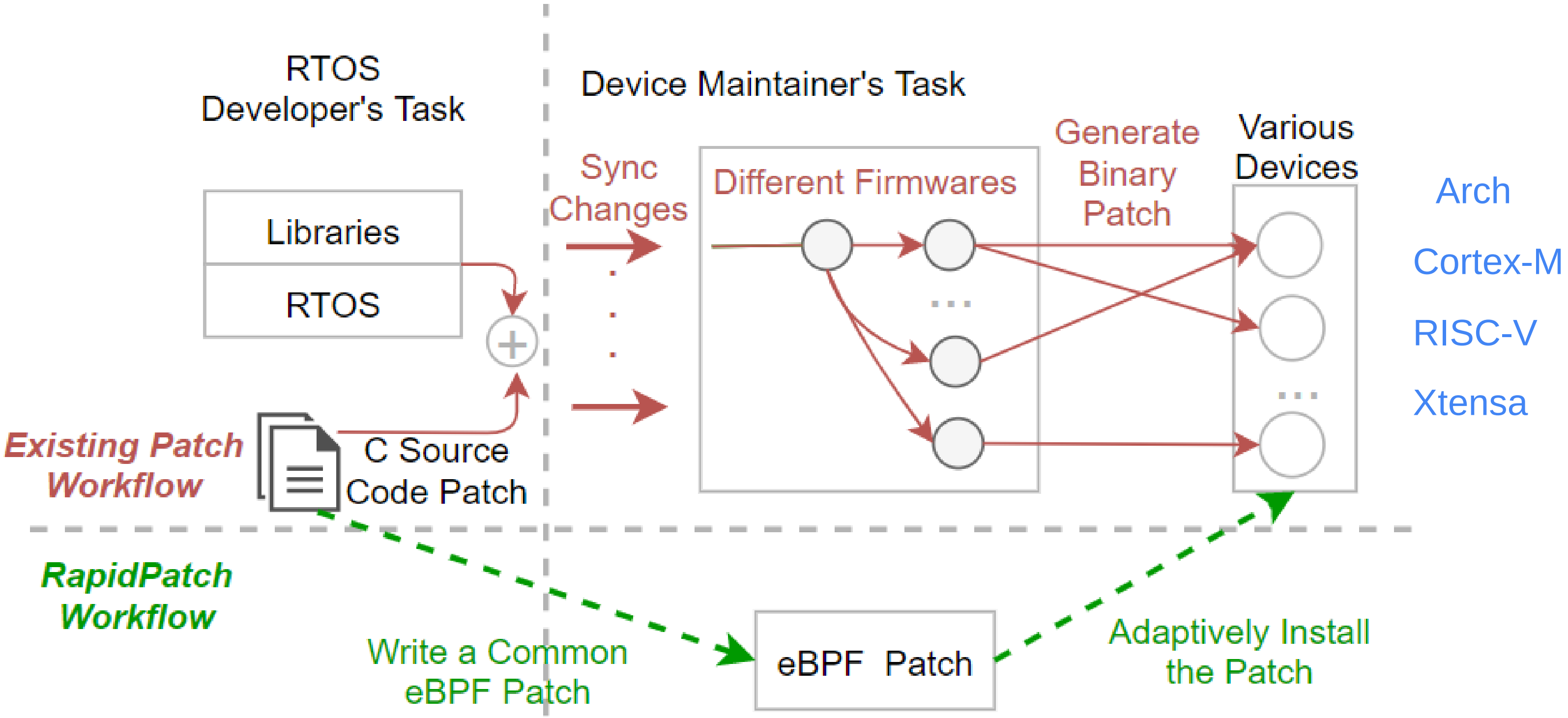
Too many patches need to be tested on the framgement devices

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.

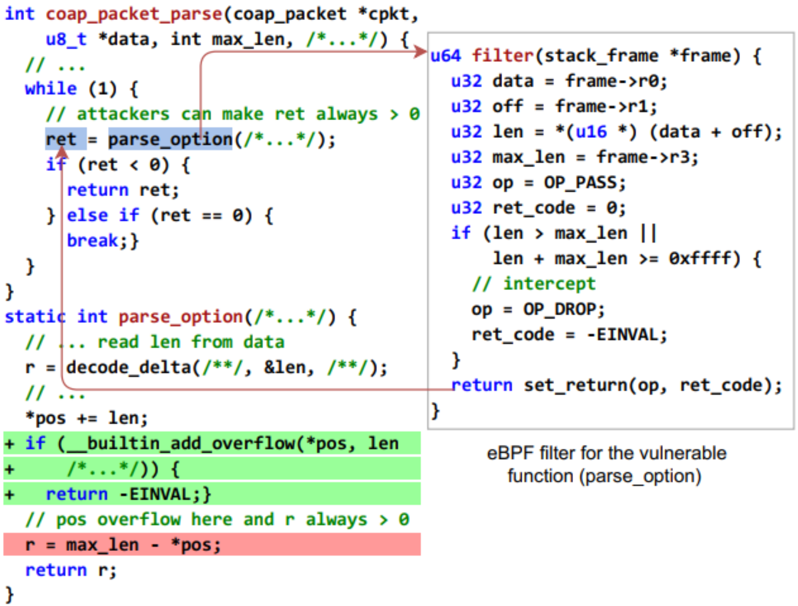
Solutions for patch development

Obstacle-1: Patch Writing

Obstacle-2: Patch Propagation

Obstacle-3: Patch Testing

Obstacle-4: Patch Deploy



```
int coap_packet_parse(coap_packet *cpkt,
    u8_t *data, int max_len, /*...*/) {
  // ...
  while (1) {
    // attackers can make ret always > 0
    ret = parse_option(/*...*/);
    if (ret < 0) {
      return ret;
    } else if (ret == 0) {
      break;}
  }
}
static int parse_option(/*...*/) {
  // ... read len from data
  r = decode_delta(/**/, &len, /**/);
  // ...
  *pos += len;
+ if (__builtin_add_overflow(*pos, len
+     /*...*/)) {
+   return -EINVAL;}
  // pos overflow here and r always > 0
  r = max_len - *pos;
  return r;
}
```

```
u64 filter(stack_frame *frame) {
  u32 data = frame->r0;
  u32 off = frame->r1;
  u32 len = *(u16 *) (data + off);
  u32 max_len = frame->r3;
  u32 op = OP_PASS;
  u32 ret_code = 0;
  if (len > max_len ||
      len + max_len >= 0xffff) {
    // intercept
    op = OP_DROP;
    ret_code = -EINVAL;
  }
  return set_return(op, ret_code);
}
```

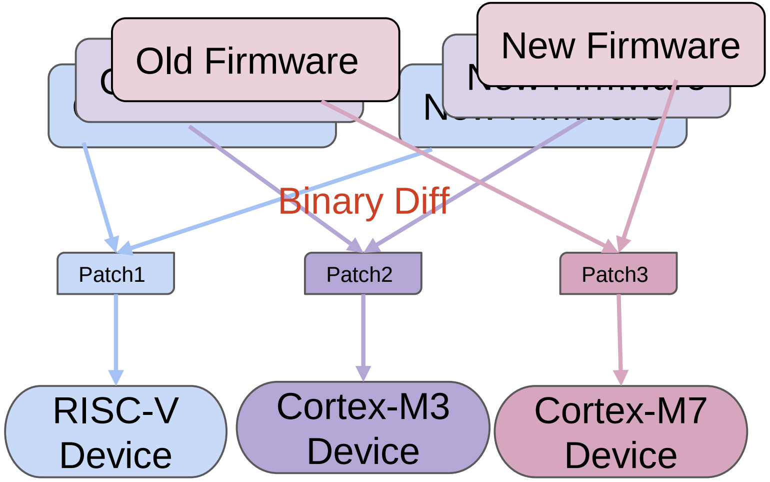
eBPF filter for the vulnerable function (parse_option)

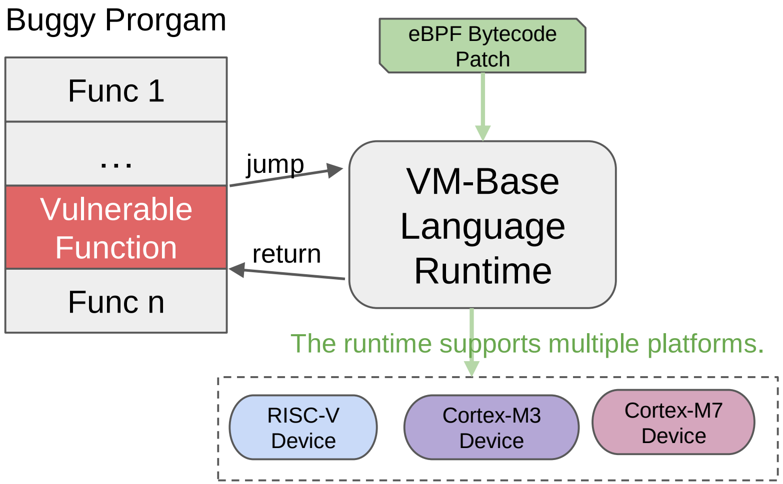
Previous: Merge C source code patch to generate **binary patch** for each type of device

Now: Use a single eBPF **bytecode patch** for all the devices

Binary Patch vs Bytecode Patch

Need to prepare patchs with different architectures.





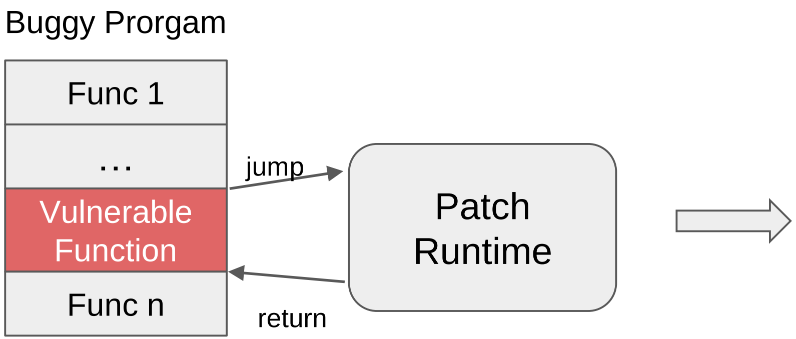
The runtime supports multiple platforms.

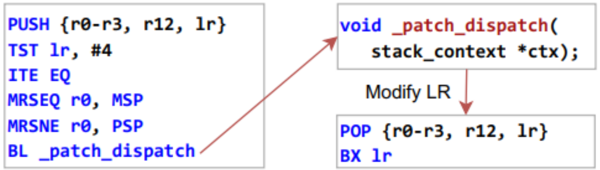
Why use eBPF as the VM-based language? Why not Lua, Python, Javascript, …?

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





```
PUSH {r0-r3, r12, lr}
TST lr, #4
ITE EQ
MRSEQ r0, MSP
MRSNE r0, PSP
BL _patch_dispatch
```

```
void _patch_dispatch(
   stack_context *ctx);
```

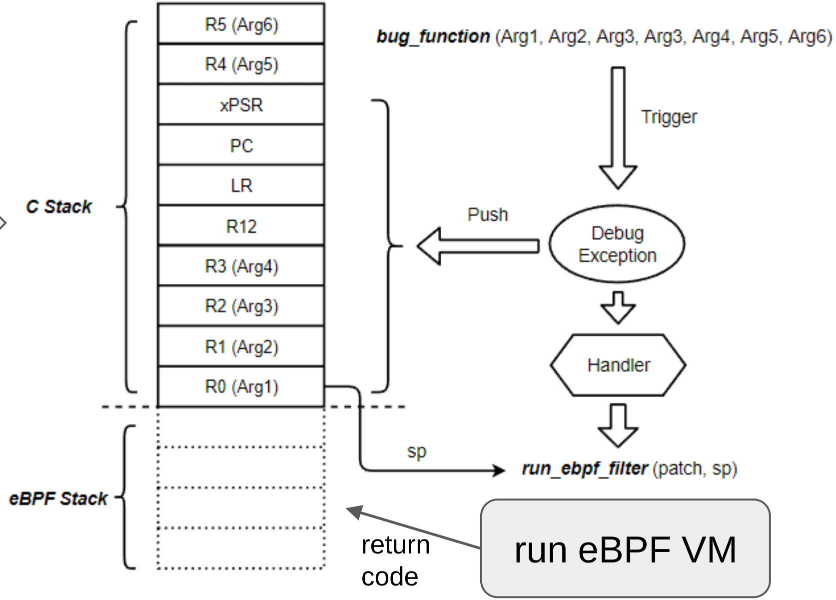
Modify LR

```
POP {r0-r3, r12, lr}
BX lr
```

Save context and switch to the patch dispatch function

Restore context

Switch to and restore from patch runtime



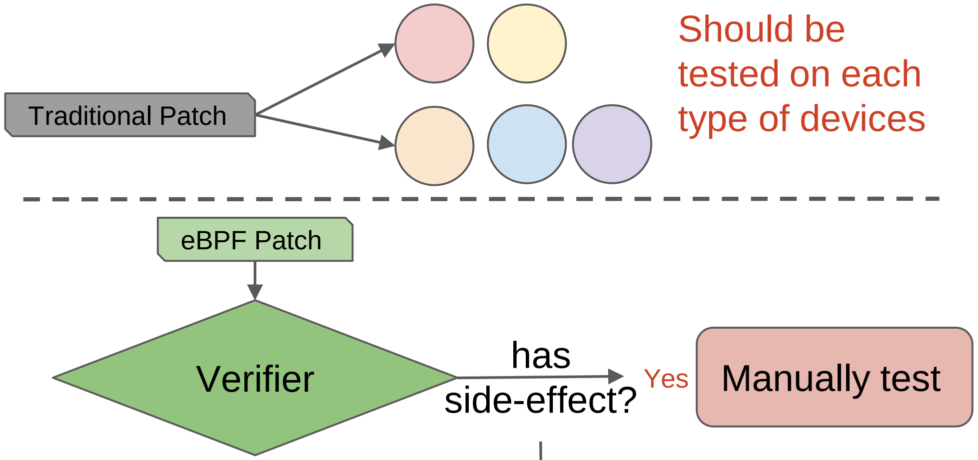
Solutions for patch testing

Obstacle-1: Patch writing

Obstacle-2: Patch Propagation

Obstacle-3: Patch Testing

Obstacle-4: Patch Deploy

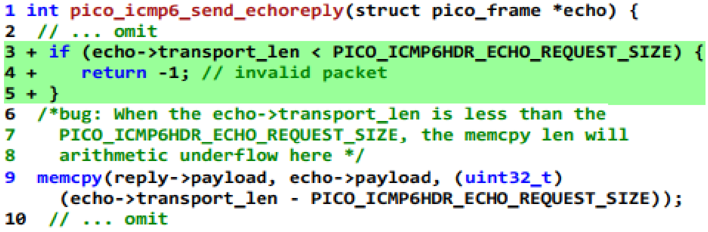


Should be tested on each type of devices

The safety is ensured by software fault isolation (SFI) of the patch runtime.

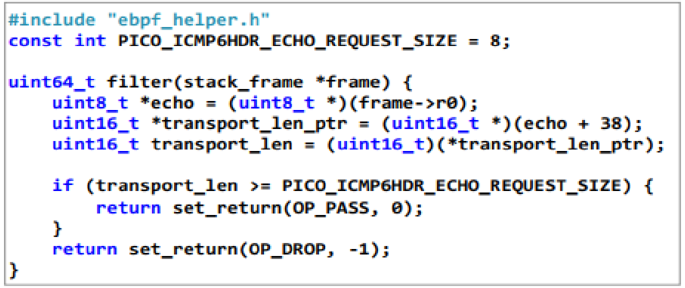
Which patches have no side effects?

Fitler Patch:



```
int pico_icmp6_send_echoreply(struct pico_frame *echo) {
 // ... omit
+ if (echo->transport_len < PICO_ICMP6HDR_ECHO_REQUEST_SIZE) {
+    return -1; // invalid packet
+ }
 /*bug: When the echo->transport_len is less than the
   PICO_ICMP6HDR_ECHO_REQUEST_SIZE, the memcpy len will
   arithmetic underflow here */
 memcpy(reply->payload, echo->payload, (uint32_t)
   (echo->transport_len - PICO_ICMP6HDR_ECHO_REQUEST_SIZE));
  // ... omit
}
```

(a). The C Souce Code Patch

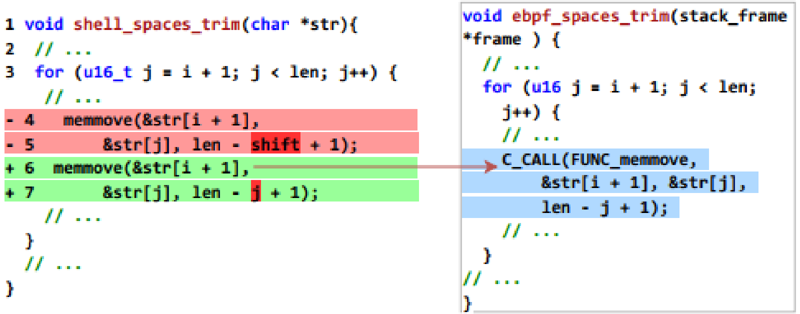


```
#include "ebpf_helper.h"
const int PICO_ICMP6HDR_ECHO_REQUEST_SIZE = 8;

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 set_return(OP_PASS, 0);
    }
    return set_return(OP_DROP, -1);
}
```

(b). The eBPF Filter Patch

Yes. It only reads the memory.

Code-Replace Patch:



```
void shell_spaces_trim(char *str){
  // ...
  for (u16_t j = i + 1; j < len; j++) {
    // ...
-    memmove(&str[i + 1],
-         &str[j], len - shift + 1);
+   memmove(&str[i + 1],
+         &str[j], len - j + 1);
    // ...
  }
  // ...
}
```

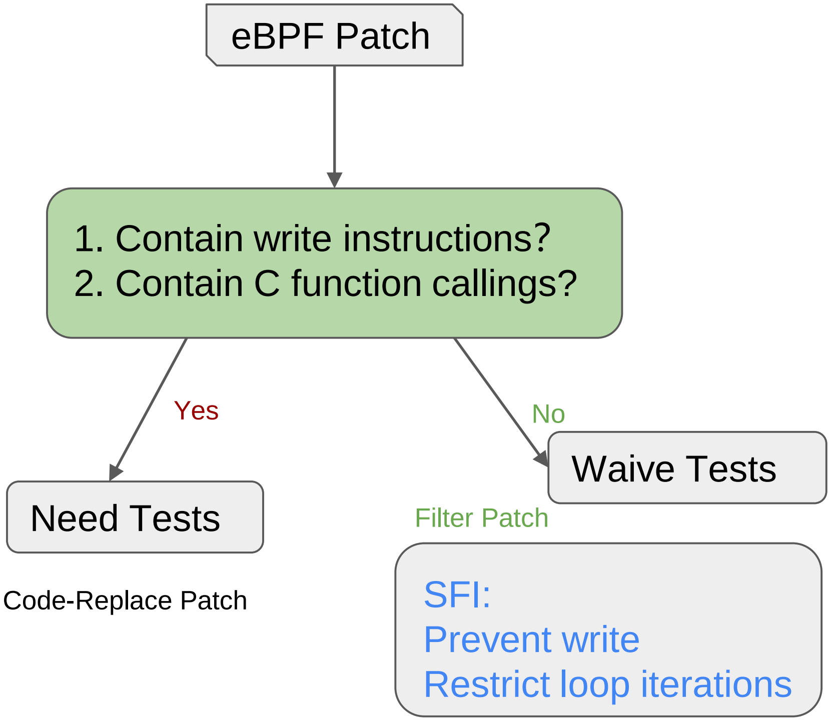
```
void ebpf_spaces_trim(stack_frame
*frame ) {
  // ...
  for (u16 j = i + 1; j < len;
    j++) {
    // ...
    C_CALL(FUNC_memmove,
        &str[i + 1], &str[j],
        len - j + 1);
    // ...
  }
// ...
}
```

(a). The C Source Code Patch

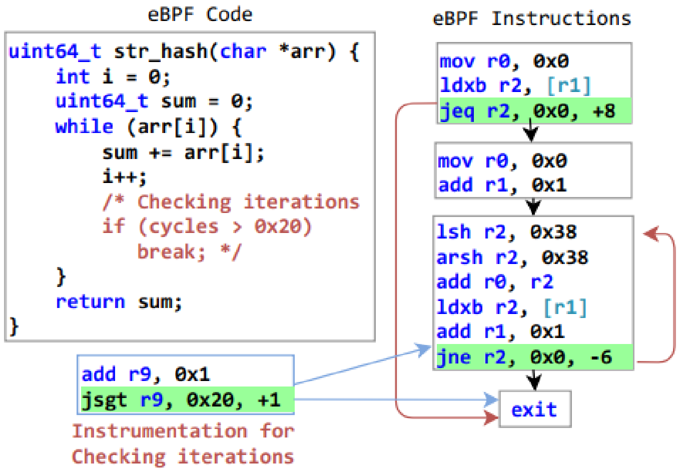
(b). The eBPF Code Replace Patch

No. It calls C functions.

Patch Verifier and SFI



SFI:
Prevent write
Restrict loop iterations



```
uint64_t str_hash(char *arr) {
    int i = 0;
    uint64_t sum = 0;
    while (arr[i]) {
        sum += arr[i];
        i++;
        /* Checking iterations
        if (cycles > 0x20)
           break; */
    }
    return sum;
}
```

```
mov r0, 0x0
ldxb r2, [r1]
jeq r2, 0x0, +8

mov r0, 0x0
add r1, 0x1

lsh r2, 0x38
arsh r2, 0x38
add r0, r2
ldxb r2, [r1]
add r1, 0x1
jne r2, 0x0, -6

exit
```

```
add r9, 0x1
jsgt r9, 0x20, +1
```

Add loop limitation for code with unbounded loops.

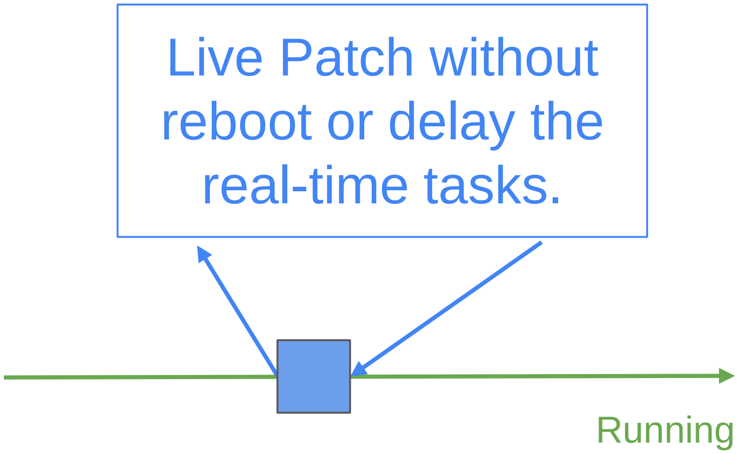
Solutions for patch deploy

Obstacle-1: Patch writing

Obstacle-2: Patch Propagation

Obstacle-3: Patch Testing

Obstacle-4: Patch Deploy



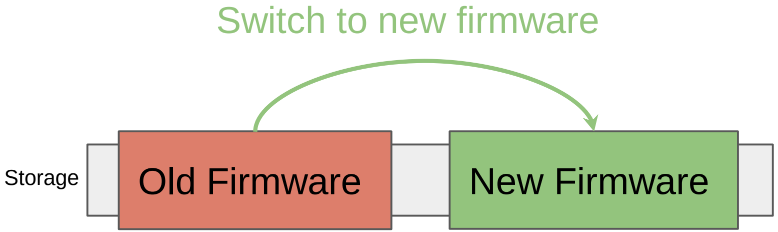
Live Patch without reboot or delay the real-time tasks.

Industry devices

Medical devices

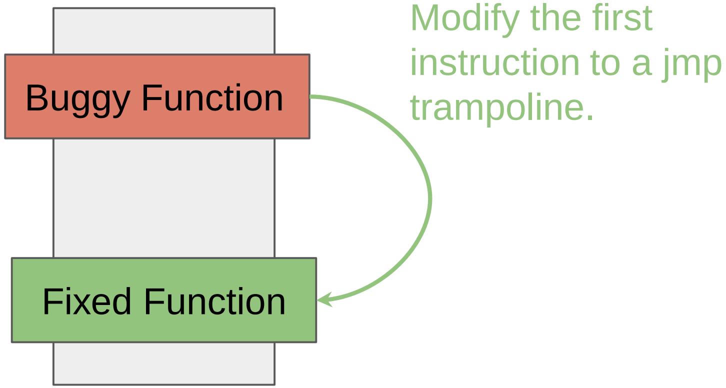
Challenges for hotpatching embedded devices

Approach-1: A/B Scheme



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

Approach-2: Inline Hook

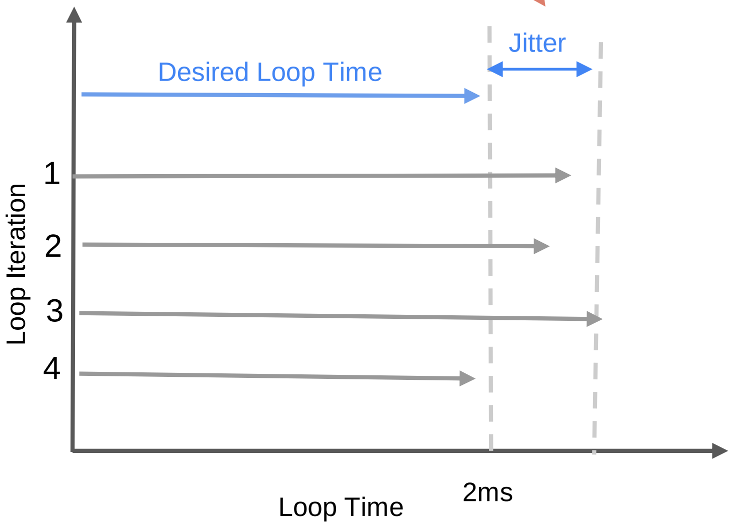


Limitation:
The code of embedded devices are stored in ROM (nor-flash) and modifying is time consuming which break the **real-time constraint**.

What is real-time constraint?

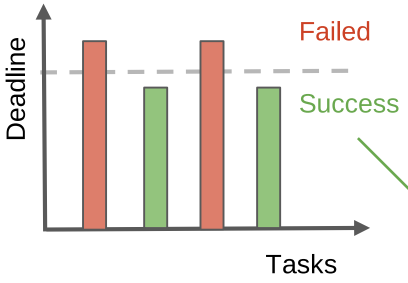
```
while (true) {
        process_task();
        sleep_for_us(2000);
}
```

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



Real-time devices:

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.



Control commands should be executed in time

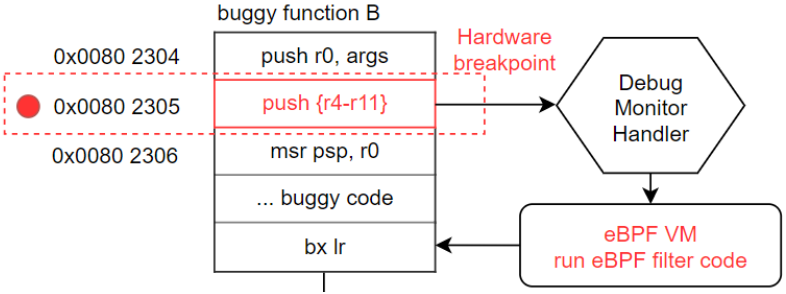
Patch Deploying under real-time constranit

Method-1: Compile-time instrument

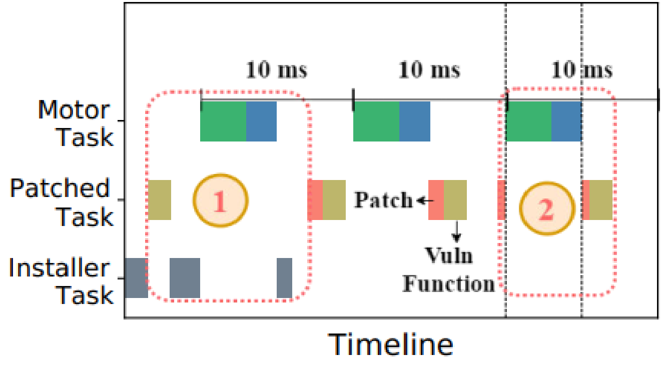
```
int buggy_function() {
            check_has_patch();
            // ...
}
```

Method-2: MCU patching feature

Method-3: hardware breakpoint based KPorbe



	Patch Point	Support Device	#Patch
Fixed Patch Points	Function Begin	All	32+
FPB	Basic Block	Only Cortex-M3/M4	6
KProbe	Basic Block	Cortex-M3∼M55(all), RISC-V	8



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 diffing

4.Test patch on devices

5. Reboot devices to update

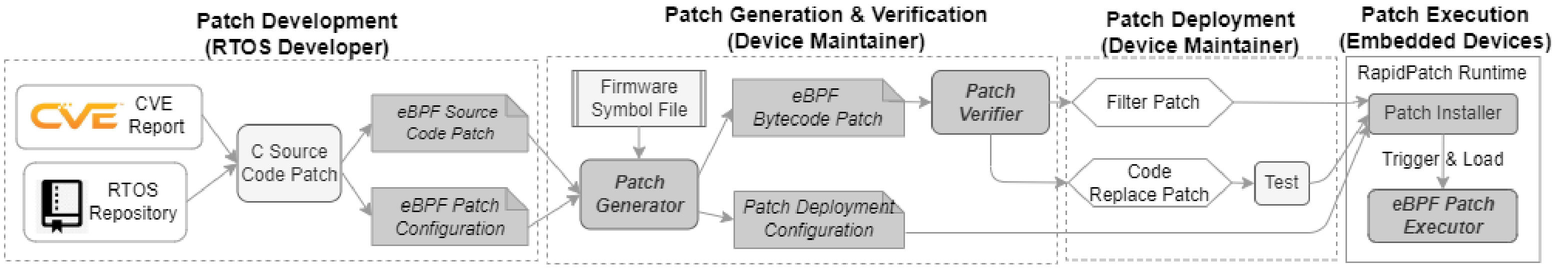
Bytecode Patch

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:

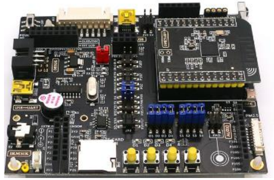
https://github.com/IoTAccessControl/RapidPatch

Module	#LoC
Patch Control	~ 1200 C
Patch Core	~ 2200 C
Libebpf	~ 3400 C
Patch Tool	~ 700 Python
Patch Verifier	~ 1200 Python

IoT Development Kits:

STM32-F429

STM32-L475



nRF52840-DK

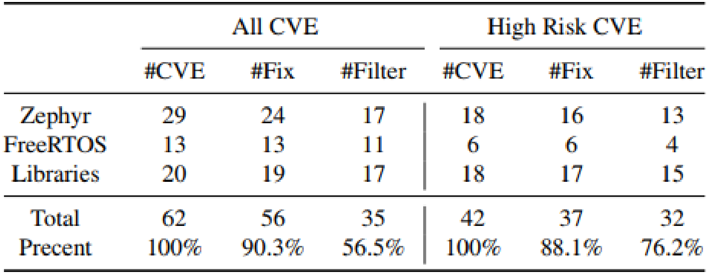
ESP32

Usability Evaluation

90% of the CVEs can be patched.

56% of them do not need test.

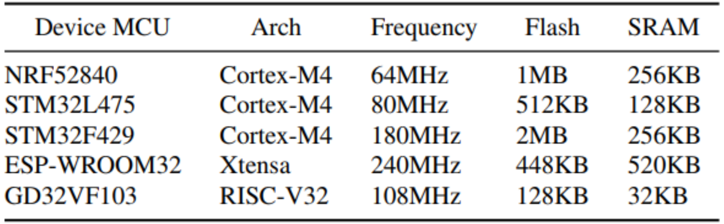
76% of the patches for high risk CVEs do not need test



	All CVE			High Risk CVE		
	#CVE	#Fix	#Filter	#CVE	#Fix	#Filter
Zephyr	29	24	17	18	16	13
FreeRTOS	13	13	11	6	6	4
Libraries	20	19	17	18	17	15
Total	62	56	35	42	37	32
Precent	100%	90.3%	56.5%	100%	88.1%	76.2%

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

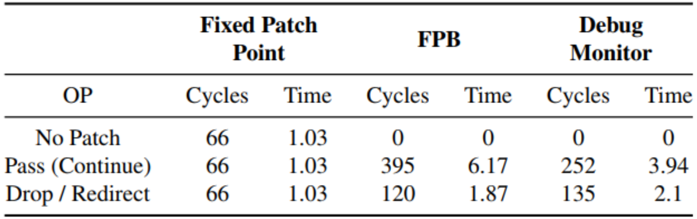


Device MCU	Arch	Frequency	Flash	SRAM
NRF52840	Cortex-M4	64MHz	1MB	256KB
STM32L475	Cortex-M4	80MHz	512KB	128KB
STM32F429	Cortex-M4	180MHz	2MB	256KB
ESP-WROOM32	Xtensa	240MHz	448KB	520KB
GD32VF103	RISC-V32	108MHz	128KB	32KB

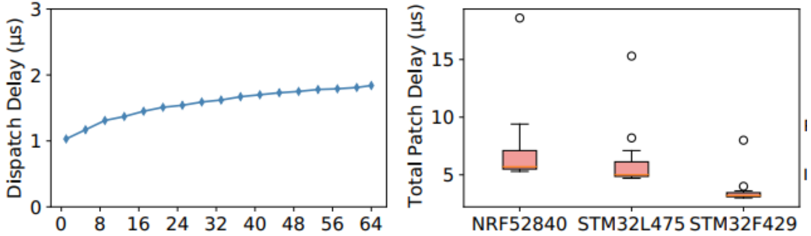
Support various RTOSs, such as NuttX, FreeRTOS, Zephyr, and LiteOS

Patch Runtime Performance Evaluation

Delays incurred by different hotpatching strategies is about 1 ~ 4 us

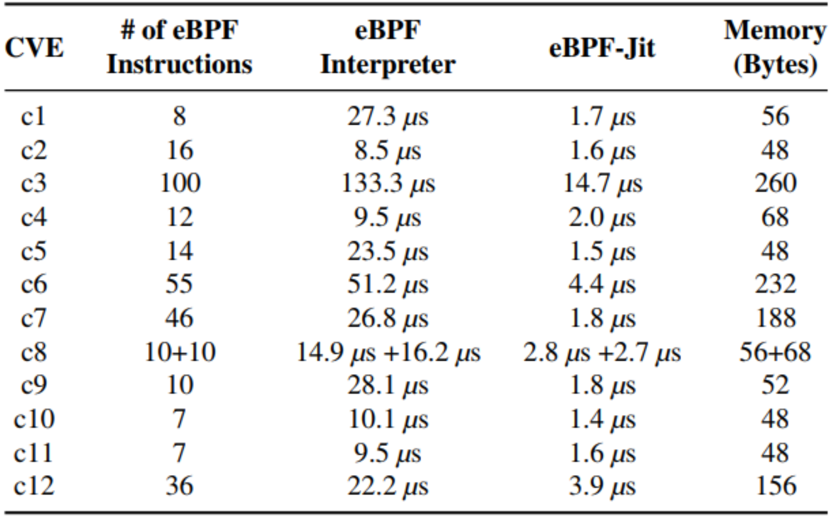


	Fixed Patch Point		FPB		Debug Monitor	
OP	Cycles	Time	Cycles	Time	Cycles	Time
No Patch	66	1.03	0	0	0	0
Pass (Continue)	66	1.03	395	6.17	252	3.94
Drop / Redirect	66	1.03	120	1.87	135	2.1



(a) Dispatch delay of various patch numbers (b) Total patch delay in various device

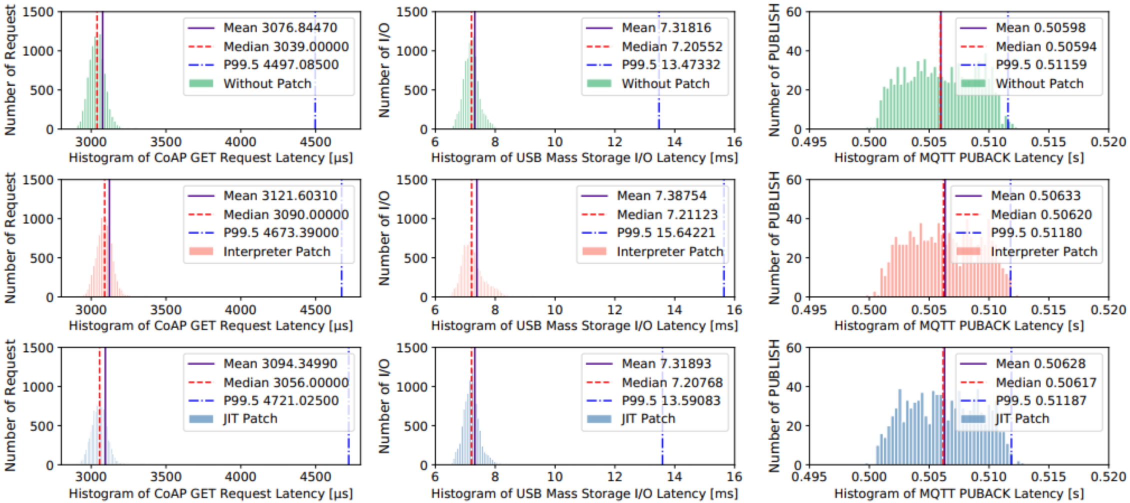
The average delays incurred by eBPF patch execution is less than 5 us



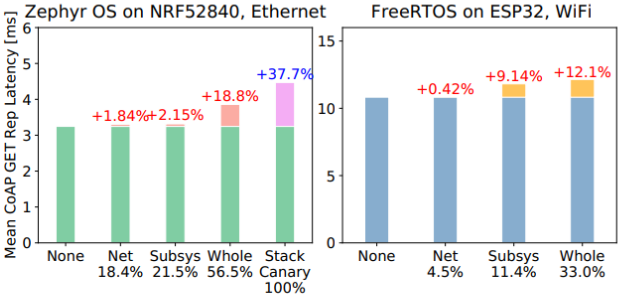
CVE	# of eBPF Instructions	eBPF Interpreter	eBPF-Jit	Memory (Bytes)
c1	8	27.3 μs	1.7 μs	56
c2	16	8.5 μs	1.6 μs	48
c3	100	133.3 μs	14.7 μs	260
c4	12	9.5 μs	2.0 μs	68
c5	14	23.5 μs	1.5 μs	48
c6	55	51.2 μs	4.4 μs	232
c7	46	26.8 μs	1.8 μs	188
c8	10+10	14.9 μs +16.2 μs	2.8 μs +2.7 μs	56+68
c9	10	28.1 μs	1.8 μs	52
c10	7	10.1 μs	1.4 μs	48
c11	7	9.5 μs	1.6 μs	48
c12	36	22.2 μs	3.9 μs	156

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 posssible 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